

MOBILE
PROGRAMMING
SERIES

A-PDF Text Replace DEMO: Purchase from



Covers
iOS 7
and
Xcode 5

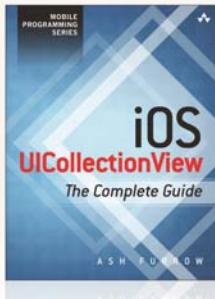
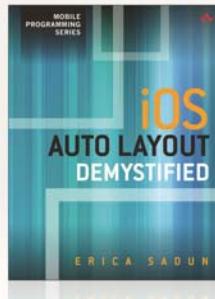
iOS DRAWING

PRACTICAL UIKIT SOLUTIONS

ERICA SADUN

iOS Drawing

Addison-Wesley Mobile Programming Series



▼ Addison-Wesley

Visit informit.com/mobile for a complete list of available publications.

The Addison-Wesley Mobile Programming Series is a collection of digital-only programming guides that explore key mobile programming features and topics in-depth. The sample code in each title is downloadable and can be used in your own projects. Each topic is covered in as much detail as possible with plenty of visual examples, tips, and step-by-step instructions. When you complete one of these titles, you'll have all the information and code you will need to build that feature into your own mobile application.



Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source

▼ Addison-Wesley

Safari
Books Online

ALWAYS LEARNING

PEARSON

iOS Drawing

Practical UIKit Solutions

Erica Sadun



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

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

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

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

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

For sales outside the United States, please contact

International Sales
international@pearsoned.com

Visit us on the Web: informat.com/aw

Copyright © 2014 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

AirPlay, AirPort, AirPrint, AirTunes, App Store, Apple, the Apple logo, Apple TV, Aqua, Bonjour, the Bonjour logo, Cocoa, Cocoa Touch, Cover Flow, Dashcode, Finder, FireWire, iMac, Instruments, Interface Builder, iOS, iPad, iPhone, iPod, iPod touch, iTunes, the iTunes logo, Leopard, Mac, Mac logo, Macintosh, Multi-Touch, Objective-C, Quartz, QuickTime, QuickTime logo, Safari, Snow Leopard, Spotlight, and Xcode are trademarks of Apple, Inc., registered in the United States and other countries. OpenGL, or OpenGL Logo, is a registered trademark of Silicon Graphics, Inc. The YouTube logo is a trademark of Google, Inc. Intel, Intel Core, and Xeon are trademarks of Intel Corp. in the United States and other countries.

ISBN-13: 978-0-321-94787-1
ISBN-10: 0-321-94787-8

Editor-in-Chief

Mark Taub

Senior Acquisitions Editor

Trina
MacDonald

Senior Development Editor

Chris Zahn

Managing Editor

Kristy Hart

Senior Project Editor

Betsy Gratner

Copy Editor

Kitty Wilson

Proofreader

Anne Goebel

Technical Reviewer

Richard Wardell

Editorial Assistant

Olivia Basegio

Cover Designer

Chuti Prasertsith



For my kids. Hang in there, guys! I love you.



Contents at a Glance

- Preface
- 1** Drawing Contexts
- 2** The Language of Geometry
- 3** Drawing Images
- 4** Path Basics
- 5** Paths in Depth
- 6** Drawing Gradients
- 7** Masks, Blurs, and Animation
- 8** Drawing Text
- A** Blend Modes
- B** Miter Threshold Limits

Contents

Preface

- How This Book Is Organized
- About the Sample Code
- Contacting the Author

1 Drawing Contexts

- Frameworks
- When to Draw
- Contexts
- Establishing Contexts in UIKit
- Building Contexts in Quartz
- Drawing into Contexts
- Drawing Within a UIKit Context
- UIKit and Quartz Colors
- The Painter's Model
- Context State
- Context Coordinate System
- Clipping
- Transforms
- Setting Line Parameters
- Summary

2 The Language of Geometry

- Points Versus Pixels
- View Coordinates
- Key Structures
- Using CGRectDivide()
- Rectangle Utilities
- Fitting and Filling
- Summary

3 Drawing Images

- UIKit Images
- Building Thumbnails
- Extracting Subimages
- Converting an Image to Grayscale

Watermarking Images
Retrieving Image Data
Creating Images from Bytes
Drawing and Auto Layout
Building Stretchable Images
Rendering a PDF
Building a Pattern Image
Summary

4 Path Basics

Why Bezier
Class Convenience Methods
Building Paths
Retrieving Path Bounds and Centers
Transforming Paths
Fitting Bezier Paths
Creating Bezier Paths from Strings
Adding Dashes
Building a Polygon Path
Line Joins and Caps
Miter Limits
Inflected Shapes
Summary

5 Paths in Depth

Path Elements
Converting Bezier Paths into Element Arrays
Legal and Illegal Paths
Math Behind the Path
Calculating Path Distance
Interpolating Paths
Retrieving Subpaths
Inverting Paths
Drawing Shadows
Drawing Inner and Outer Glows
Reversing a Path
Visualizing Path Order
Summary

6 Drawing Gradients

- Gradients
- Drawing Gradients
- Building Gradients
- Adding Edge Effects
- State and Transparency Layers
- Flipping Gradients
- Mixing Linear and Radial Gradients
- Drawing Gradients on Path Edges
- Drawing 3D Letters
- Building Indented Graphics
- Combining Gradients and Texture
- Basic Button Gloss
- Adding Bottom Glows
- Building an Elliptical Gradient Overlay
- Summary

7 Masks, Blurs, and Animation

- Drawing into Images with Blocks
- Simple Masking
- Complex Masking
- Blurring
- Drawing Reflections
- Creating Display Links for Drawing
- Building Marching Ants
- Drawing Sampled Data
- Applying Core Image Transitions
- Summary

8 Drawing Text

Drawing Strings

iOS 7 Changes

Text Attributes

Kinds of Attributes

Drawing with Core Text

Drawing Columns

Drawing Attributed Text Along a Path

Fitting Text

Summary

A Blend Modes

B Miter Threshold Limits

Acknowledgments

No book is the work of one person. I want to thank my team, who made this possible. The lovely Trina MacDonald green-lighted this title, so you have the opportunity now to read it. Chris Zahn is my wonderful development editor, and Olivia Basegio makes everything work even when things go wrong.

I send my thanks to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Betsy Gratner, Kitty Wilson, Anne Goebel, and Chuti Prasertsith.

Thanks go as well to Stacey Czarnowski, my new agent now that Neil Salkind retired; to Rich Wardwell, my technical editor; and to my colleagues, both present and former, at TUAW and the other blogs I've worked at.

I am deeply indebted to the wide community of iOS developers who have supported me in IRC and who helped read this book and offer feedback. Particular thanks go to Oliver Drobnik, Aaron Basil (Ethervision), Harsh Trivedi, Michael Prenez-Isbell, Alex Hertzog, Neil Taylor, Maurice Sharp, Rod Strougo, Chris Samuels, Wess Cope, Hamish Allan, Jeremy Tregunna, Mahipal Raythattha, Eddie Kim, Jeremy Sinclair, Jeff Tentschert, Greg Hartstein, Mike Greiner, Ajay Gautam, Shane Zatezalo, Doug Drummond, Wil Macaulay, Mike Kale, Bill DeMuro, Evan Stone, Ashley Ward, Nick Pannuto, Diederik Hoogenboom, Nico Ameghino, Duncan Champney, August Joki, Remy "psy" Demarest, Joshua Weinburg, Emanuele Vulcano, and Charles Choi. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks also go to my husband and kids. You are wonderful.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The Core iOS 6 Developer's Cookbook*, Fourth Edition. She currently blogs at TUAW.com and has blogged in the past at O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in computer science from Georgia Tech's Graphics, Visualization, and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement when they're not busy rewiring the house or plotting global dominance.

Preface

Apple has lavished iOS with a rich and evolving library of 2D drawing utilities. These APIs include powerful features like transparency, path-based drawing, antialiasing, and more. They offer low-level, lightweight, easy-to-integrate drawing routines that you can harness to build images, to display views, and to print.

Importantly, these APIs are resolution independent. With careful thought, your drawing code can automatically adapt to new geometries and screen resolutions. What you draw fits itself to each device and its screen scale, allowing you to focus on the drawing content, not on the specifics of pixel implementations.

This book introduces Quartz and UIKit drawing. From basic painting to advanced Photoshop-like effects, you'll discover a wealth of techniques you can use in your day-to-day development tasks. It's written for iOS 7, so if you're looking for material that reflects iOS's newest incarnation, it's here.

Ready to get started? The power you need is already in iOS. This book helps you unlock that power and bring it into your apps.

—Erica Sadun, October 2013

How This Book Is Organized

This book offers practical drawing how-to for iOS development. Here's a rundown of what you find in this book's chapters:

- **Chapter 1, “Drawing Contexts”**—This chapter introduces the virtual canvases you draw into from your applications. It reviews the core technologies that underlie drawing. Then it dives into contexts themselves, explaining how to build and draw into them and use their content to create images, documents, and custom views. By the time you finish this chapter, you'll have explored the core of iOS drawing.
- **Chapter 2, “The Language of Geometry”**—Drawing and geometry are inextricably linked. In order to express a drawing operation to the compiler, you must describe it with geometric descriptions that iOS can interpret on your behalf. This chapter reviews basics you'll need to get started. It begins with the point–pixel dichotomy, continues by diving into core structures, and then moves to UIKit objects. You'll learn what these items are and the roles they play in drawing.
- **Chapter 3, “Drawing Images”**—This chapter surveys the techniques you need to create, adjust, and retrieve image instances. You'll read about drawing into image instances, creating thumbnails, working with byte arrays, and more.

- **Chapter 4, “Path Basics”**—Paths are some of the most important tools for iOS drawing, enabling you to create and draw shapes, establish clipping paths, define animation paths, and more. Whether you’re building custom view elements, adding Photoshop-like special effects, or performing ordinary tasks like drawing lines and circles, a firm grounding in the `UIBezierPath` class will make your development easier and more powerful.
- **Chapter 5, “Paths in Depth”**—Exposing path internals ratchets up the way you work with the `UIBezierPath` class. This chapter explains how to leverage the `CGPathElement` data structures stored in each instance’s underlying `CGPath` to produce solutions for many common iOS drawing challenges. Want to place objects along a path’s curves? Want to divide a path into subpaths and color them individually? There are element-based solutions for these tasks, which you’ll discover in this chapter.
- **Chapter 6, “Drawing Gradients”**—In iOS, a gradient is a progression between colors. Gradients are used to shade drawings and simulate real-world lighting in computer-generated graphics. Gradients are an important component for many drawing tasks and can be leveraged for powerful visual effects. This chapter introduces iOS gradients and demonstrates how to use them to add UI pizzazz.
- **Chapter 7, “Masks, Blurs, and Animation”**—Masking, blurring, and animation are day-to-day development challenges you experience when drawing. This chapter introduces techniques that enable you to add soft edges, depth-of-field effects, and updates that change over time. This chapter surveys these technologies, introducing solutions for your iOS applications.
- **Chapter 8, “Drawing Text”**—The story of text drawing extends well beyond picking a point on a context and painting some text or transforming a string into a Bezier path. This chapter dives deep into text, covering advanced UIKit and Core Text techniques for drawing, measuring, and laying out strings.
- **Appendix A, “Blend Modes”**—This appendix offers a quick reference for Quartz’s important blend modes that enable you to blend new source material over an already-drawn destination.
- **Appendix B, “Miter Threshold Limits”**—You’ll read here about miter limits, the feature that automatically converts line joins into cropped, beveled results.

About the Sample Code

This book follows the trend I started in my *iOS Developer Cookbooks*. This book’s iOS sample code always starts off from a single `main.m` file, where you’ll find the heart of the application powering the example. This is not how people normally develop iOS or Cocoa applications or, honestly, how they should be developing them, but it’s a great way of presenting a single big idea. It’s hard to tell a story when readers must look through five or

seven or nine files at once, trying to find out what is relevant and what is not. Offering a single file heart concentrates that story, allowing access to that idea in a single chunk.

The power of this book, however, lies in routines collected into the “Quartz Book Pack” in the sample code projects. These are standalone classes and functions intended to inspire use outside this book. I do recommend, however, that you add your own namespacing to avoid any potential conflict with future Apple updates. They include concise solutions that you can incorporate into your work as needed.

For the most part, the examples for this book use a single application identifier: com.sadun.hello-world. This avoids clogging up your iOS devices with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples simultaneously, simply edit the identifier by adding a unique suffix, such as com.sadun.helloworld.sample5.

You can also edit the custom display name to make the apps visually distinct. Your Team Provisioning Profile matches every application identifier, including com.sadun.helloworld. This allows you to install compiled code to devices without having to change the identifier; just make sure to update your signing identity in each project’s build settings.

Getting the Sample Code

You’ll find the source code for this book at github.com/erica/iOS-Drawing on the open-source GitHub hosting site. There you’ll find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book.

If you do not feel comfortable using git directly, you can use the GitHub download button. It was at the left of the page at the time this book was being written. It allows you to retrieve the entire repository as a ZIP archive.

Contribute!

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

Getting Git

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

Getting GitHub

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

Contacting the Author

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

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

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

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

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

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

E-mail: trina.macdonald@pearson.com
Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

This page intentionally left blank

Drawing Contexts

Drawing contexts are the virtual canvases you draw into from your applications. In this chapter, you review core technologies that underlie iOS drawing. You dive into contexts, discovering how to build and draw into them. You use contexts to create images, documents, and custom views, learning about basic drawing fundamentals in UIKit, Core Graphics, and Quartz.

Frameworks

iOS's drawing routines are primarily sourced from the UIKit and QuartzCore frameworks. They consist of a mix of modern Objective-C interfaces (from UIKit) along with older C-based functions and Core Foundation-style objects (from QuartzCore). These items live side by side in your code. Here's an example that shows a UIKit drawing operation followed by a Quartz one:

```
// Draw a rounded rectangle in UIKit
UIBezierPath *bezierPath =
    [UIBezierPath bezierPathWithRoundedRect:inset cornerRadius:12];
[bezierPath stroke];

// Fill an ellipse in Quartz
CGContextFillEllipseInRect(context, rect);
```

The QuartzCore framework is commonly referred to as Quartz or Quartz 2D. The latter is Apple's official name for its native 2D rendering and antialiasing API. It's also the name used for the primary drawing reference in Apple's developer library, the "Quartz 2D Programming Guide."

The reach of this framework is wider than the Quartz 2D name might suggest. The QuartzCore framework implements compositing as well as rendering, along with many other graphics features. For example, the QuartzCore APIs include animation, tiling, and Core Image filtering (even though Core Image is properly its own framework).

To iOS developers, Quartz is often better known by its internal implementation name, Core Graphics. This book uses the terms Quartz and Core Graphics synonymously throughout. Most of the C-based APIs start with the two-letter CG prefix, derived from the Core Graphics name. From geometry (`CGRect` and `CGPoint`) to objects (`CGColorRef`, `CGImageRef`, and `CGDataProviderRef`), Quartz offers a vast, rich trove of drawing technology.

Quartz and its predecessors have been around for a long time. Contributing elements date back to the 1980s, when Display PostScript powered the graphics on the NeXTStep operating system. Quartz still uses an internal imaging model that's very similar to modern PDF.

With each iOS release, updates continue to move the drawing APIs further toward Objective-C, simplifying, enhancing, and refining programmatic drawing. Core Graphics functions, however, continue to play an important role in day-to-day drawing development. Although newer routines replace many common tasks, they haven't supplanted them all. Be prepared to work with both frameworks for the foreseeable future.

When to Draw

Although iOS drawing is a fairly common task for developers, the task itself is not very general. Most drawing is limited to certain specific arenas, namely four extremely common scenarios where it makes sense to work directly with graphics: creating custom views, building images, creating PDFs, and building with Core Graphics.

Creating Custom Views

Every UIKit view is, essentially, a blank canvas. You can fully customize a view by drawing whatever contents best express the view's role in your application. You do this by drawing that content in a special method called `drawRect:`. This method enables you to customize a view's look by calling UIKit and Quartz drawing routines.

Figure 1-1 shows a custom color control. It consists of a deck of swatch views, each of which is a `UIView` subclass that implements the `drawRect:` method. These views draw the border with its rounded edges, the informational text, and the splash of color to create a fully realized, yet fully custom, look.

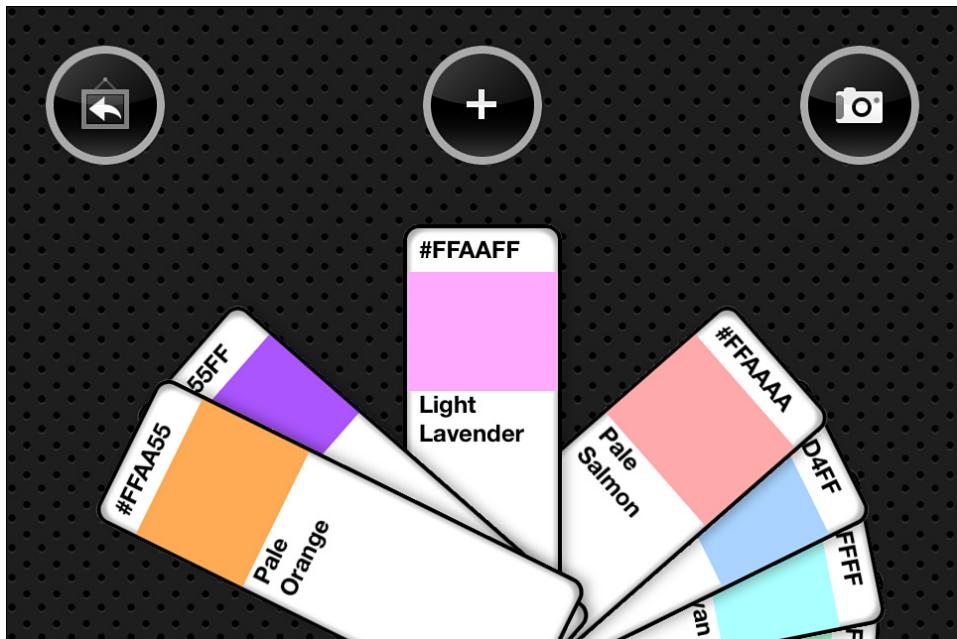


Figure 1-1 These swatches are drawn by custom `drawRect:` methods.

Unless you override it in a subclass, the default `drawRect:` method basically does nothing. Subclasses that create a presentation using UIKit and Core Graphics routines override this method. They add their drawing requests to their subclass implementation. Simple views that merely update a background color or that supply content in other ways apart from drawing should not override `drawRect:`. Similarly, OpenGL ES-powered views do not use this entry point for drawing.

The `drawRect:` method has a close cousin that is meant only for printing: `drawRect:forViewPrintFormatter:` allows you to customize content that should print differently than it displays.

Building Images

Not every image starts its life as a PNG or JPEG file. On iOS, you can draw into a UIKit image context and retrieve a `UIImage` instance. This enables you to create new images or modify existing images.

Figure 1-2 shows a programmatically constructed color wheel. This image was created via a series of colored Bezier path arcs drawn into a UIKit image context. The resulting image was then added to a standard image view. Drawing allows you to build custom images as you need them without having to rely on a preexisting library of image files.

Code-based drawing makes an important trade-off. Although you need greater processing time to create images (not a huge amount, but it's measurable), you end up with a slimmer application bundle, with fewer required resources. Your images are far more flexible—limited only to the code you use to create them.

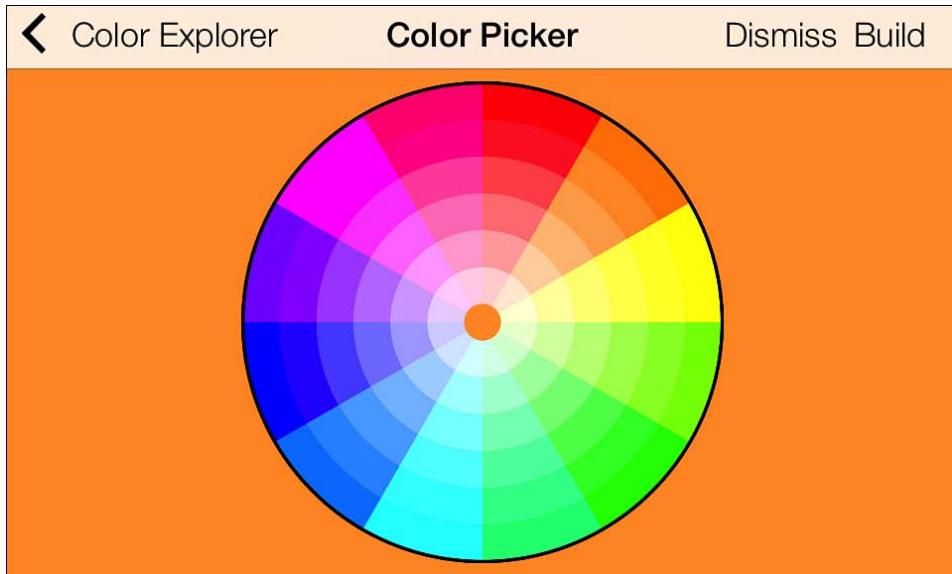


Figure 1-2 This color wheel was built from code into a custom `UIImage`.

Creating PDFs

The same kinds of calls that help you build images also support PDF creation. You can draw into a UIKit PDF context, which is either sent directly to a file or saved to data. This enables you to build PDF content from your apps and then share them, store them, or display them, as in Figure 1-3.

PDFs provide a highly portable, system-independent standard that encapsulates a complete document description. The document you create on iOS looks the same, more or less, on any computer you use to view it. The operating system's color management system may affect color presentation.

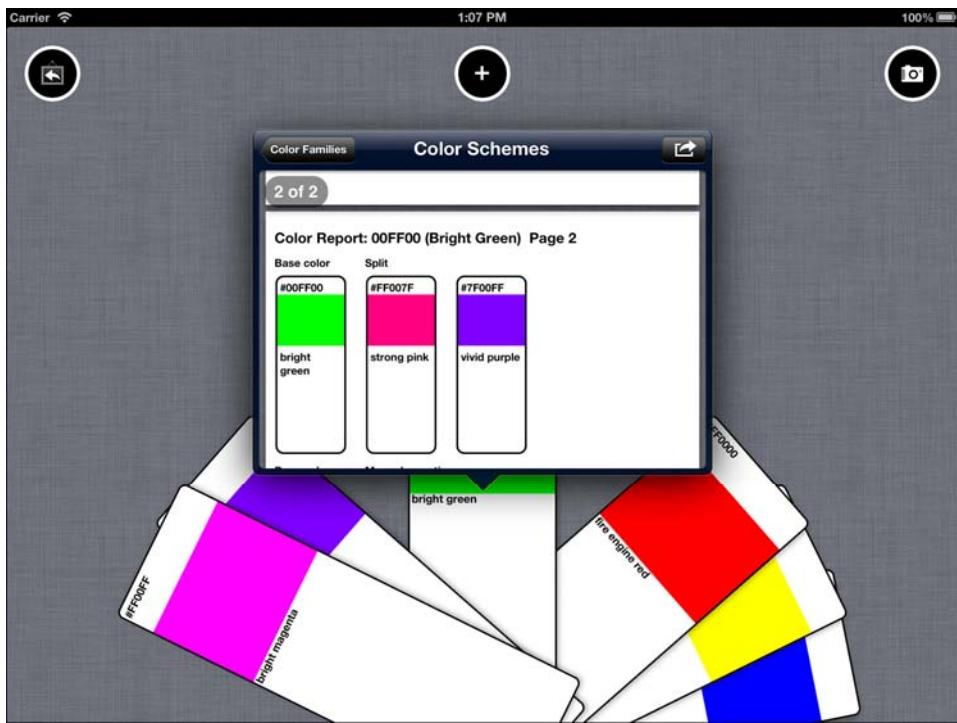


Figure 1-3 A multipage PDF is previewed in this iOS 6 popover.

Building with Core Graphics

When you hit the edges of UIKit's current capabilities, you can fall back to standard Quartz. Core Graphics has been around a long time, and its drawing features have powered OS X for a decade. Any feature that isn't immediately tweakable in UIKit is almost always available through Quartz.

Drawing items into a Core Graphics context provides flexible and powerful drawing solutions—even if they're not quite as simple as their UIKit siblings. Core Graphics uses Core Foundation-style C-based objects and requires manual retain and release development strategies.

For example, you might want to access image data on a byte-by-byte level. That's a task that isn't well handled by UIKit (yet!) but is perfect for Core Graphics's bitmap contexts. Figure 1-4 shows an example of why you might turn to Quartz functions. In this example, the RGB image on the left is rendered into a grayscale color space using Core Graphics.

The resulting images must be transformed from Quartz `CGImageRef` (CG types that end with `Ref` are pointers to objects) into `UIImage` instances (`imageWithCGImage:`) and displayed in a standard image view.



Figure 1-4 The routine that transformed this RGB image to a grayscale representation was written using Core Graphic primitives.

Contexts

Every iOS drawing operation begins with a context. Conceptually, contexts are very similar to blank pages of paper or empty canvases. They represent an iOS destination for drawing. They contain all the information about the state of the drawing medium—for example, whether the canvas is rotated or transformed in some way, what kind of colors can be drawn onto the canvas, the degree of detail you can paint at any point, and so forth.

In iOS, you primarily work with two kinds of drawing contexts: bitmap contexts and PDF contexts. The Core Image framework offers a third context type, which is used for performing image processing tasks rather than drawing.

Bitmap Contexts

Bitmap contexts are essentially rectangular arrays of data. The size of that data depends on the kinds of color each picture element (or “pixel”) represents. *Device RGB*, as shown in the left image of Figure 1-4, uses three or four bytes per pixel, depending on whether the bitmap is opaque (3 bytes) or not (4 bytes).

An opaque bitmap ignores translucency values, optimizing storage. Translucent images use what’s called an *alpha* value. This value is stored in a separate byte from the actual color or luminance information. It refers to each pixel’s translucency. The color information for Device RGB is stored in 3 bytes, each corresponding to a single red, green, or blue level.

Device Gray images, as shown on the right in Figure 1-4, use 1 or 2 bytes per pixel. One luminance byte is stored for each pixel and, optionally, one transparency byte.

PDF Contexts

From a developer’s point of view, PDF contexts work in much the same way as bitmap contexts. You draw to them using identical commands and functions. You set colors and draw shapes and text just as if you were drawing in a view or to an image. There are, however, differences.

PDF contexts contain vector data in their “backing store,” describing the drawing in a resolution-independent manner. Bitmap contexts are rasterized. They use pixel arrays to store the data drawn into them.

PDF contexts also may contain more than one page. You establish a bounding rectangle that specifies the default size and location of each PDF page. An empty rectangle (`CGRectZero`) defaults to a standard A (letter) page. That is 8.5 by 11 inches, or 612 by 792 points. (Chapter 2 discusses the difference between points and pixels.)

PDF drawings are stored internally as vector-based command sequences. This provides an inherent resolution independence that you don’t see in bitmap drawings. Apple writes in its documentation, “PDF files are resolution independent by nature—the size at which they are drawn can be increased or decreased infinitely without sacrificing image detail. The user-perceived quality of a bitmap image is tied to the resolution at which the bitmap is intended to be viewed.”

You draw into these contexts just like you’d draw into bitmap contexts. The differences primarily lie in their destination (files and data representations) and when you start a new page.

In addition to bitmap and PDF contexts, you may also encounter Core Image contexts.

Core Image Contexts

The Core Image framework helps you process images extremely quickly. It supplies routines that apply digital image processing and computer vision to image sources. With it,

you can apply filters, chain filters together, implement feature detection (to find faces and eyes), and analyze images to produce auto-adjustments.

Core Image contexts are image contexts specific to rendering Core Image objects to Quartz 2D and OpenGL. Accelerated by the onboard graphics processing unit (GPU), Core Image contexts integrate with Core Video pixel buffers. Core Image uses its own style of colors (`CIColor`) and images (`CIImage`), which have been optimized for Core Image's fast filtering and image processing features.

Establishing Contexts in UIKit

UIKit provides friendly entry points for building contexts. The simplest image drawing pattern is shown in Listing 1-1. This involves nothing more than starting a new context (you specify the size of that context), drawing to it, retrieving a new image, and ending the context. This pattern produces images at a base 1:1 scale, where every drawing operation corresponds to exact pixels within the bitmap context.

Listing 1-1 Creating a Basic Bitmap Context in UIKit

```
UIGraphicsBeginImageContext(size);  
  
// Perform drawing here  
  
UIImage *image =  
    UIGraphicsGetImageFromCurrentImageContext();  
UIGraphicsEndImageContext();
```

Device Scale

Listing 1-2 shows a more sophisticated (and recommended) entry point. It enables you to build images that respect device scale. The `withOptions` variant used here to create the bitmap context specifies the scale for drawing the image. *Scale* refers to the relationship between logical space (measured in points) and physical displays (measured in pixels).

An image created with a scale of 2 produces Retina-ready graphics. The pixel extent doubles in each direction, producing an image that stores four times the data compared to an image drawn at the base scale of 1. The `opacity` parameter allows you to ignore the alpha channel to optimize bitmap storage. You need only 3 bytes per pixel versus 4 with alpha.

Listing 1-2 Context Options Enable Drawing at Device Scale

```
UIGraphicsBeginImageContextWithOptions(  
    targetSize, isOpaque, deviceScale);
```

```
// Perform drawing here

UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
```

I strongly encourage you to use the approach shown in Listing 1-2 for images you use in your application. For example, when drawing an image for a `UIImageView`, always use the `with-options` approach.

You can supply a screen scale to the third parameter by querying the `UIScreen` class. For images drawn to the primary screen, use the `mainScreen` method, as in this example:

```
UIGraphicsBeginImageContextWithOptions(
    targetSize, isOpaque, [UIScreen mainScreen].scale);
```

You may also pass `0.0` as the third parameter. This automatically determines the scaling, based on the device's main screen.

When working with video out, where you have connected a monitor to an iOS device using a cable or when you're using AirPlay to wirelessly present a second screen, use the scale of that particular destination. Query `UIScreen` for that information.

Creating PDF Contexts

With PDF context creation, you must draw either to a file path or to a mutable data object, which is supplied to a data consumer. Listing 1-3 shows an example of the former. You supply a bounds rectangle (pass `CGRectZero` for an A-sized page) and a dictionary where you specify metadata and security information for the new document. For example, you might specify the author, owner and user passwords, printing and copying permission, keywords, and so on.

As with the bitmap context, Core Graphics implementations (specifically, `CGPDFContext`) have been around a lot longer than their UIKit equivalents. If you want to dive into the C-based classes and functions, you'll discover another take on PDF production.

Listing 1-3 Drawing an Image into a PDF File

```
UIGraphicsBeginPDFContextToFile(pathToFile, theBounds, documentInfo);
UIGraphicsBeginPDFPage();

// Perform drawing here

UIGraphicsEndPDFContext();
```

You begin each page of drawing by calling `UIGraphicsBeginPDFPage()`. This establishes a new page in the document for you to write to. You do not have to end each page explicitly, but you do have to end the PDF context, just as you end any other context, when you are done creating your output.

Building Contexts in Quartz

Core Graphics enables you to build bitmap contexts without using the UIKit entry points. This approach uses an older API set and is helpful when you need to access drawing data on a byte-by-byte basis. Chapter 3 uses Quartz-based contexts to power several of its image-processing examples.

Listing 1-4 showcases the calls involved. These highlight the greater complexity of using Quartz compared to UIKit. That's because, among other things, Quartz uses the older-style Core Foundation system of objects, with their manual retain and release patterns.

Always run the Static Analyzer (Product > Analyze in Xcode) on Quartz code to check whether your references are properly released. It provides a source code analysis tool that discovers potential bugs in your iOS code for both UIKit methods and Quartz functions. Read more about the Clang analyzer on the LLVM website, at <http://clang-analyzer.llvm.org>.

In Listing 1-4, notice the iterative nature of the release patterns. If the context cannot be created, the color space must be freed. At each stage of Core Graphics drawing, you accumulate a trail of allocated objects, which you must properly manage before returning control from a method or function.

As a final point, note how this example uses `kCGImageAlphaPremultipliedFirst`. This specifies an ARGB byte order, using Quartz-friendly alpha premultiplication. For each pixel, the alpha value is stored in the first of 4 bytes and the blue value in the last. This arrangement is documented in the `CGImageAlphaInfo` definition in the `CGImage.h` header file.

Listing 1-4 Building an Image Using Core Graphics Calls

```
// Create a color space
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
if (colorSpace == NULL)
{
    NSLog(@"Error allocating color space");
    return nil;
}

// Create the bitmap context. (Note: in new versions of
// Xcode, you need to cast the alpha setting.)
CGContextRef context = CGBitmapContextCreate(
```

```

        NULL, width, height,
        BITS_PER_COMPONENT, // bits = 8 per component
        width * ARGB_COUNT, // 4 bytes for ARGB
        colorSpace,
        (CGBitmapInfo) kCGImageAlphaPremultipliedFirst);
if (context == NULL)
{
    NSLog(@"Error: Context not created!");
    CGColorSpaceRelease(colorSpace );
    return nil;
}

// Push the context.
// (This is optional. Read on for an explanation of this.)
// UIGraphicsPushContext(context);

// Perform drawing here

// Balance the context push if used.
// UIGraphicsPopContext();

// Convert to image
CGImageRef imageRef = CGBitmapContextCreateImage(context);
UIImage *image = [UIImage imageWithCGImage:imageRef];

// Clean up
CGColorSpaceRelease(colorSpace );
CGContextRelease(context);
CFRelease(imageRef);

```

Drawing into Contexts

Many Quartz functions depend on referencing a context that you can draw into. For example, consider the function calls in Listing 1-5. These set a 4-pixel-wide line width, set a gray color, and then stroke an ellipse within a rectangular container. Each function call requires a `context` parameter, which must be a valid `CGContextRef`. You can build this context yourself (as in Listing 1-4) or retrieve one from UIKit, which is explored in the next section.

Listing 1-5 Drawing an Ellipse

```

// Set the line width
CGContextSetLineWidth(context, 4);

```

```
// Set the line color  
CGContextSetStrokeColorWithColor(context,  
    [UIColor grayColor].CGColor);  
  
// Draw an ellipse  
CGContextStrokeEllipseInRect(context, rect);
```

The code in Listing 1-5 naturally lives where Listing 1-4 says, *Perform drawing here*. At this point in Listing 1-4, you have fully created a bitmap context reference, and you can use that reference with these drawing requests.

By the end of Listing 1-4, you've created an image and manually released the context. Figure 1-5 shows the output created by merging Listing 1-5 into Listing 1-4. The image you produce is a gray ellipse, stroked with a 4-pixel-wide line. Figure 1-5 shows that image displayed in a `UIImageView` instance.

This entire process is performed outside the auspices of UIKit. The only UIKit call is `imageWithCGImage:`, which converts the newly created `CGImageRef` to a `UIImage` instance.

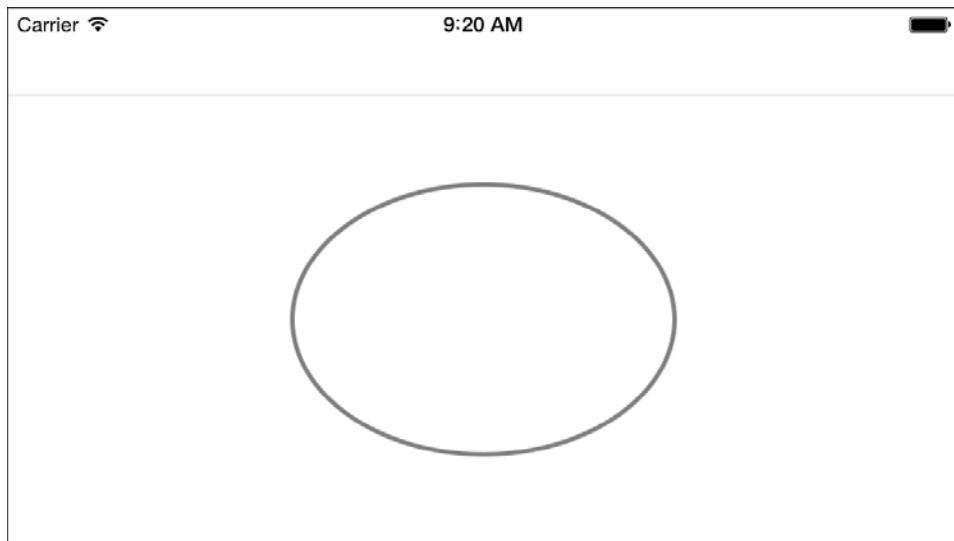


Figure 1-5 Core Graphics functions require a context to draw this ellipse.

Drawing Within a UIKit Context

UIKit simplifies the task of creating and managing contexts. It takes just one line to establish a new image or PDF context, and another to wrap things up. Between those lines, you are free to apply any drawing requests. These are applied to the current context.

Listing 1-6 applies the same drawing functions as in Listing 1-5, and it produces an identical image result. However, instead of drawing to a custom Quartz context, these updates are drawn to a newly established image context. In order to bridge between Quartz and UIKit, you call `UIGraphicsGetCurrentContext()`. This returns a `CGContextRef`, enabling you to use that value in your Core Graphics drawing calls.

Once again, when you compare the number of lines of code in Listing 1-6 with the combination of Listings 1-4 and 1-5, you see how much simpler UIKit drawing can be.

Listing 1-6 Drawing an Ellipse Within a UIKit Image Context

```
// Establish the image context
UIGraphicsBeginImageContextWithOptions(
    targetSize, isOpaque, 0.0);

// Retrieve the current context
CGContextRef context = UIGraphicsGetCurrentContext();

// Perform the drawing
CGContextSetLineWidth(context, 4);
CGContextSetStrokeColorWithColor(context,
    [UIColor grayColor].CGColor);
CGContextStrokeEllipseInRect(context, rect);

// Retrieve the drawn image
UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();

// End the image context
UIGraphicsEndImageContext();
```

You can call `UIGraphicsGetCurrentContext()` after starting any image or PDF context. That context persists until you call `UIGraphicsEndImageContext()` or `UIGraphicsEndPDFContext()`.

Similarly, you can establish a context outside these routines by calling `UIGraphicsPushContext(context)`. Supply a manually built Quartz context like the one created in Listing 1-4. You end that context with `UIGraphicsPopContext()`. These calls establish the context you’re drawing to inside the UIKit context stack. Pushing and popping the stack enables you to temporarily switch between drawing destinations as needed.

Otherwise, outside an explicit context environment, the current context is `nil`—with one exception. Upon calling `drawRect:`, views push a context onto the UIKit graphics context stack. So if you are implementing the `drawRect:` method, you can assume that there's always a valid context that you can retrieve:

```
- (void) drawRect: (CGRect) rect
{
    // Perform drawing here
    // If called, UIGraphicsGetCurrentContext()
    // returns a valid context
}
```

UIKit Current Context

In Quartz, nearly every drawing function typically requires a context parameter. You pass it explicitly to each function. For example, you might set the stroke color to gray:

```
CGContextSetStrokeColorWithColor(context, [UIColor grayColor].CGColor);
```

In UIKit drawing, context is established from runtime conditions. Consider Listing 1-7. This code once again builds the same 4-pixel-wide ellipse shown in Figure 1-5. However, nowhere does this code explicitly reference a context.

This listing creates an elliptical path. It sets the line width to 4, establishes a gray color for stroking, and then strokes the path. In each step, the context is accessed on your behalf. As with Listing 1-6, the same gray color is applied to the current context, but that context is not specified. Listing 1-7 does this without mentioning the context.

Listing 1-7 Drawing an Ellipse Within a UIKit Image Context

```
// Stroke an ellipse using a Bezier path
UIBezierPath *path = [UIBezierPath bezierPathWithOvalInRect:rect];
path.lineWidth = 4;
[[UIColor grayColor] setStroke];
[path stroke];
```

What's happening is that UIKit owns a stack of graphics contexts. It applies drawing operations to whatever context is at the top of that stack. The settings that update the context state to 4 for line width and gray for stroke color apply to that topmost context.

So how do you integrate Core Graphics contexts with UIKit? UIKit provides two key functions, which were briefly mentioned earlier in the chapter:

- You manually push a context by calling `UIGraphicsPushContext(context)`. This function pushes the context onto the UIKit stack and sets the active context you'll be drawing to.

- You balance this by calling `UIGraphicsPopContext()`. This pops the current context off the UIKit stack, resetting the active context either to the next item on the stack or to `nil`.

By surrounding Listing 1-7 with these calls, you can embed the Bezier path-based drawing code from Listing 1-7 into the Quartz-based context in Listing 1-3. This approach bridges UIKit drawing into Core Graphics context creation and management.

To summarize, the steps for mixing a Core Graphics context with UIKit drawing are as follows:

1. Create a Core Graphics context.
2. Push the context using `UIGraphicsPushContext()`.
3. Use a combination of UIKit drawing calls that infer the current context and Quartz calls that use the context explicitly.
4. (Optionally retrieve the context contents as an image.)
5. Pop the context with `UIGraphicsPopContext()`.
6. Release the context's memory.

If you try to draw with UIKit with no active context, you'll receive warnings that the context is invalid.

UIKit and Quartz Colors

Many Core Foundation classes have UIKit equivalents and vice versa. Often these are *toll-free bridged*, which means the Core Foundation-style data types can be used interchangeably with UIKit version. You use ARC bridging calls (`__bridge`) to move between the Core Foundation-style version and the UIKit version. Toll-free bridging is notably absent from many Quartz/UIKit relationships, including colors.

Most drawing routines and classes in UIKit represent Objective-C wrappers around Quartz functions and Core Graphics objects. `UIColor` holds a `CGColor` inside it. `UIBezierPath` instances include `CGPath` internals. `UIImage` wraps `CGImage` or `CIImage`. These wrappers are not, however, equivalent objects. Although you can easily access the backing Quartz elements, they can't be bridged.

In UIKit, the `UIColor` class represents color and opacity values. You create these by using a variety of entry points, but the most common approaches involve supplying either RGB values (`colorWithRed:green:blue:alpha:`) or HSV values (`colorWithHue:saturation:brightness:alpha:`) to the class.

When working with Core Graphics, you'll find yourself moving back and forth between UIKit and Quartz. To assist with this, every `UIColor` offers a `CGColor` property. This

property offers a Quartz `CGColorRef` corresponding to the instance's color and alpha values.

You use this property to provide compliant parameters to Quartz. For example, consider the following line of code:

```
CGContextSetFillColorWithColor(context,  
    [UIColor greenColor].CGColor);
```

This function call consists of a Quartz routine that sets a context's fill color. The second parameter starts with the standard green preset from `UIColor`. Its `CGColor` property provides a type that represents a Quartz 2D drawing color, which can safely be used with the Core Graphics function.

Take special care when separating or referencing a Quartz color from its UIKit wrapper. You may need to manually retain that color so its lifetime extends beyond the scope of its parent. Many devs have gotten caught by ARC's memory management, encountering freed memory errors when passing a `CGColorRef` variable that no longer points to a valid instance. Mark Dalrymple of the Big Nerd Ranch discusses this issue in <http://blog.bignerdranch.com/296-arc-gotcha-unexpectedly-short-lifetimes>.

To move back to UIKit from Quartz, use the class constructor method. This builds a UIKit instance from the Core Graphics object reference:

```
UIColor *color = [UIColor colorWithRed:myCGColorRef];
```

Note

Take care when using the `CGColor` property of `UIColor` instances. ARC may assume you won't use the UIKit color any further after you assign `CGColor` to a variable. This can raise an exception. Always retain and release Core Graphics properties you assign to variables, even if you extract them from UIKit objects.

The Painter's Model

iOS uses a *painter's model* to draw in contexts. Unless you specify otherwise, all new drawing is added on top of the existing drawing. This is similar to the way a painter physically applies pigments to a canvas. You modify a context by overlaying new drawing operations.

Listing 1-8 demonstrates this model. Its code builds two circular Bezier paths. It draws the left one in purple, then the right one in green. Figure 1-6 shows the result. The green circle overlaps and obscures part of the original purple drawing.

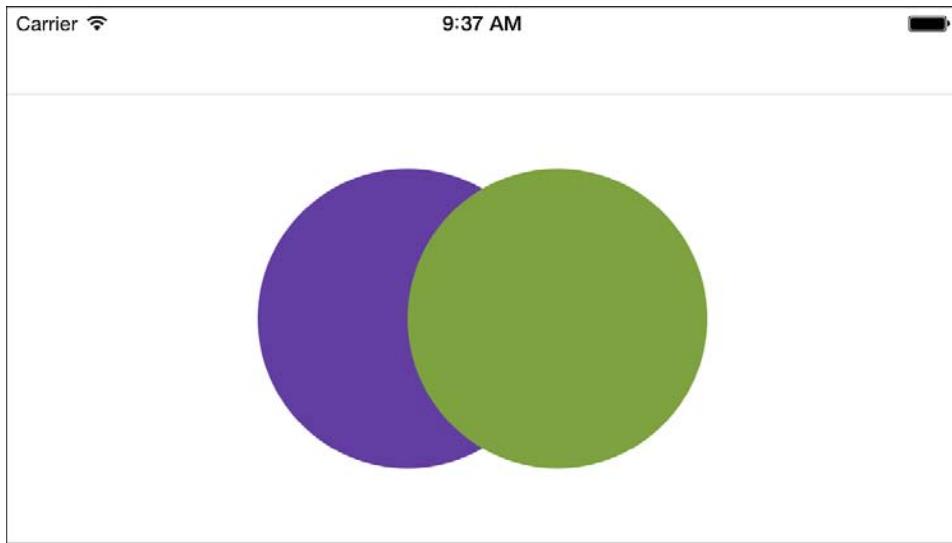


Figure 1-6 The green circle is drawn after the purple one, covering up part of that original drawing.

Note

The sample code in this chapter refers to two UIColor instance variables called `greenColor` and `purpleColor`. These are not the UIKit-supplied standard colors. They derive from the tentative cover art used during the writing of this book. The green color RGB values are (125, 162, 63). The purple color values are (99, 62, 162).

Interested in exploring iOS color further? Visit <http://github.com/erica/uicolor-utilities>.

Listing 1-8 Drawing Overlapping Circles

```
- (UIImage *) buildImage
{
    // Create two circular shapes
    CGRect rect = CGRectMake(0, 0, 200, 200);
    UIBezierPath *shape1 = [UIBezierPath bezierPathWithOvalInRect:rect];
    rect.origin.x += 100;
    UIBezierPath *shape2 = [UIBezierPath bezierPathWithOvalInRect:rect];

    UIGraphicsBeginImageContext(CGSizeMake(300, 200));
```

```
// First draw purple
[purpleColor set];
[shape1 fill];

// Then draw green
[greenColor set];
[shape2 fill];

UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
return image;
}
```

If you reverse this drawing order, drawing `shape2` first and then `shape1`, you get the results shown in Figure 1-7. Although the positions and colors are identical to those in Figure 1-6, the purple shape is drawn second. It obscures part of the green circle. The most recently drawn pigments are added to the canvas on top of the existing content.

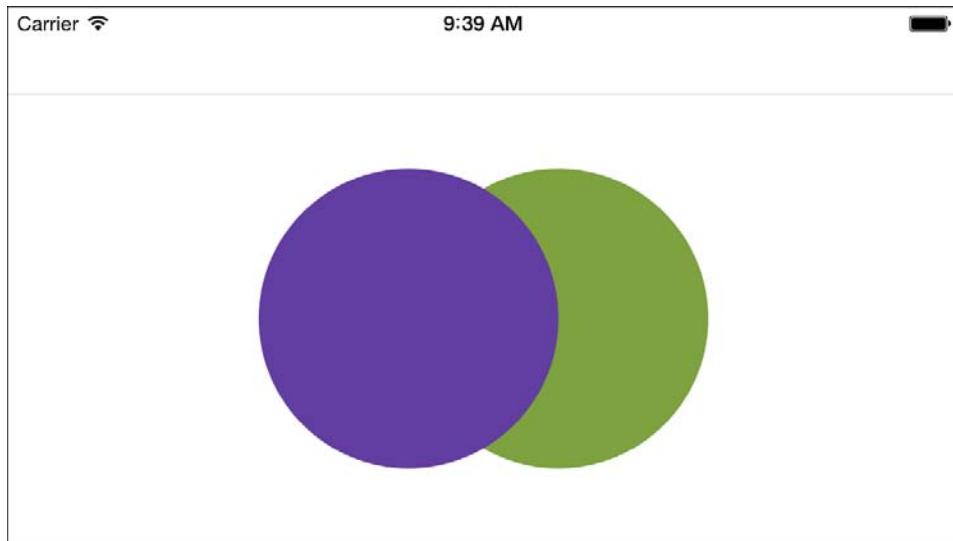


Figure 1-7 Reversing the drawing order causes the purple circle to obscure part of the green circle.

Translucency

When drawing, translucency plays an important role when painting. Adjusting the purple color's alpha component changes the result of the drawing operation. Here, the color's translucency is set to half its normal value:

```
purpleColor = [purpleColor colorWithAlphaComponent:0.5f];
```

Figure 1-8 shows the result of drawing the purple circle with the adjusted alpha level. Although the purple color still covers the green shape, its translucency allows you to see through it to the green color, which was laid down first.

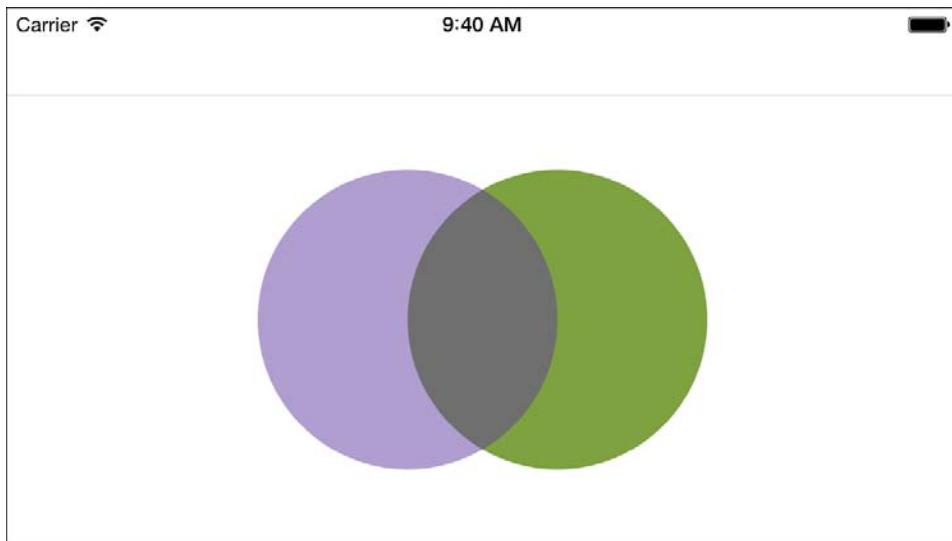


Figure 1-8 Using a partially translucent purple color enables you to see the green shape “behind” the purple drawing.

Although, as you're about to discover, the “rules” of drawing can change programmatically, one thing stays constant: The new source material you draw into a context always affects whatever context is already there, regardless of the mathematics applied. This applies even when using blend modes, such as “destination atop,” where the source material is only drawn into clear areas that are not already filled by context data.

The key lies in understanding that there's a source that you're adding, whether it's a shape, a line, or an image, and a destination you're drawing to, which is represented by the context. The additive nature of drawing routines enables you to lay down one drawing element at a time, to iteratively build up to the goal you're working toward, just as a painter draws on a real-world canvas.

Context State

In Listing 1-8, the `set` method, when called on a color instance, specified the color of subsequent fill and stroke operations within the current context. In that listing, the purple color was set first, and then the green color. After each color was specified, it applied to all subsequent drawing operations.

Two related methods specify whether a color is used for just fill (`setFill`) or just stroke (`setStroke`) operations. A fill is used to completely color the interior of a shape. A stroke is used to outline that shape. In Figure 1-9, the fill color is green, and the stroke color is purple.

All three of these methods (`set`, `setFill`, and `setStroke`) update a current drawing state, specifying the active fill and stroke colors.

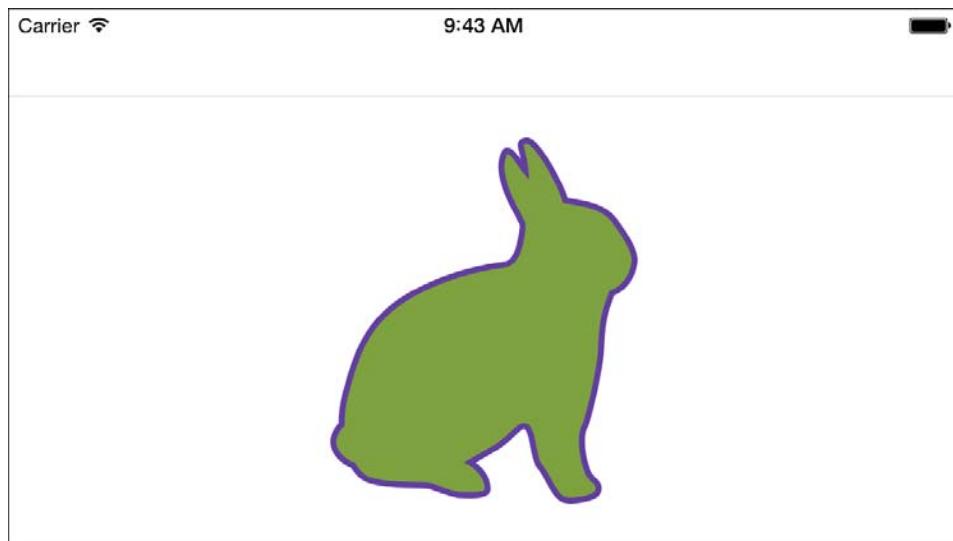


Figure 1-9 The context stroke and fill colors apply to all subsequent drawing operations.

Applying State

Consider the following lines of code, which were used to create the graphic shown in Figure 1-9:

```
[greenColor setFill];
[purpleColor setStroke];
[bunnyPath fill];
[bunnyPath stroke];
```

They set fill and stroke colors and then apply them to a Bezier path. The big question is this: Who is the target of these calls, storing the fill and stroke color states, allowing them to be applied by later operations?

The answer is the current context. The object returned by `UIGraphicsGetCurrentContext()` stores the fill and stroke colors. That context is inferred by each setting and drawing method.

All contexts store graphic state information, which acts as drawing operation parameters. Fill and stroke color are just two types of state saved to a context. As you’re about to discover, contexts can store quite a lot of information. The graphics state affects drawing operations by tweaking the ways each operation is realized.

Pushing and Popping Graphic State

Every context owns a stack of graphic state settings. Each time you create a new context, the stack starts with a fresh state. You can then modify that state and, if needed, push and pop copies of that state onto and off of a graphics state (`GState`) stack.

This stack is different from the context stack maintained by UIKit. That stack stores drawing destinations, letting you move between contexts by pushing and popping the stack. A drawing destination is like a canvas. When you change the context stack, you choose which canvas to draw to. The state stack is specific to each context. It holds sets of drawing preferences that apply to this context alone, changing how drawing operations apply to each “canvas.” Both approaches use stacks, but they affect different parts of the graphics system.

Each graphics state remembers any changes made to it. For example, if you push a new state onto the stack and adjust the default line width to 10, that context state persists until it’s popped off the stack. After that, the default line width returns to whatever value it was before that state was created.

Listing 1-9 demonstrates the process of managing the graphics state stack. It starts by setting the fill and stroke colors to the same green and purple you saw used in Figure 1-9. It draws a bunny and then “saves” the current state by calling `CGContextSaveGState()`. This pushes a copy of the state onto the context’s `GState` stack. Any changes to the context will now apply to that new copy of the graphics state.

If you kept drawing without making any changes to that state, you’d keep creating green bunnies with purple outlines. However, Listing 1-9 updates its colors before drawing. These new colors are orange and blue. And they override any previous color settings for the current state. When the second bunny is drawn, it displays in orange and blue.

Finally, the listing restores the previous graphics state by calling `CGContextRestoreGState()`. This pops the stack, discarding any changes made to the newer state copy. The final bunny therefore falls back to the original color states, namely purple and green. Figure 1-10 shows the result of the drawing operations detailed in Listing 1-9.

Listing 1-9 Managing State

```
UIGraphicsBeginImageContext(size);
CGContextRef context = UIGraphicsGetCurrentContext();

// Set initial stroke/fill colors
[greenColor setFill];
[purpleColor setStroke];

// Draw the bunny
[bunnyPath fill];
[bunnyPath stroke];

// Save the state
CGContextSaveGState(context);

// Change the fill/stroke colors
[[UIColor orangeColor] setFill];
[[UIColor blueColor] setStroke];

// Move then draw again
[bunnyPath applyTransform:
    CGAffineTransformMakeTranslation(50, 0)];
[bunnyPath fill];
[bunnyPath stroke];

// Restore the previous state
CGContextRestoreGState(context);

// Move then draw again
[bunnyPath applyTransform:
    CGAffineTransformMakeTranslation(50, 0)];
[bunnyPath fill];
[bunnyPath stroke];

UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
```

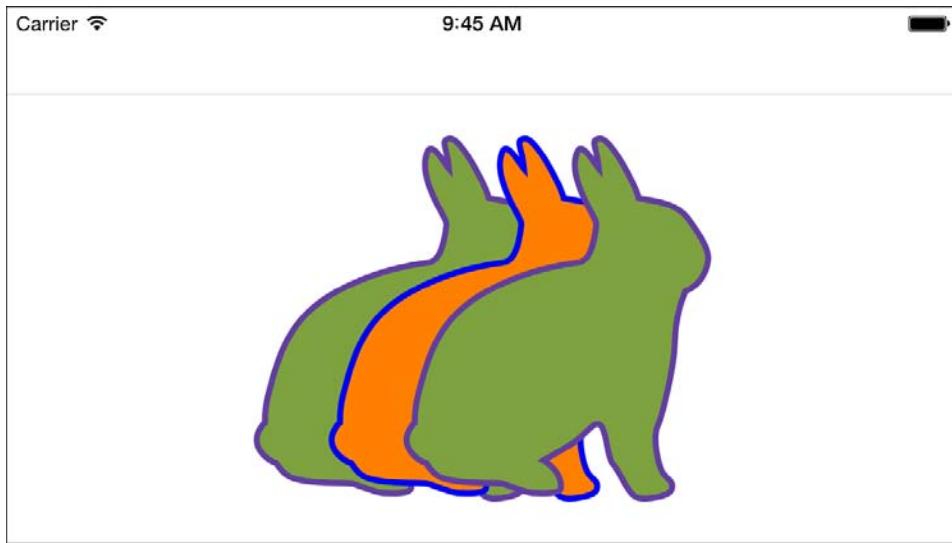
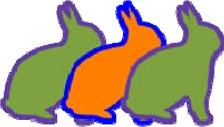


Figure 1-10 The color changes used to draw the second bunny are discarded upon restoring the graphic state to its previous settings.

State Types

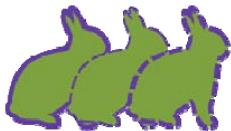
A context saves many kinds of state, not just fill and stroke settings. Each of these states expresses a persistent aspect of the current context. Table 1-1 lists customizable state attributes that you can adjust using Core Graphics context calls and provides visual examples that roughly demonstrate the kinds of changes you can make to these settings.

Table 1-1 Context States

Technology	Explanation
	Color —Color states consist of the fill and stroke settings that specify how items are drawn to the context.
	Transformation matrices —These apply geometric transformations to contexts, allowing you to rotate, scale, and translate the canvas you’re drawing to, in order to create sophisticated geometric results.



Clipping—When you clip a context, you create a shape that automatically excludes content. This enables you to build content limited to circles, rectangles, or any other shape you can imagine.



Line parameters—Line states describe how Quartz draws your lines. These states include width (the thickness of the line), dash patterns (the pattern used to draw the line), miter limits (how pointy angles are), join styles (how corners are expressed; styles include miter, round, or bevel), and caps (the ends of lines, drawn as butt, round, or square).



Flatness—This is a factor that determines how accurate each curved path segment can be, specifying the maximum permissible distance between a point on the mathematical curve and the rendered point. The default is 0.6. Larger values produce more jagged curves but they are rendered faster as they require fewer computations.



Antialiasing—This determines whether Quartz mathematically smoothes jagged lines on curves and diagonals by averaging values between pixels. Antialiasing renders more slowly than normal drawing, but its results are visually superior. Quartz defaults to using antialiasing.



Alpha levels—These control the transparency of the material drawn to the context. As the alpha level decreases from 1 (fully opaque) to 0 (fully invisible), drawn material becomes more and more transparent.

bunny
bunny



Text traits—Text states include font, font size, character spacing, and text drawing modes. Modes specify how the text is drawn (by stroking, filling, etc.). Other details control font smoothing and subpixel positioning.

Blend modes—Blend modes use color and alpha levels to determine how to blend each new layer of color into the material already in the destination. Quartz supplies numerous blend modes. Appendix A explores these modes in exhaustive detail.

Context Coordinate System

When you work primarily in UIKit, the coordinate system starts at the top-left corner of the screen (0, 0) and extends to the right and down. In Quartz, the coordinate system starts at the bottom left of the screen.

Look at Figure 1-11. It depicts a square {20, 20, 40, 40} drawn in UIKit (left) and Quartz (right). In each case, the object begins 20 points off the origin. However, as you can see, that origin differs in each circumstance.

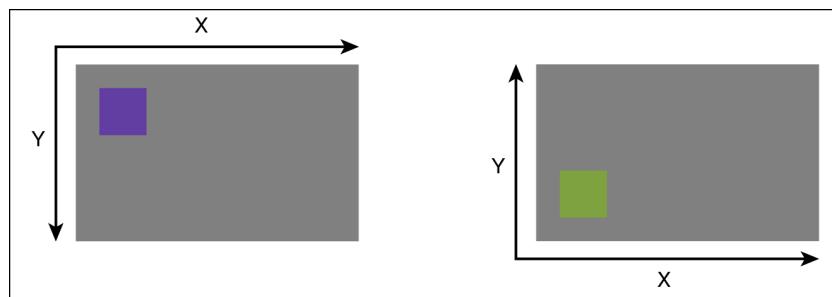


Figure 1-11 The UIKit origin (top left) is different from the Quartz origin (bottom left).

Which origin holds sway depends on how you created the context. If you built it using any of the UIKit functions, the origin is at the top left. If you used `CGBitmapContextCreate()`, the origin is at the bottom left.

Flipping Contexts

You can adjust Core Graphics contexts to use the UIKit origin. Listing 1-10 shows the code, which implements these steps:

1. Push the `CGContextRef` onto the UIKit graphics stack.
2. Flip the context vertically. You do this by building a transform that scales and translates the original context.
3. Concatenate that transform to the context's current transformation matrix (CTM). This adjusts the context's coordinate system to mimic UIKit, letting you draw starting from the top-left corner.
4. Perform any drawing operations using the new coordinate system.
5. Pop the graphics stack.

You can work directly in Quartz without applying this coordinate workaround. Most UIKit developers, however, appreciate the ability to use a single system that matches drawing to familiar view placement tasks. Some developers create dual macros, defined to the same flipping function. This enables them to visually match a `QUARTZ_ON` request to a `QUARTZ_OFF` one. The routine doesn't change, but a developer gains a sense of the current state for code inspection.

Listing 1-10 contains a secondary flipping routine, one that does not require you to supply a context size in points. Quite honestly, it's a bit of a hack, but it does work because the image it retrieves will use the same size and scale as the context.

You can also retrieve a context's size by calling `CGBitmapContextGetHeight()` and `CGBitmapContextGetWidth()` and dividing the number of pixels these functions return by the screen scale. This assumes that you're working with a bitmap context of some kind (like the one created by `UIGraphicsBeginImageContextWithOptions()`) and that you're matching the screen's scale in that context.

Listing 1-10 Adjusting Coordinate Origins

```
// Flip context by supplying the size
void FlipContextVertically(CGSize size)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to flip");
        return;
    }
    CGAffineTransform transform = CGAffineTransformIdentity;
    transform = CGAffineTransformScale(transform, 1.0f, -1.0f);
    transform = CGAffineTransformTranslate(transform, 0.0f, -size.height);
    CGContextConcatCTM(context, transform);
}

// Flip context by retrieving image
void FlipImageContextVertically()
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to flip");
        return;
    }
    UIImage *image =
        UIGraphicsGetImageFromCurrentImageContext();
    FlipContextVertically(image.size);
}
```

```

// Query context for size and use screen scale
// to map from Quartz pixels to UIKit points
CGSize GetUIKitContextSize()
{
    CGContextRef context =
        UIGraphicsGetCurrentContext();
    if (context == NULL) return CGSizeZero;
    CGSize size = CGSizeMake(
        CGBitmapContextGetWidth(context),
        CGBitmapContextGetHeight(context));
    CGFloat scale = [UIScreen mainScreen].scale;
    return CGSizeMake(
        size.width / scale, size.height / scale);
}

// Initialize the UIKit context stack
UIGraphicsPushContext(context);

// Flip the context vertically
FlipContextVertically(size);

// Draw the test rectangle. It will now use the UIKit origin
// instead of the Quartz origin.
CGRect testRect = CGRectMake(20, 20, 40, 40);
UIBezierPath *path = [UIBezierPath bezierPathWithRect:testRect];
[greenColor set];
[path fill];

// Pop the context stack
UIGraphicsPopContext();

```

Clipping

Clipping enables you to exclude any painting operations that fall outside a path within your context. To clip, add a path to your context and then call the `CGContextClip()` clipping function. Figure 1-12 shows an example, where colored circles drawn to a context are clipped within the letters of the word *Hello*.



Figure 1-12 Randomly colored circles are clipped to the boundaries of the word *Hello*.

Listing 1-11 demonstrates the steps involved in clipping:

1. Save the graphic state. This enables you to restore the state to a preclipped version at a later time. If you won't need to return to an unclipped state, skip this step.
2. Add a path to the context and clip to it using `CGContextClip()`. Adding a path temporarily stores it in the graphics context. When stored, you create a mask by clipping. This blocks out the parts of the context you don't want to paint to. This example uses a `UIBezierPath` instance, which is not compatible with the `CGContextClip()` function. Retrieve the `CGPathRef` from the Bezier path's `CGPath` property and pass that instead.
3. When clipped, perform any standard drawing operations. Material drawn outside the bounds of the clipping path is automatically discarded on your behalf.
4. To finish, restore the graphic state. This allows you to resume normal drawing without any further clipping, returning the context to the state before you began clipped operations.

This process amounts to saving, clipping, drawing, and restoring. (Say it out loud.) Chapter 6 introduces ways to use Objective-C blocks to clip and draw within saved-and-restored graphics states rather than managing those states through explicit calls.

Listing 1-11 Drawing Using Clipping

```
// Save the state
CGContextSaveGState(context);

// Add the path and clip
CGContextAddPath(context, path.CGPath);
CGContextClip(context);

// Perform clipped drawing here
```

```
// Restore the state  
CGContextRestoreGState(context);  
  
// Drawing done here is not clipped
```

Transforms

The sequence of letters in Figure 1-13 is built by drawing each character at points around a circle. This figure takes advantage of one of UIKit's terrific built-in features: `NSString` instances know how to draw themselves into contexts. You just tell a string to draw at a point or into a rectangle, as in this example:

```
[@"Hello" drawAtPoint:CGPointMake(100, 50)  
    withAttributes:@{NSFontAttributeName:font}]
```

The syntax for this call changed in iOS 7, deprecating earlier APIs. Prior to iOS 7, you'd say:

```
[@"Hello" drawAtPoint:CGPointMake(100, 50) withFont:font]
```

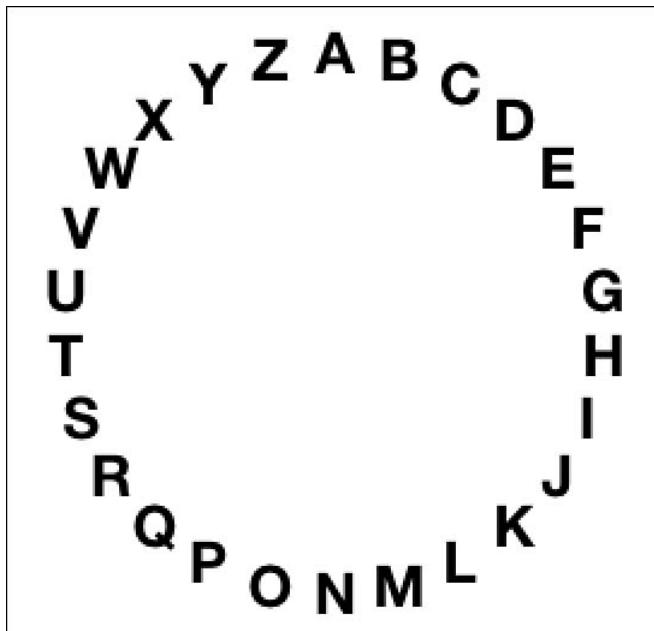


Figure 1-13 Drawing letters in a circle.

This circle progresses clockwise, with a letter deposited every $(2 \times \pi / 26)$ radians. Each x and y position was calculated as an offset from a common center. The following code iteratively calculates points around the circle using $r \times \sin(\theta)$ and $r \times \cos(\theta)$, using those points to place each letter:

```
NSString *alphabet = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
for (int i = 0; i < 26; i++)
{
    NSString *letter =
        [alphabet substringWithRange:NSMakeRange(i, 1)];
    CGSize letterSize =
        [letter sizeWithAttributes:@{NSFontAttributeName:font}];

    CGFloat theta = M_PI - i * (2 * M_PI / 26.0);
    CGFloat x = center.x + r * sin(theta) - letterSize.width / 2.0;
    CGFloat y = center.y + r * cos(theta) - letterSize.height / 2.0;

    [letter drawAtPoint:CGPointMake(x, y)
        withAttributes:@{NSFontAttributeName:font}];
}
```

This is an acceptable way to approach this challenge, but you could greatly improve it by leveraging context transforms.

Note

High school algebra can be essential for developers working with Core Graphics. Revisiting concepts like sine, cosine, tangents, and matrices can offer you exciting tools to work with. Rusty? Khan Academy (www.khanacademy.org) offers an indispensable resource for freshening your skills.

Transform State

Every context stores a 2D affine transform as part of its state. This transform is called the current transform matrix. It specifies how to rotate, translate, and scale the context while drawing. It provides a powerful and flexible way to create advanced drawing operations. Contrast the layout shown in Figure 1-14 with the one in Figure 1-13. This improvement is achieved through the magic of context transforms.



Figure 1-14 Drawing letters in a circle using transforms.

Listing 1-12 shows the steps that went into creating this. It consists of a series of transform operations that rotate the canvas and draw each letter. Context save and restore operations ensure that the only transform that persists from one drawing operation to the next is the one that appears in boldface in the listing. This translation sets the context's origin to its center point.

This enables the context to freely rotate around that point, so each letter can be drawn at an exact radius. Moving to the left by half of each letter width ensures that every letter is drawn centered around the point at the end of that radius.

Listing 1-12 Transforming Contexts During Drawing

```
NSString *alphabet = @"ABCDEFGHIJKLMNPQRSTUVWXYZ";  
  
// Start drawing  
UIGraphicsBeginImageContext(bounds.size);  
CGContextRef context = UIGraphicsGetCurrentContext();  
  
// Retrieve the center and set a radius  
CGPoint center = RectGetCenter(bounds);  
CGFloat r = center.x * 0.75f;
```

```

// Start by adjusting the context origin
// This affects all subsequent operations
CGContextTranslateCTM(context, center.x, center.y);

// Iterate through the alphabet
for (int i = 0; i < 26; i++)
{
    // Retrieve the letter and measure its display size
    NSString *letter =
        [alphabet substringWithRange:NSMakeRange(i, 1)];
    CGSize letterSize =
        [letter sizeWithAttributes:@{NSFontAttributeName:font}];

    // Calculate the current angular offset
    CGFloat theta = i * (2 * M_PI / (float) 26);

    // Encapsulate each stage of the drawing
    CGContextSaveGState(context);

    // Rotate the context
    CGContextRotateCTM(context, theta);

    // Translate up to the edge of the radius and move left by
    // half the letter width. The height translation is negative
    // as this drawing sequence uses the UIKit coordinate system.
    // Transformations that move up go to lower y values.
    CGContextTranslateCTM(context, -letterSize.width / 2, -r);

    // Draw the letter and pop the transform state
    [letter drawAtPoint:CGPointMake(0, 0)
        withAttributes:@{NSFontAttributeName:font}];
    CGContextRestoreGState(context);
}

// Retrieve and return the image
UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
return image;

```

Building a More Exacting Layout

A more pedantic solution to drawing the letters in a circle avoids the extra spacing around the *I* and the squeezing around the *W*. Listing 1-13 details the steps needed to create the more precise layout shown in Figure 1-15. This example demonstrates finer layout placement.

Start by calculating the total width of the layout. Sum the width of each individual letter (as done here) or just measure the string as a whole. This enables you to mark your progress along the layout, producing a percentage of travel from beginning to end.

Next, adjust the placement for each letter based on the percentage consumed by each iteration. Use this percentage to calculate a rotation angle for laying down the letter.

Finish by drawing the letter just as you did in Listing 1-12. What you end up with is a layout that respects the width of each letter, moving forward proportionately according to the letter's natural size.

Listing 1-13 Precise Text Placement Around a Circle

```
// Calculate the full extent
CGFloat fullSize = 0;
for (int i = 0; i < 26; i++)
{
    NSString *letter =
        [alphabet substringWithRange:NSMakeRange(i, 1)];
    CGSize letterSize =
        [letter sizeWithAttributes:@{{NSFontAttributeName:font}}];
    fullSize += letterSize.width;
}

// Initialize the consumed space
CGFloat consumedSize = 0.0f;

// Iterate through each letter, consuming that width
for (int i = 0; i < 26; i++)
{
    // Measure each letter
    NSString *letter =
        [alphabet substringWithRange:NSMakeRange(i, 1)];
    CGSize letterSize =
        [letter sizeWithAttributes:@{{NSFontAttributeName:font}}];

    // Move the pointer forward, calculating the
    // new percentage of travel along the path
    consumedSize += letterSize.width / 2.0f;
    CGFloat percent = consumedSize / fullSize;
    CGFloat theta = percent * 2 * M_PI;
    consumedSize += letterSize.width / 2.0f;

    // Prepare to draw the letter by saving the state
    CGContextSaveGState(context);

    // Rotate the context by the calculated angle
    CGContextRotateCTM(context, theta);
}
```

```
CGContextRotateCTM(context, theta);

// Move to the letter position
CGContextTranslateCTM(context, -letterSize.width / 2, -r);

// Draw the letter
[letter drawAtPoint:CGPointMake(0, 0) withFont:font];

// Reset the context back to the way it was
CGContextRestoreGState(context);
}
```

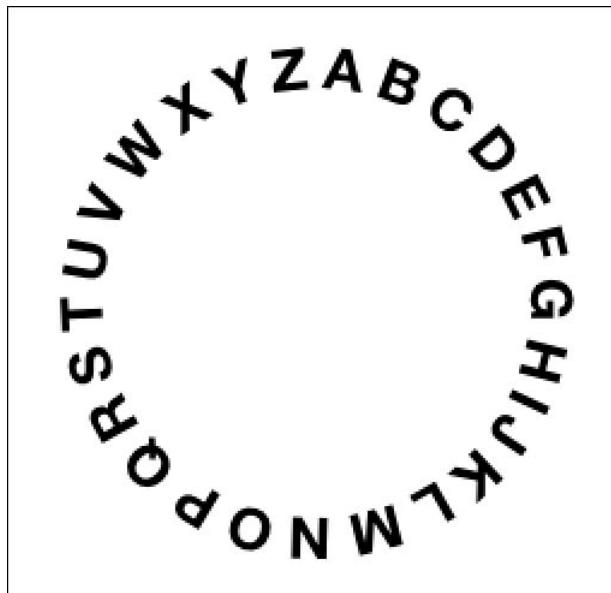


Figure 1-15 This version uses better spacing, based on the width of each letter.

Setting Line Parameters

Each context stores line width as part of its graphics state. You can adjust that line width by calling `CGContextSetLineWidth()` and passing the point size as its parameter. Subsequent drawing operations inherit the change in that state. That width does not, however, apply to `UIBezierPath` instances, the primary drawing tool for UIKit. Every UIKit path permits you

to adjust its stroke width by setting its `lineWidth` parameter. Whatever value this is set to wins out over any context settings.

Take a look at Listing 1-14. It creates a path and sets its line width to 4. Then it instructs the context to use a 20-point-wide line. Figure 1-16 shows the result of these drawing requests. As the image demonstrates, drawing the path through Quartz honors that 20-point width. Drawing through UIKit overrides the context state with the path parameter.

Listing 1-14 Conflicting Line Widths

```
UIGraphicsBeginImageContext(bounds.size);
CGContextRef context = UIGraphicsGetCurrentContext();

// Build a Bezier path and set its width
UIBezierPath *path =
    [UIBezierPath bezierPathWithRoundedRect:rect cornerRadius:32];
path.lineWidth = 4;

// Update the context state to use 20-point wide lines
CGContextSetLineWidth(context, 20);

// Draw this path using the context state
[purpleColor set];
CGContextAddPath(context, path.CGPath);
CGContextStrokePath(context);

// Draw the path directly through UIKit
[greenColor set];
[path stroke];
```

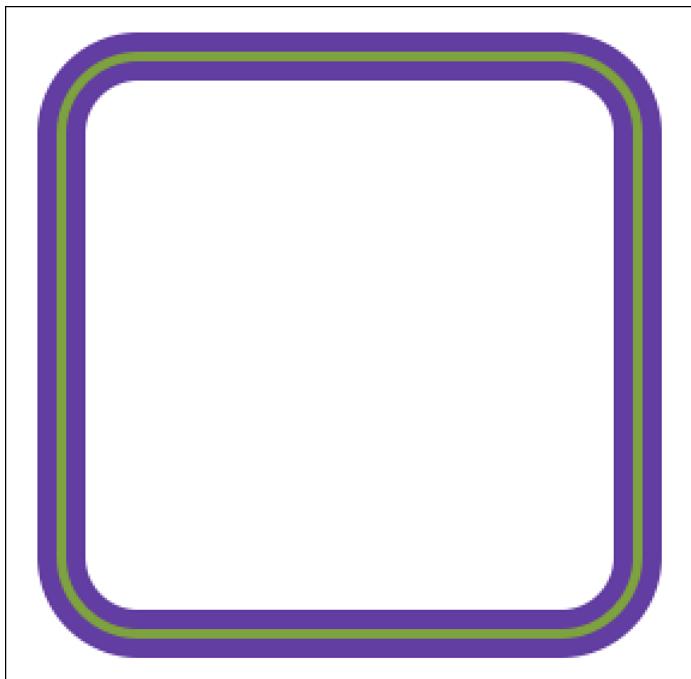


Figure 1-16 Some Quartz context semantics are implemented orthogonally to UIKit. The smaller interior stroke does not inherit the context's default line width.

UIKit's approach binds line width semantics to the `UIBezierPath` object and not to the context as a whole. That's not a bad thing. Pushing a line width state in order to draw a shape makes less sense than associating that width with a particular object. It is not, however, a consistent thing. This approach is not universal—at least not yet. You can see this by trying out dashes.

Dashes

Dash patterns do not demonstrate the UIKit/Quartz dichotomy seen with line width. Changes made to the context state *will* affect UIKit paths, whether you use the UIKit or Quartz entry points. To set a path's dash pattern in UIKit, use the `setLineDash:dashes count:phase:` method.

This code snippet creates the output shown in Figure 1-17:

```
[greenColor set];
CGFloat dashes[] = {6, 2};
[path setLineDash:dashes count:2 phase:0];
[path stroke];
```

This snippet uses three arguments. The first sets the dash pattern (6 points on, 2 points off), the second counts the first argument (2 items), and the third specifies a phase offset—that is, how far into the pattern to start. 0 means default to the normal layout pattern. You'll read more about dashes in Chapter 4.

The same effect can be applied to the context as a whole by calling the equivalent Quartz function, `CGContextSetLineDash()`. This next code snippet also produces the output shown in Figure 1-17. It just gets there in a different way:

```
[greenColor set];
CGFloat dashes[] = {6, 2};
CGContextSetLineDash(context, 0, dashes, 2);
[path stroke];
```

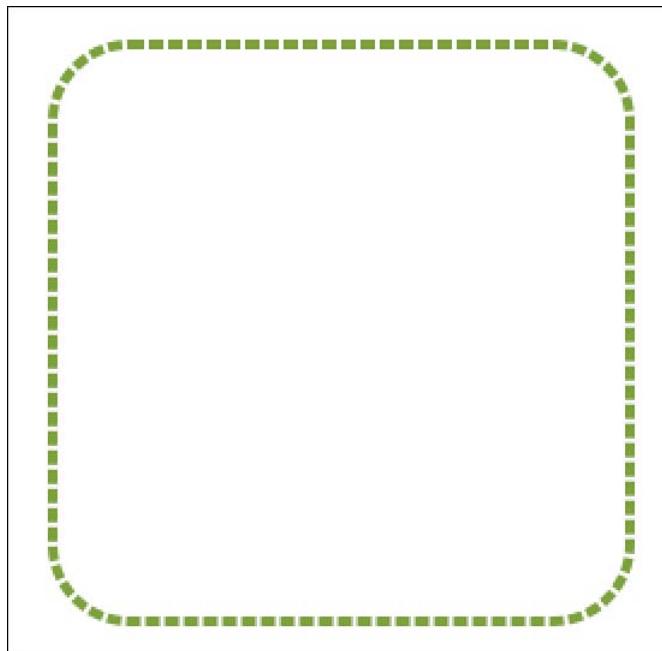


Figure 1-17 Dashes can be set in the context state or applied to a path instance.

Summary

This chapter introduces the basis of iOS drawing: the context. You saw how to create contexts, read about the different kinds of contexts available, and discovered how to

retrieve content for use in images and PDF files. In iOS, contexts offer a wide array of nuanced state control. Before you leave this chapter, here are a few points to think about:

- For any drawing requirement, there are usually many solutions that get you to the same place. Your comfort in using Core Graphics functions versus UIKit classes may influence your choice of APIs, but there's often no *right* answer in how you must or should draw. Do whatever works best for your development needs.
- UIKit classes continue evolving. With each generation of iOS, they provide more and more resources that enable you to skip direct Quartz. UIKit APIs offer the advantages of simplicity, parsimony, and ease of inspection. Most iOS developers will find that these advantages create a powerful case for primarily using UIKit.

Apple generally advises developers to work with the highest-level API possible. Use lower-level APIs when you need functionality that higher-level versions do not provide. There are numerous things you can do in the older-style APIs that you simply cannot do in UIKit, such as painting with calculated shadows or working with output color spaces.

- Being comfortable using Core Foundation-style C with manual memory management continues to be a rewarding skill. Knowing how the `Ref` style objects work and when and how to release remains relevant for the foreseeable future. Learn to use ARC-style bridging to transfer memory management to and from the Core Foundation system. You'll be glad you invested the time because of the greater flexibility you gain.

Remember this rule of thumb: If a function's name says `Copy` or `Create`, you have to release items. If it says `Get`, you don't. And, wherever possible, always look for a release function with the same prefix as the `Create` function. For example, if you create a color space, use `CGColorSpaceRelease()` to balance that.

- Both Quartz and UIKit drawing *are* thread-safe. Apple Technical Q&A 1637 notes that beginning with iOS 4.0, “drawing to a graphics context in UIKit is thread-safe. This includes accessing and manipulating the current graphics stack, drawing images and strings, and usage of color and font objects from secondary threads.”

2

The Language of Geometry

Drawing and geometry are inextricably linked. In order to express a drawing operation to the compiler, you must describe it with geometric descriptions that iOS can interpret on your behalf. This chapter reviews basics you'll need to get started. It begins with the point-pixel dichotomy, continues by diving into core structures, and then moves to UIKit objects. You'll learn what these items are and the roles they play in drawing.

Points Versus Pixels

In iOS, *points* specify locations on the screen and in drawings. They provide a unit of measure that describes positions and extents for drawing operations. Points have no fixed relationship to physical-world measurements or specific screen hardware. They enable you to refer to positions independently of the device being used.

Points are not pixels. *Pixels* are addressable screen components and are tied directly to specific device hardware. Each pixel can individually be set to some brightness and color value. A point, in contrast, refers to a logical coordinate space.

For example, all members of the iPhone family present a display that is 320 points wide in portrait orientation. Those points correspond to 320 pixels wide on older units and 640 pixels wide on newer Retina-based models. A unified coordinate system applies across all iPhone devices, regardless of whether you're using newer Retina models or older, lower pixel-density units.

As Figure 2-1 shows, the position (160.0, 240.0) sits in the center of the 3.5-inch iPhone or iPod touch screens in portrait orientation, regardless of pixel density. That same point rests a bit above center on 4-inch Retina-based iPhones and iPod touches. The natural center for these newer devices is (160.0, 284.0).

In landscape orientation, the same point lies toward the bottom-left corner of iPhone screens. On the larger iPad screen, this appears towards the top-left corner of the screen.

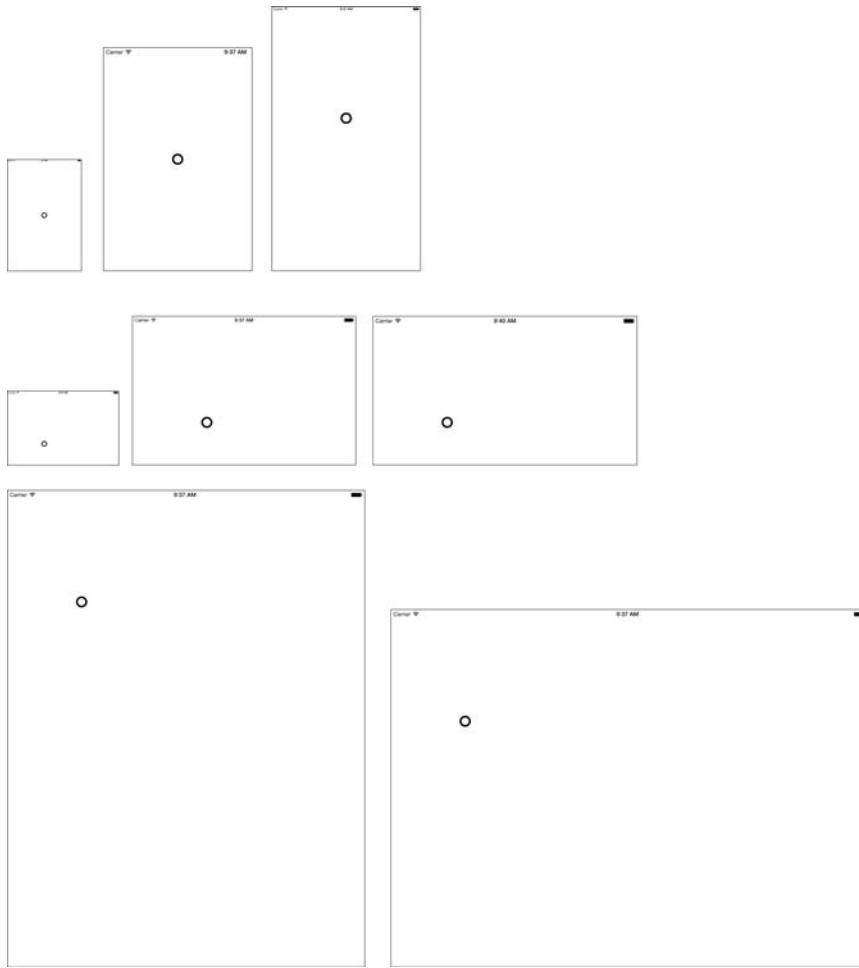


Figure 2-1 The logical point (160.0, 240.0) displayed on a variety of target devices. Top row: Original iPhone, 3.5-inch Retina iPhone, 4-inch Retina iPhone. Middle row: Original iPhone, 3.5-inch Retina iPhone, 4-inch Retina iPhone. Bottom row: Portrait (Retina) iPad, landscape (Retina) iPad.

Scale

The `UIScreen` class provides a property called `scale`. This property expresses the relationship between a display's pixel density and its point system. A screen's scale is used to convert from the logical coordinate space of the view system measured in points to the physical pixel coordinates. Retina displays use a scale of 2.0, and non-Retina displays use a scale of 1.0. You can test for a Retina device by checking the main screen's scale:

```

- (BOOL) hasRetinaDisplay
{
    return ([UIScreen mainScreen].scale == 2.0f);
}

```

The main screen always refers to the device's onboard display. Other screens may be attached over AirPlay and via Apple's connector cables. Each screen provides an `availableModes` property. This supplies an array of resolution objects, ordered from lowest to highest resolution.

Many screens support multiple modes. For example, a VGA display might offer as many as a half dozen or more different resolutions. The number of supported resolutions varies by hardware. There will always be at least one resolution available, but you should be able to offer choices to users when there are more.

The `UIScreen` class also offers two useful display-size properties. The `bounds` property returns the screen's bounding rectangle, measured in points. This gives you the full size of the screen, regardless of any onscreen elements like status bars, navigation bars, or tab bars. The `applicationFrame` property is also measured in points. It excludes the status bar if it is visible, providing the frame for your application's initial window size.

iOS Devices

Table 2-1 summarizes the iOS families of devices, listing the addressable coordinate space for each family. While there are five distinct display families to work with, you encounter only three logical spaces at this time. (Apple may introduce new geometries in the future.) This breaks down design to three groups:

- **3.5-inch iPhone-like devices** (320.0 by 480.0 points)
- **4-inch iPhone-like devices** (320.0 by 568.0 points)
- **iPads** (768.0 by 1024.0 points)

Table 2-1 iOS Devices and Their Logical Coordinate Spaces

Family	Devices	Logical Space	Scale	Physical
3.5-inch iPhone family	3GS and earlier; iPod touch third generation and earlier	320.0 by 480.0 points (portrait)	1.0	320 by 480 pixels (portrait)
3.5-inch iPhone with Retina display family	4 and 4S; iPod touch fourth generation	320.0 by 480.0 points (portrait)	2.0	640 by 960 pixels (portrait)

4-inch iPhone family with Retina display	iPhone 5 and later; iPod touch fifth generation and later	320.0 by 568.0 points (portrait)	2.0	640 by 1136 pixels (portrait)
iPad family	iPad 2 and earlier; iPad mini first generation	768.0 by 1024.0 points (portrait)	1.0	768 by 1024 pixels (portrait)
iPad with Retina display family	Third-generation iPad and later; the rumored but yet-unannounced iPad mini Retina	768.0 by 1024.0 points (portrait)	2.0	1536 by 2048 pixels (portrait)

View Coordinates

The numbers you supply to drawing routines are often tightly tied to a view you’re drawing to, especially when working within `drawRect:` methods. Every view’s native coordinate system starts at its top-left corner.

In iOS 7 and later, a view controller’s origin may or may not start below a navigation bar depending on how you’ve set the controller’s `edgesForExtendedLayout` property. By default, views now stretch under bars, offering an edge-to-edge UI experience.

Frames and Bounds

Views live in (at least) two worlds. Each view’s `frame` is defined in the coordinate system of its parent. It specifies a rectangle marking the view’s location and size within that parent. A view’s `bounds` is defined in its own coordinate system, so its origin defaults to `(0, 0)`. (This can change when you display just a portion of the view, such as when working with scroll views.) A view’s `bounds` and `frame` are tightly coupled. Changing a view’s `frame` size affects its `bounds` and vice versa.

Consider Figure 2-2. The gray view’s origin starts at position `(80, 100)`. It extends 200 points horizontally and 150 points vertically. Its frame in its parent’s coordinate system is `{80, 100, 200, 150}`. In its own coordinate system, its bounds are `{0, 0, 200, 150}`. The extent stays the same; the relative origin changes.

	Gray View	Parent
Origin	<code>{0, 0}</code>	<code>{80, 100}</code>
Circle	<code>{66, 86}</code>	<code>{146, 186}</code>

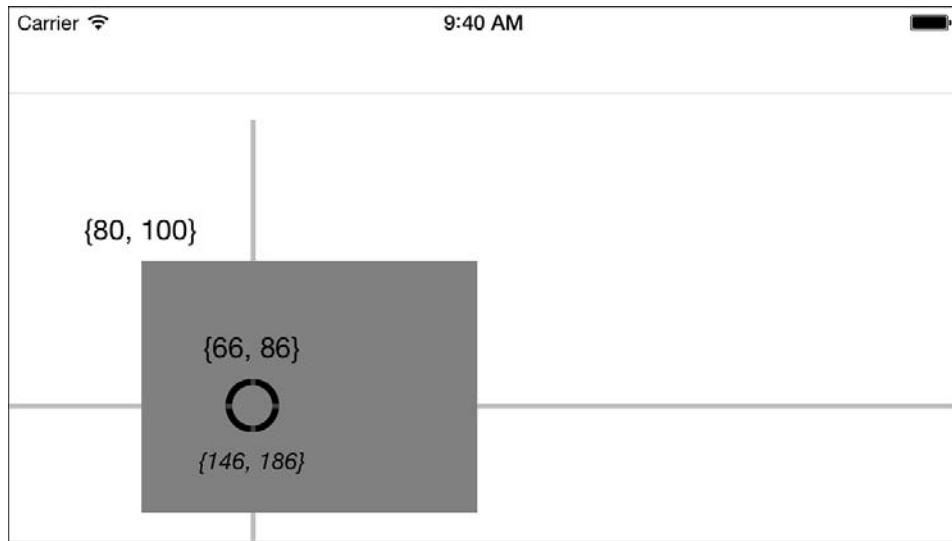


Figure 2-2 The position of any point depends on the coordinate system it's participating in.

A point's location is contingent on the coordinate system it's being defined for. In Figure 2-2, a circle surrounds a point in the gray view. In the gray view's coordinate system, this point appears at position (66, 86)—that is, 66 points down and 86 points to the right of the view's origin.

In the parent's coordinate system, this point corresponds to position (146, 186). That is 146 points down and 186 points to the right of the parent's origin, which is the top left of the white backsplash. To convert from the view's coordinate system to the parent's coordinate system, you add the view's origin (80, 100) to the point. The result is (66 + 80, 100 + 86), or (146, 186).

Note

A view frame is calculated from its bounds, its center, and any transform applied to the view. It expresses the smallest rectangle that contains the view within the parent.

Converting Between Coordinate Systems

The iOS SDK offers several methods to move between coordinate systems. For example, you might want to convert the position of a point from a view's coordinate system to its parent's coordinate system to determine where a drawn point falls within the parent view. Here's how you do that:

```
CGPoint convertedPoint =  
[outerView convertPoint:samplePoint fromView:grayView];
```

You call the conversion method on any view instance. Specify whether you want to convert a point into another coordinate system (`toView:`) or from another coordinate system (`fromView:`), as was done in this example.

The views involved must live within the same `UIWindow`, or the math will not make sense. The views do not have to have any particular relationship, however. They can be siblings, parent/child, ancestor/child, or whatever. The methods return a point with respect to the origin you specify.

Conversion math applies to rectangles as well as points. To convert a rectangle from a view into another coordinate system, use `convertRect:fromView:`. To convert back, use `convertRect:toView:`.

Key Structures

iOS drawing uses four key structures to define geometric primitives: points, sizes, rectangles, and transforms. These structures all use a common unit, the logical point. Points are defined using `CGFloat` values. These are type defined as float on iOS and double on OS X.

Unlike pixels, which are inherently integers, points are not tied to device hardware. Their values refer to mathematical coordinates, offering subpixel accuracy. The iOS drawing system handles the math on your behalf.

The four primitives you work with are as follows:

- **`CGPoint`**—Points structures are made up of an `x` and a `y` coordinate. They define logical positions.
- **`CGSize`**—Size structures have a `width` and a `height`. They establish extent.
- **`CGRect`**—Rectangles include both an `origin`, defined by a point, and a `size`.
- **`CGAffineTransform`**—Affine transform structures describe changes applied to a geometric item—specifically, how an item is placed, scaled, and rotated. They store the `a`, `b`, `c`, `d`, `tx`, and `ty` values in a matrix that defines a particular transform.

The next sections introduce these items in more depth. You need to have a working knowledge of these geometric basics before you dive into the specifics of drawing.

Points

The `CGPoint` structure stores a logical position, which you define by assigning values to the `x` and `y` fields. A convenience function `CGPointMake()` builds a point structure from two parameters that you pass:

```
struct CGPoint {  
    CGFloat x;  
    CGFloat y;  
};
```

These values may store any floating-point number. Negative values are just as valid as positive ones.

Sizes

The `CGSize` structure stores two extents, `width` and `height`. Use `CGSizeMake()` to build sizes. Heights and widths may be negative as well as positive, although that's not usual or common in day-to-day practice:

```
struct CGSize {  
    CGFloat width;  
    CGFloat height;  
};
```

Rectangles

The `CGRect` structure is made up of two substructures: `CGPoint`, which defines the rectangle's origin, and `CGSize`, which defines its extent. The `CGRectMake()` function takes four arguments and returns a populated `rect` structure. In order, the arguments are `x-origin`, `y-origin`, `width`, and `height`. For example, `CGRectMake(0, 0, 50, 100)`:

```
struct CGRect {  
    CGPoint origin;  
    CGSize size;  
};
```

Call `CGRectStandardize()` to convert a rectangle with negative extents to an equivalent version with a positive width and height.

Transforms

Transforms represent one of the most powerful aspects of iOS geometry. They allow points in one coordinate system to transform into another coordinate system. They allow you to

scale, rotate, mirror, and translate elements of your drawings, while preserving linearity and relative proportions. You encounter drawing transforms primarily when you adjust a drawing context, as you saw in Chapter 1, or when you manipulate paths (shapes), as you read about in Chapter 5.

Widely used in both 2D and 3D graphics, transforms provide a complex mechanism for many geometry solutions. The Core Graphics version (`CGAffineTransform`) uses a 3-by-3 matrix, making it a 2D-only solution. 3D transforms, which use a 4-by-4 matrix, are the default for Core Animation layers. Quartz transforms enable you to scale, translate, and rotate geometry.

Every transform is represented by an underlying transformation matrix, which is set up as follows:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

This matrix corresponds to a simple C structure:

```
struct CGAffineTransform {  
    CGFloat a;  
    CGFloat b;  
    CGFloat c;  
    CGFloat d;  
    CGFloat tx;  
    CGFloat ty;  
};
```

Creating Transforms

Unlike other Core Graphics structures, you rarely access an affine transform's fields directly. Most people won't ever use or need the `CGAffineTransformMake()` constructor function, which takes each of the six components as parameters.

Instead, the typical entry point for creating transforms is either `CGAffineTransformMakeScale()`, `CGAffineTransformMakeRotation()`, or `CGAffineTransformMakeTranslation()`. These functions build scaling, rotation, and translation (position offset) matrices from the parameters you supply. Using these enables you to discuss the operations you want to apply in the terms you want to apply them. Need to rotate an object? Specify the number of degrees. Want to move an object? State the offset in points. Each function creates a transform that, when applied, produces the operation you specify.

Layering Transforms

Related functions enable you to layer one transform onto another, building complexity. Unlike the first set of functions, these take a transform as a parameter. They modify that transform to layer another operation on top. Unlike the “make” functions, these are always applied in sequence, and the result of each function is passed to the next. For example, you might create a transform that rotates and scales an object around its center. The following snippet demonstrates how you might do that:

```
CGAffineTransform t = CGAffineTransformIdentity;
t = CGAffineTransformTranslate(t, -center.x, -center.y);
t = CGAffineTransformRotate(t, M_PI_4);
t = CGAffineTransformScale(t, 1.5, 1.5);
t = CGAffineTransformTranslate(t, center.x, center.y);
```

This begins with the identity transform. The identity transform is the affine equivalent of the number 0 in addition or the number 1 in multiplication. When applied to any geometric object, it returns the identical presentation you started with. Using it here ensures a consistent starting point for later operations.

Because transforms are applied from their origin, and not their center, you then translate the center to the origin. Scale and rotation transforms always relate to (0, 0). If you want to relate them to another point, such as the center of a view or a path, you should always move that point to (0, 0). There, you perform the rotation and scaling before resetting the origin back to its initial position. This entire set of operations is stored in the single, final transform, `t`.

Exposing Transforms

The UIKit framework defines a variety of helper functions specific to graphics and drawing operations. These include several affine-specific functions. You can print out a view’s transform via UIKit’s `NSStringFromCGAffineTransform()` function. Its inverse is `CGAffineTransformFromString()`. Here’s what the values look like when logged for a transform that’s scaled by a factor of 1.5 and rotated by 45 degrees:

```
2013-03-31 09:43:20.837 HelloWorld[41450:c07]
[1.06066, 1.06066, -1.06066, 1.06066, 0, 0]
```

This particular transform skips any center reorientation, so you’re only looking at these two operations.

These raw numbers aren’t especially meaningful. Specifically, this representation does not tell you directly exactly how much the transform scales or rotates. Fortunately, there’s an easy way around this, a way of transforming unintuitive parameters into more helpful representations. Listing 2-1 shows how. It calculates x- and y-scale values as well as rotation, returning these values from the components of the transform structure.

There’s never any need to calculate the translation (position offset) values. These values are stored for you directly in the `tx` and `ty` fields, essentially in “plain text.”

Listing 2-1 Extracting Scale and Rotation from Transforms

```
// Extract the x scale from transform
CGFloat TransformGetXScale(CGAffineTransform t)
{
    return sqrt(t.a * t.a + t.c * t.c);
}

// Extract the y scale from transform
CGFloat TransformGetYScale(CGAffineTransform t)
{
    return sqrt(t.b * t.b + t.d * t.d);
}

// Extract the rotation in radians
CGFloat TransformGetRotation(CGAffineTransform t)
{
    return atan2f(t.b, t.a);
}
```

Predefined Constants

Every Core Graphics structure has predefined constants. The most common ones you'll encounter when working in Quartz are as follows:

- **Geometric zeros**—These constants provide defaults with zero values. `CGPointZero` is a point constant that refers to the location (0, 0). The `CGSizeZero` constant references an extent whose width and height are 0. `CGRectZero` is equivalent to `CGRectMake(0, 0, 0, 0)`, with a zero origin and size.
- **Geometric identity**—`CGAffineTransformIdentity` supplies a constant identity transform. When applied to any geometric element, this transform returns the same element you started with.
- **Geometric infinity**—`CGRectInfinite` is a rectangle with infinite extent. The width and heights are set to `CGFLOAT_MAX`, the highest possible floating-point value. The origin is set to the most negative number possible (`-CGFLOAT_MAX/2`).
- **Geometric nulls**—`CGRectNull` is distinct from `CGRectZero` in that it has no position. `CGRectZero`'s position is (0, 0). `CGRectNull`'s position is (`INFINITY, INFINITY`). You encounter this rectangle whenever you request the intersection of disjoint rectangles.

The union of any rectangle with `CGRectNull` always returns the original `rect`. For example, the union of `CGRectMake(10, 10, 10, 10)` with `CGRectNull` is `\{\{10, 10\}, {10, 10\}\}`. Contrast this with the union with `CGRectZero`, which is `\{\{0, 0\}, {20, 20\}\}`.

instead. The origin is pulled toward (0, 0), doubling the rectangle's size and changing its position.

Conversion to Objects

Because geometric items are structures and not objects, integrating them into standard Objective-C can be challenging. You cannot add a size or a rectangle to an `NSArray` or a dictionary. You can't set a default value to a point or a transform. Because of this, Core Graphics and Core Foundation provide functions and classes to convert and encapsulate structures within objects. The most common object solutions for geometric structures are strings, dictionaries, and values.

Strings

You morph structures to string representation and back by using a handful of conversion functions, which are listed in Table 2-2. These convenience functions enable you to log information in human-readable format or store structures to files using a well-defined and easy-to-recover pattern. A converted point or size (for example, `NSStringFromCGPoint(CGPointMake(5, 2))`) looks like this: {5, 2}.

Strings are most useful for logging information to the debugging console.

Table 2-2 String Utility Functions

Type	Convert to String	Convert from String
<code>CGPoint</code>	<code>NSStringFromCGPoint()</code>	<code>CGPointFromString()</code>
<code>CGSize</code>	<code>NSStringFromCGSize()</code>	<code>CGSizeFromString()</code>
<code>CGRect</code>	<code>NSStringFromCGRect()</code>	<code>CGRectFromString()</code>
<code>CGAffineTransform</code>	<code>NSStringFromCGAffineTransform()</code>	<code>CGAffineTransformFromString()</code>

Dictionaries

Dictionaries provide another way to transform geometry structures to and from objects for storage and logging. As Table 2-3 reveals, these are limited to points, sizes, and rectangles. What's more, they return Core Foundation `CFDictionaryRef` instances, not `NSDictionary` items. You need to bridge the results.

Dictionaries are most useful for storing structures to user defaults.

Table 2-3 Dictionary Utility Functions

Type	Convert to Dictionary	Convert from Dictionary
CGPoint	CGPointCreateDictionaryRepresentation() ()	CGPointMakeWithDictionaryRepresentation() ()
CGSize	CGSizeCreateDictionaryRepresentation()	CGSizeMakeWithDictionaryRepresentation()
CGRect	CGRectCreateDictionaryRepresentation()	CGRectMakeWithDictionaryRepresentation()

The following code converts a rectangle to a Cocoa Touch dictionary representation and then back to a CGRect, logging the results along the way:

```
// Convert CGRect to NSDictionary representation
CGRect testRect = CGRectMake(1, 2, 3, 4);
NSDictionary *dict = ((__bridge_transfer NSDictionary *)  

    CGRectMakeDictionaryRepresentation(testRect));
NSLog(@"Dictionary: %@", dict);

// Convert NSDictionary representation to CGRect
CGRect outputRect;
BOOL success = CGRectMakeMakeWithDictionaryRepresentation(
    ((__bridge CFDictionaryRef) dict, &outputRect);
if (success)
    NSLog(@"Rect: %@", NSStringFromCGRect(outputRect));
```

Here is the output this code produces. In it you see the dictionary representation for the test rectangle and the restored rectangle after being converted back to a CGRect:

```
2013-04-02 08:23:07.323 HelloWorld[62600:c07] Dictionary: {  

    Height = 4;  

    Width = 3;  

    X = 1;  

    Y = 2;  

}  

2013-04-02 08:23:07.323 HelloWorld[62600:c07]  

    Rect: {{1, 2}, {3, 4}}
```

Note

The CGRectMakeMakeWithDictionaryRepresentation() function has a known bug that makes some values created on OS X unreadable on iOS.

Values

The `NSValue` class provides a container for C data items. It can hold scalar types (like integers and floats), pointers, and structures. UIKit extends normal `NSValue` behavior to encapsulate Core Graphics primitives. When a scalar type is placed into a value, you can add geometric primitives to Objective-C collections and treat them like any other object.

Values are most useful for adding structures to arrays and dictionaries for in-app algorithms.

Table 2-4 Value Methods

Type	Place in <code>NSValue</code>	Retrieve from <code>NSValue</code>
<code>CGPoint</code>	<code>+ valueWithCGPoint:</code>	<code>- CGPointValue</code>
<code>CGSize</code>	<code>+ valueWithCGSize:</code>	<code>- CGSizeValue</code>
<code>CGRect</code>	<code>+ valueWithCGRect:</code>	<code>- CGRectValue</code>
<code>CGAffineTransform</code>	<code>+ valueWithCGAffineTransform:</code>	<code>- CGAffineTransformValue</code>

Note

`NSCoder` provides UIKit-specific geometric encoding and decoding methods for `CGPoint`, `CGSize`, `CGRect`, and `CGAffineTransform`. These methods enable you to store and recover these geometric structures, associating them with a key you specify.

Geometry Tests

Core Graphics offers a basic set of geometry testing functions, which you see listed in Table 2-5. These items enable you to compare items against each other and check for special conditions that you may encounter. These functions are named to be self-explanatory.

Table 2-5 Checking Geometry

Topic	Functions
Special conditions	<code>CGRectIsEmpty(rect)</code> <code>CGRectIsNull(rect)</code> <code>CGRectIsInfinite(rect)</code> <code>CGAffineTransformIsIdentity(transform)</code>

Equality	<code>CGPointEqualToPoint(p1, p2)</code> <code>CGSizeEqualSize(s1, s2)</code> <code>CGRectEqualRect(r1, r2)</code> <code>CGAffineTransformEqualToTransform(t1, t2)</code>
Relationships	<code>CGRectContainsPoint(rect, point)</code> <code>CGRectContainsRect(r1, r2)</code> <code>CGRectIntersectsRect(r1, r2)</code>

Other Essential Functions

Here are a few final functions you'll want to know about:

- **CGRectInset(rect, xinset, yinset)**—This function enables you to create a smaller or larger rectangle that's centered on the same point as the source rectangle. Use a positive inset for smaller rectangles, negative for larger ones. *This function is particularly useful for moving drawings and subimages away from view edges to provide whitespace breathing room.*
- **CGRectOffset(rect, xoffset, yoffset)**—This function returns a rectangle that's offset from the original rectangle by an x and y amount you specify. *Offsets are handy for moving frames around the screen and for creating easy drop-shadow effects.*
- **CGRectGetMidX(rect)** and **CGRectGetMidY(rect)**—These functions recover the x and y coordinates in the center of a rectangle. These functions make it very convenient to recover the midpoints of bounds and frames. Related functions return `minX`, `maxX`, `minY`, `maxY`, `width`, and `height`. *Midpoints help center items in a drawing.*
- **CGRectUnion(rect1, rect2)**—This function returns the smallest rectangle that entirely contains both source rectangles. This function helps you establish the minimum bounding box for distinct elements you're drawing. *The union of drawing paths lets you frame items together and build backsplashes for just those items.*
- **CGRectIntersection(rect1, rect2)**—This function returns the intersection of two rectangles or, if the two do not intersect, `CGRectNull`. *Intersections help calculate rectangle insets when working with Auto Layout. The intersections between a path's bounds and an image frame can define the intrinsic content used for alignment.*
- **CGRectIntegral(rect)**—This function converts the source rectangle to integers. The origin values are rounded down from any fractional values to whole integers. The size is rounded upward. You are guaranteed that the new rectangle fully contains the original rectangle. *Integral rectangles speed up and clarify your drawing. Views drawn exactly on pixel boundaries require less antialiasing and result in less blurry output.*

- `CGRectStandardize(rect)`—This function returns an equivalent rectangle with positive height and width values. *Standardized rectangles simplify your math when you're performing interactive drawing, specifically when users may interactively move left and up instead of right and down.*
- `CGRectDivide(rect, &sliceRect, &remainderRect, amount, edge)`—This function is the most complicated of these Core Graphics functions, but it is also among the most useful. *Division enables you to iteratively slice a rectangle into portions, so you can subdivide the drawing area.*

Using `CGRectDivide()`

The `CGRectDivide()` function is terrifically handy. It provides a really simple way to divide and subdivide a rectangle into sections. At each step, you specify how many to slice away and which side to slice it away from. You can cut from any edge, namely `CGRectMinXEdge`, `CGRectMinYEdge`, `CGRectMaxXEdge`, and `CGRectMaxYEdge`.

A series of these calls built the image in Figure 2-3. Listing 2-2 shows how this was done. The code slices off the left edge of the rectangle and then divides the remaining portion into two vertical halves. Removing two equal sections from the left and the right further decomposes the bottom section.



Figure 2-3 You can subdivide rectangles by iteratively slicing off sections.

Listing 2-2 Creating a Series of Rectangle Divisions

```
UIBezierPath *path;
CGRect remainder;
CGRect slice;

// Slice a section off the left and color it orange
CGRectDivide(rect, &slice, &remainder, 80, CGRectGetMinXEdge);
[[UIColor orangeColor] set];
path = [UIBezierPath bezierPathWithRect:slice];
[path fill];

// Slice the other portion in half horizontally
rect = remainder;
CGRectDivide(rect, &slice, &remainder,
    remainder.size.height / 2, CGRectGetMinYEdge);

// Tint the sliced portion purple
[[UIColor purpleColor] set];
path = [UIBezierPath bezierPathWithRect:slice];
[path fill];

// Slice a 20-point segment from the bottom left.
// Draw it in gray
rect = remainder;
CGRectDivide(rect, &slice, &remainder, 20, CGRectGetMinXEdge);
[[UIColor grayColor] set];
path = [UIBezierPath bezierPathWithRect:slice];
[path fill];

// And another 20-point segment from the bottom right.
// Draw it in gray
rect = remainder;
CGRectDivide(rect, &slice, &remainder, 20, CGRectGetMaxXEdge);
// Use same color on the right
path = [UIBezierPath bezierPathWithRect:slice];
[path fill];

// Fill the rest in brown
[[UIColor brownColor] set];
path = [UIBezierPath bezierPathWithRect:remainder];
[path fill];
```

Rectangle Utilities

You use the `CGRectMake()` function to build frames, bounds, and other rectangle arguments. It accepts four floating-point arguments: `x`, `y`, `width`, and `height`. This is one of the most important functions you use in Quartz drawing.

There are often times when you'll want to construct a rectangle from the things you usually work with: points and size. Although you can use component fields to retrieve arguments, you may find it helpful to have a simpler function on hand—one that is specific to these structures. Listing 2-3 builds a rectangle from point and size structures.

Listing 2-3 Building Rectangles from Points and Sizes

```
CGRect RectMakeRect(CGPoint origin, CGSize size)
{
    return (CGRect){.origin = origin, .size = size};
}
```

Quartz, surprisingly, doesn't supply a built-in routine to retrieve a rectangle's center. Although you can easily pull out `x` and `y` midpoints by using Core Graphics functions, it's handy to implement a function that returns a point directly from the `rect`. Listing 2-4 defines that function. It's one that I find myself using extensively in drawing work.

Listing 2-4 Retrieving a Rectangle Center

```
CGPoint RectGetCenter(CGRect rect)
{
    return CGPointMake(CGRectGetMidX(rect),
                      CGRectGetMidY(rect));
}
```

Building a rectangle around a center is another common challenge. For example, you might want to center text or place a shape around a point. Listing 2-5 implements this functionality. You supply a center and a size. It returns a rectangle that describes your target placement.

Listing 2-5 Creating a Rectangle Around a Target Point

```
CGRect RectAroundCenter(CGPoint center, CGSize size)
{
    CGFloat halfWidth = size.width / 2.0f;
    CGFloat halfHeight = size.height / 2.0f;
```

```
        return CGRectMake(center.x - halfWidth,
                           center.y - halfHeight, size.width, size.height);
    }
```

Listing 2-6 uses Listings 2-4 and 2-5 to draw a string centered in a given rectangle. It calculates the size of the string (using iOS 7 APIs) and the rectangle center and builds a target around that center. Figure 2-4 shows the result.

Listing 2-6 Centering a String

```
NSString *string = @"Hello World";
UIFont *font =
    [UIFont fontWithName:@"HelveticaNeue" size:48];

// Calculate string size
CGSize stringSize =
    [string sizeWithAttributes:{NSFontAttributeName:font}];

// Find the target rectangle
CGRect target =
    RectAroundCenter(RectGetCenter(grayRectangle), stringSize);

// Draw the string
[greenColor set];
[string drawInRect:target withFont:font];
```



Figure 2-4 Drawing a string into the center of a rectangle. The dashed line indicates the outline of the target rectangle.

Another way to approach the same centering problem is shown in Listing 2-7. It takes an existing rectangle and centers it within another rectangle rather than working with a size and a target point. You might use this function when working with a Bezier path whose `bounds` property returns a bounding rectangle. You can center that path in a rectangle by calling `RectCenteredInRect()`.

Listing 2-7 Centering a Rectangle

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

Fitting and Filling

Often you need to resize a drawing to fit into a smaller or larger space than its natural size. To accomplish this, you calculate a target for that drawing, whether you’re working with paths, images, or context drawing functions. There are four basic approaches you take to accomplish this: centering, fitting, filling, and squeezing.

If you've worked with view content modes, these approaches may sound familiar. When drawing, you apply the same strategies that UIKit uses for filling views with content. The corresponding content modes are center, scale aspect fit, scale aspect fill, and scale to fill.

Centering

Centering places a rectangle at its natural scale, directly in the center of the destination. Material larger than the pixel display area is cropped. Material smaller than that area is matted. (Leaving extra area on all four sides of the drawing is called *windowboxing*.) Figure 2-5 shows two sources, one larger than the destination and the other smaller.

In each case, the items are centered using `RectAroundCenter()` from Listing 2-5. Since the outer figure is naturally larger, it would be cropped when drawn to the gray rectangle. The smaller inner figure would be matted (padded) on all sides with extra space.

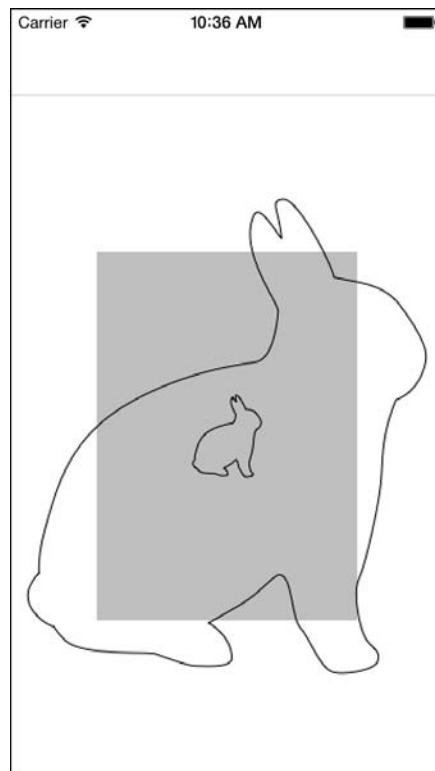


Figure 2-5 Centering a rectangle leaves its original size unchanged. Accordingly, cropping and padding may or may not occur.

Fitting

When you fit items into a target, you retain the original proportions of the source rectangle and every part of the source material remains visible. Depending on the original aspect ratio, the results are either letterboxed or pillarboxed, with some extra area needed to mat the image.

In the physical world, matting refers to a backdrop or border used in framing pictures. The backdrop or border adds space between the art and the physical frame. Here, I use the term *matting* to mean the extra space between the edges of the drawing area and the extent of the destination rectangle.

Figure 2-6 shows items fit into target destinations. The gray background represents the destination rectangle. The light purple background shows the fitting rectangle calculated by Listing 2-8. Unless the aspects match exactly, the drawing area will be centered, leaving extra space on the top and bottom or on the sides.



Figure 2-6 Fitting into the destination preserves the original aspect but may leave letterbox (top) or pillarbox (bottom) extra spaces.

Listing 2-8 Calculating a Destination by Fitting to a Rectangle

```
// Multiply the size components by the factor
CGSize SizeScaleByFactor(CGSize aSize, CGFloat factor)
{
    return CGSizeMake(aSize.width * factor,
                      aSize.height * factor);
}

// Calculate scale for fitting a size to a destination
CGFloat AspectScaleFit(CGSize sourceSize, CGRect destRect)
{
    CGSize destSize = destRect.size;
    CGFloat scaleW = destSize.width / sourceSize.width;
    CGFloat scaleH = destSize.height / sourceSize.height;
    return MIN(scaleW, scaleH);
}

// Return a rect fitting a source to a destination
CGRect RectByFittingInRect(CGRect sourceRect,
                           CGRect destinationRect)
{
    CGFloat aspect =
        AspectScaleFit(sourceRect.size, destinationRect);
    CGSize targetSize =
        SizeScaleByFactor(sourceRect.size, aspect);
    return RectAroundCenter(
        RectGetCenter(destinationRect), targetSize);
}
```

Filling

Filling, which is shown in Figure 2-7, ensures that every pixel of the destination space corresponds to the drawing area within the source image. Not to be confused with the drawing “fill” operation (which fills a path with a specific color or pattern), this approach avoids the excess pixel areas you saw in Figure 2-6.

To accomplish this, filling grows or shrinks the source rectangle to exactly cover the destination. It crops any elements that fall outside the destination, either to the top and bottom, or to the left and right. The resulting target rectangle is centered, but only one dimension can be fully displayed.

Listing 2-9 calculates the target rectangle, and Figure 2-7 shows the results. In the left image, the ears and feet would be cropped. In the right image, the nose and tail would be cropped. Only images that exactly match the target’s aspect remain uncropped. Filling trades off cropping against fitting’s letterboxing.

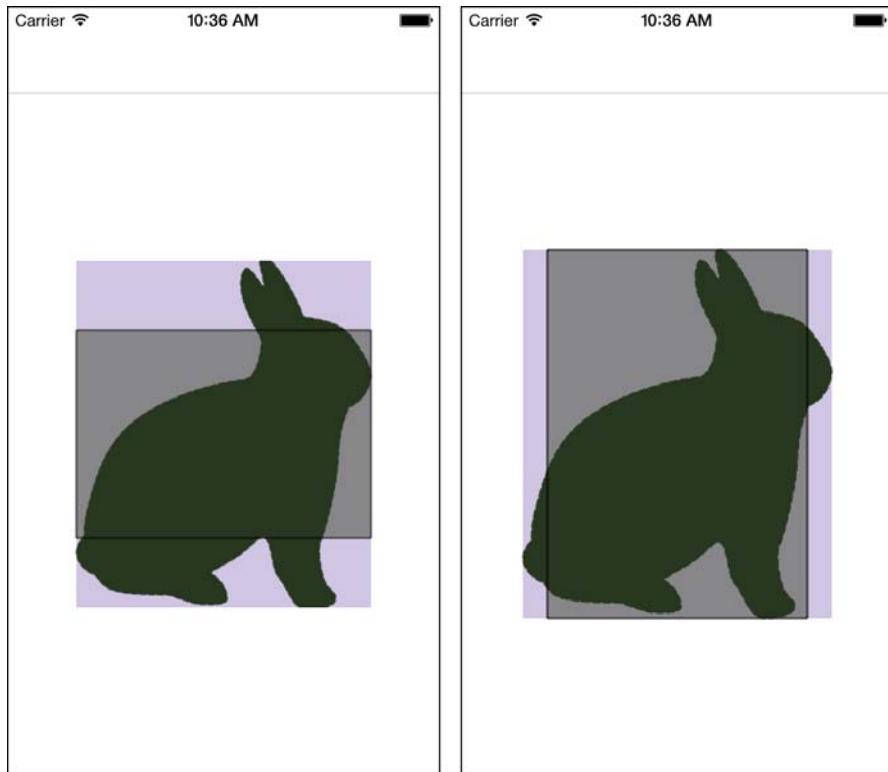


Figure 2-7 Filling a destination ensures that the target rectangle covers every pixel.

Listing 2-9 Calculating a Destination by Filling a Rectangle

```
// Calculate scale for filling a destination
CGFloat AspectScaleFill(CGSize sourceSize, CGRect destRect)
{
    CGSize destSize = destRect.size;
    CGFloat scaleW = destSize.width / sourceSize.width;
    CGFloat scaleH = destSize.height / sourceSize.height;
    return MAX(scaleW, scaleH);
}

// Return a rect that fills the destination
CGRect RectByFillingRect(CGRect sourceRect, CGRect destinationRect)
{
    CGFloat aspect = AspectScaleFill(sourceRect.size, destinationRect);
```

```
CGSize targetSize = SizeScaleByFactor(sourceRect.size, aspect);
return RectAroundCenter(RectGetCenter(destinationRect), targetSize);
}
```

Squeezing

Squeezing adjusts the source's aspect to fit all available space within the destination. All destination pixels correspond to material from the source rectangle. As Figure 2-8 shows, the material resizes along one dimension to accommodate. Here, the bunny squeezes horizontally to match the destination rectangle.

Unlike with the previous fitting styles, with squeezing, you don't have to perform any calculations. Just draw the source material to the destination rectangle and let Quartz do the rest of the work.

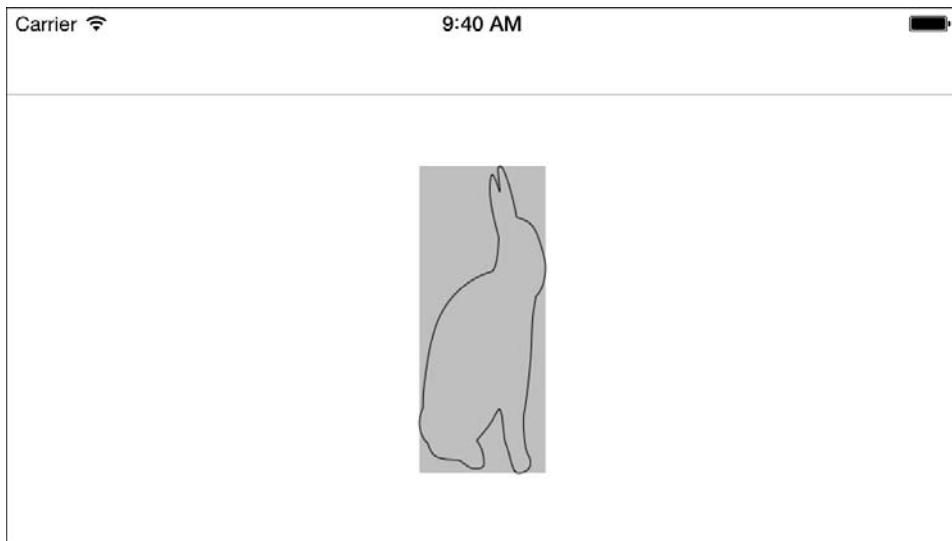


Figure 2-8 Squeezing a source rectangle ignores its original aspect ratio, drawing to the destination's aspect.

Summary

This chapter introduces basic terminology, data types, and manipulation functions for Core Graphics drawing. You read about the difference between points and pixels, explored common data types, and learned ways to calculate drawing positions. Here are a few final thoughts to carry on with you as you leave this chapter.

- Always write your code with an awareness of screen scale. Relatively modern devices like the iPad 2 and the first-generation iPad mini do not use Retina displays. Focusing on logical space (points) instead of device space (pixels) enables you to build flexible code that supports upgrades and new device geometries.
- A lot of drawing is relative, especially when you’re working with touches on the screen. Learn to migrate your points from one view coordinate system to another, so your drawing code can use the locations you intend instead of the point locations you inherit.
- Affine transforms are often thought of as the “other” Core Graphics struct, but they play just as important a role as their more common cousins—the points, sizes, and rectangles. Transforms are wicked powerful and can simplify drawing tasks in many ways. If you can spare the time, you’ll find it’s a great investment of your effort to learn about transforms and the matrix math behind them. Khan Academy (www.khanacademy.org) offers excellent tutorials on this subject.
- A robust library of routine geometric functions can travel with you from one project to the next. Basic geometry never goes out of style, and the same kinds of tasks occur over and over again. Knowing how to center rectangles within one another and how to calculate fitting and filling destinations are skills that will save you time and effort in the long run.

This page intentionally left blank

3

Drawing Images

This chapter introduces image drawing. It surveys techniques for creating, adjusting, and retrieving image instances. You can do a lot with images, drawing, and iOS. You can render items into graphics contexts, building altered versions. You can produce thumbnail versions or extract portions of the original. You can create items that know how to properly stretch for buttons and others that work seamlessly with Auto Layout. Contexts provide ways to convert image instances to and from data representations. This enables you to apply image-processing techniques and incorporate the results into your interfaces. In this chapter, you'll read about common image drawing challenges and discover the solutions you can use.

UIKit Images

UIKit images center around the `UIImage` class. It's a powerful and flexible class that hides its implementation details, enabling you to perform many presentation tasks with a minimum of code. Its most common pattern involves loading data from a file and adding the resulting image to a `UIImageView` instance. Here's an example of this:

```
UIImage *image = [UIImage imageNamed:@"myImage"];
myImageView.image = image;
```

You are not limited to loading images from external data, of course. iOS enables you to create your own images from code when and how you need to. Listing 3-1 shows a trivial example. It builds a new `UIImage` instance, using a color and size you specify. This produces a color swatch that is returned from the function.

To accomplish this, Listing 3-1 starts by establishing an image context. It then sets a color and fills the context using `UIRectFill()`. It concludes by retrieving and returning a new image from the context.

Listing 3-1 demonstrates the basic drawing skeleton. Where this function draws a colored rectangle, you can create your own Mona Lisas. Just supply your own custom drawing routines and set the target drawing size to the extent your app demands.

Listing 3-1 Creating a Custom Image

```
UIImage *SwatchWithColor(UIColor *color, CGFloat side)
{
    // Create image context (using the main screen scale)
    UIGraphicsBeginImageContextWithOptions(
        CGSizeMake(side, side), YES, 0.0);

    // Perform drawing
    [color setFill];
    UIRectFill(CGRectMake(0, 0, side, side));

    // Retrieve image
    UIImage *image =
        UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return image;
}
```

Here are a few quick things you'll want to remember about images:

- You query an image for its extent by inspecting its `size` property, as in this example. The size is returned in points rather than pixels, so data dimensions may be double the number returned on Retina systems:

```
UIImage *swatch = SwatchWithColor(greenColor, 120);
 NSLog(@"%@", NSStringFromCGSize(swatch.size));
```
- You transform an image instance to PNG or JPEG data by using the `UIImagePNGRepresentation()` function or the `UIImageJPEGRepresentation()` function. These functions return `NSData` instances containing the compressed image data.
- You can retrieve an image's Quartz representation through its `CGImage` property. The `UIImage` class is basically a lightweight wrapper around Core Graphics or Core Image images. You need a `CGImage` reference for many Core Graphics functions. Because this property is not available for images created using Core Image, you must convert the underlying `CIIImage` into a `CGImage` for use in Core Graphics.

Note

`UIImage` supports TIFF, JPEG, GIF, PNG, DIB (that is, BMP), ICO, CUR, and XBM formats. You can load additional formats (like RAW) by using the `ImageIO` framework.

Building Thumbnails

By producing thumbnails, you convert large image instances into small ones. Thumbnails enable you to embed versions into table cells, contact summaries, and other situations where images provide an important supporting role. Chapter 2 introduced functions that calculate destination aspect. Thumbnails provide a practical, image-oriented use case for these, as well as a good jumping-off point for simple image drawing.

You build thumbnails by creating an image context sized as desired, such as 100 by 100. Use `drawInRect:` to draw the source into the context. Finish by retrieving your new thumbnail.

```
UIImage *image = [UIImage imageNamed:@"myImage"];
[image drawInRect: destinationRect];
UIImage *thumbnail =
    UIGraphicsGetImageFromCurrentImageContext();
```

The key to proper image thumbnails is aspect. Regardless of whether you fit or fill, you'll want to preserve the image's internal features without distortion. Figure 3-1 shows a photograph of a cliff. The image is taller than it is wide—specifically, it is 1,933 pixels wide by 2,833 pixels high. The thumbnail shown on the right was drawn without any concern for aspect. Because the source is taller than it is wide, the result is vertically squeezed.

It's not a bad-looking result—in fact, if you didn't have the reference on the left, you might not notice any issues at all—but it's not an accurate result. Using an image like this helps showcase the perils of incorrect thumbnail production. Errors may not jump out at you for your test data, but they assuredly will for users when applied to more scale-sensitive data like human faces.



Figure 3-1 The thumbnail on the right is vertically compressed, providing an inaccurate representation of the original image. Pay attention to image aspect when drawing thumbnails to avoid squeezing. *Public domain images courtesy of the National Park Service.*

The images in Figure 3-2 demonstrate what proper thumbnails should look like. Instead of drawing directly into the target, they calculate rectangles that fill (left) or fit (right) the target area. Listing 3-2 reveals the difference in approach, creating a fitting or filling rectangle to draw to instead of using the target `rect` directly.



Figure 3-2 Filling (left) and fitting (right) create thumbnail representations that preserve aspect. *Public domain images courtesy of the National Park Service.*

Listing 3-2 Building a Thumbnail Image

```
UIImage *BuildThumbnail(UIImage *sourceImage,
    CGSize targetSize, BOOL useFitting)
{
    UIGraphicsBeginImageContextWithOptions(
        targetSize, NO, 0.0);

    // Establish the output thumbnail rectangle
    CGRect targetRect = SizeMakeRect(targetSize);

    // Create the source image's bounding rectangle
    CGRect naturalRect = (CGRect){.size = sourceImage.size};

    // Calculate fitting or filling destination rectangle
    // See Chapter 2 for a discussion on these functions
    CGRect destinationRect = useFitting ?
        RectByFittingRect(naturalRect, targetRect) :
        RectByFillingRect(naturalRect, targetRect);

    // Draw the new thumbnail
    [sourceImage drawInRect:destinationRect];

    // Retrieve and return the new image
    UIImage *thumbnail =
```

```
    UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return thumbnail;
}
```

Extracting Subimages

Unlike thumbnails, which compress image data to a smaller version, subimages retrieve portions of an image at the same resolution as the original. Figure 3-3 shows a detail version of a ferret's head, extracted from the image at the top left. The enlarged subimage highlights the extracted portion. As you see, you cannot add resolution; the result grows fuzzier as you zoom in.

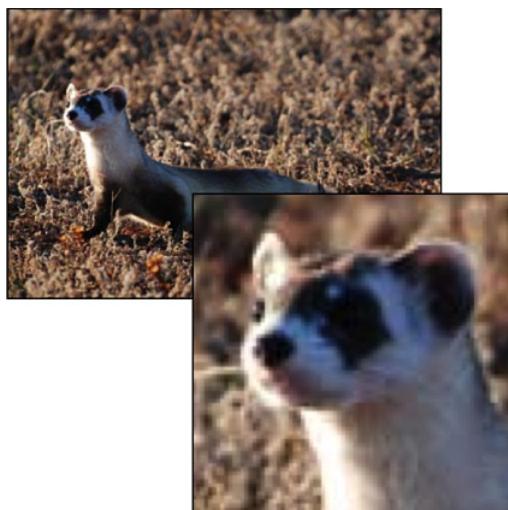


Figure 3-3 The bottom image is a subimage extract of a portion from the original image. *Public domain images courtesy of the National Park Service.*

Listing 3-3 shows the function that created this subimage. It uses the simple Quartz function `CGImageCreateWithImageInRect()` to build its new image from the content of the original. Going with Quartz instead of UIKit in this case provides a ready-built single function to work with.

When you use this Core Graphics function, the rectangle is automatically adjusted on your behalf to pixel lines using `CGRectIntegral()`. It's then intersected with the natural image rectangle, so no portions of the subrectangle fall outside the original image bounds. This saves you a lot of work. All you have to do is convert the `CGImageRef` returned by the function into a `UIImage` instance.

A drawback occurs when your reference rectangle is defined for a Retina system, and you’re extracting data in terms of Quartz coordinates. For this reason, I’ve included a second function in Listing 3-3, one that operates entirely in UIKit. Rather than convert rectangles between coordinate systems, this function assumes that the rectangle you’re referencing is defined in points, not pixels. This is important when you’re asking an image for its bounds and then building a rectangle around its center. That “center” for a Retina image may actually be closer to its top-left corner if you’ve forgotten to convert from points to pixels. By staying in UIKit, you sidestep the entire issue, making sure the bit of the picture you’re extracting is the portion you really meant.

Listing 3-3 Extracting Portions of Images

```
UIImage *ExtractRectFromImage(
    UIImage *sourceImage, CGRect subRect)
{
    // Extract image
    CGImageRef imageRef = CGImageCreateWithImageInRect(
        sourceImage.CGImage, subRect);
    if (imageRef != NULL)
    {
        UIImage *output = [UIImage imageWithCGImage:imageRef];
        CGImageRelease(imageRef);
        return output;
    }

    NSLog(@"Error: Unable to extract subimage");
    return nil;
}

// This is a little less flaky
// when moving to and from Retina images
UIImage *ExtractSubimageFromRect(
    UIImage *sourceImage, CGRect rect)
{
    UIGraphicsBeginImageContextWithOptions(rect.size, NO, 1);
    CGRect destRect = CGRectMake(
        -rect.origin.x, -rect.origin.y,
        sourceImage.size.width, sourceImage.size.height);
    [sourceImage drawInRect:destRect];
    UIImage *newImage =
        UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return newImage;
}
```

Converting an Image to Grayscale

Figure 3-4 shows an image of a black bear. The center of this picture is a grayscale representation of the bear image, with all color removed.

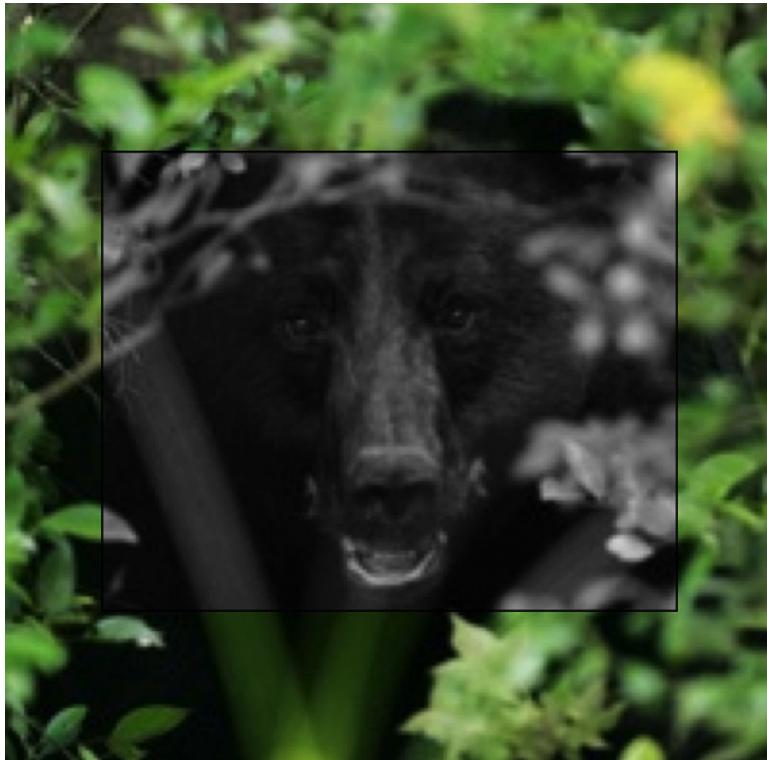


Figure 3-4 Color spaces enable you to convert images to grayscale. *Public domain images courtesy of the National Park Service.*

To create this figure, I drew the black bear image twice. The first time, I drew the entire image. The second, time I clipped the context to the inner rectangle and converted the image to grayscale. I drew the new version on top of the original and added a black border around the grayscale image.

Here are the steps involved:

```
// Clip the context
CGContextSaveGState(context);
CGRect insetRect = RectInsetByPercent(destinationRect, 0.40);
UIRectClip(insetRect);
```

```

// Draw the grayscale version
[GrayscaleVersionOfImage(sourceImage)
    drawInRect:destinationRect];
CGContextRestoreGState(context);

// Outline the border between the two versions
UIRectFrame(insetRect);

```

The details of the grayscale conversion appear in Listing 3-4. The `GrayscaleVersionOfImage()` function works by building a new context using the source image dimensions. This “device gray”-based context stores 1 byte per pixel and uses no alpha information. It forms a new drawing area that can only handle grayscale results.

In the real world, when you draw with a purple crayon, the mark that appears on any paper will be purple. A grayscale drawing context is like black-and-white photo film. No matter what color you draw to it, the results will always be gray, matching the brightness of the crayon you’re drawing with but discarding the hue.

As with all contexts, it’s up to you to retrieve the results and store the data to an image. Here, the function draws the source image and retrieves a grayscale version, using `CGBitmapContextCreateImage()`. This call is analogous to the `UIGraphicsGetImageFromCurrentImageContext()` function for bitmap contexts, with a bit more memory management required.

You end up with a `UIImage` instance that retains the original’s luminance values but not its colors. Quartz handles all the detail work on your behalf. You don’t need to individually calculate brightness levels from each set of source pixels. You just specify the characteristics of the destination (its size and its color space), and the rest is done for you. This is an extremely easy way to work with images.

Listing 3-4 Building a Grayscale Version of an Image

```

UIImage *GrayscaleVersionOfImage(UIImage *sourceImage)
{
    // Establish grayscale color space
    CGColorSpaceRef colorSpace =
        CGColorSpaceCreateDeviceGray();
    if (colorSpace == NULL)
    {
        NSLog(@"Error creating grayscale color space");
        return nil;
    }

    // Extents are integers
    int width = sourceImage.size.width;
    int height = sourceImage.size.height;

```

```

// Build context: one byte per pixel, no alpha
CGContextRef context = CGBitmapContextCreate(
    NULL, width, height,
    8, // 8 bits per byte
    width, colorSpace,
    (CGBitmapInfo) kCGImageAlphaNone);
CGColorSpaceRelease(colorSpace);
if (context == NULL)
{
    NSLog(@"Error building grayscale bitmap context");
    return nil;
}

// Replicate image using new color space
CGRect rect = SizeMakeRect(sourceImage.size);
CGContextDrawImage(
    context, rect, sourceImage.CGImage);
CGImageRef imageRef =
    CGBitmapContextCreateImage(context);
CGContextRelease(context);

// Return the grayscale image
UIImage *output = [UIImage imageWithCGImage:imageRef];
CFRelease(imageRef);
return output;
}

```

Watermarking Images

Watermarking is a common image drawing request. Originally, watermarks were faint imprints in paper used to identify the paper's source. Today's text watermarks are used differently. They're added over an image either to prevent copying and reuse or to brand the material with a particular hallmark or provenance.

Listing 3-5 shows how to create the simple text watermark shown in Figure 3-5. Watermarking involves nothing more than drawing an image and then drawing something else—be it a string, a logo, or a symbol—over that image and retrieving the new version.



Figure 3-5 A text watermark overlays images with words, logos, or symbols.
Public domain images courtesy of the National Park Service.

The example in Listing 3-5 draws its string (“watermark”) diagonally across the image source. It does this by rotating the context by 45 degrees. It uses a blend mode to highlight the watermark while preserving details of the original photo. Because this listing is specific to iOS 7, you must include a text color along with the font attribute when drawing the string. If you do not, the string will “disappear,” and you’ll be left scratching your head—as I did when updating this example.

Other common approaches are using diffuse white overlays with a moderate alpha level and drawing just a logo’s shadow (without drawing the logo itself) onto some part of an image. Path clipping helps with that latter approach; it is discussed in more detail in Chapter 5.

Each watermark approach changes the original image in different ways. As image data becomes changed or obscured, removing the watermark becomes more difficult.

Listing 3-5 Watermarking an Image

```
UIGraphicsBeginImageContextWithOptions(
    targetSize, NO, 0.0);
CGContextRef context = UIGraphicsGetCurrentContext();

// Draw the original image into the context
CGRect targetRect = CGRectMake(0, 0, targetSize.width, targetSize.height);
UIImage *sourceImage = [UIImage imageNamed:@"pronghorn.jpg"];
CGRect imgRect = CGRectMake(0, 0, sourceImage.size.width, sourceImage.size.height);

// Set the text color and font
UIColor *textColor = [UIColor blueColor];
UIFont *font = [UIFont boldSystemFontOfSize:24];

// Create a string
NSString *text = @"Watermark";
CGContextSaveGState(context);
CGContextTranslateCTM(context, targetRect.origin);
CGContextRotateCTM(context, -45.0);
CGContextSetBlendMode(context, kCGBlendModeMultiply);
CGContextSetTextFont(context, font);
CGContextSetTextColor(context, textColor.CGColor);
CGContextShowTextAtPoint(context, targetRect.size.width / 2.0, targetRect.size.height / 2.0, text);
CGContextRestoreGState(context);

[sourceImage drawInRect:imgRect];
```

```

// Rotate the context
CGPoint center = RectGetCenter(targetRect);
CGContextTranslateCTM(context, center.x, center.y);
CGContextRotateCTM(context, M_PI_4);
CGContextTranslateCTM(context, -center.x, -center.y);

// Create a string
NSString *watermark = @"watermark";
UIFont *font =
    [UIFont fontWithName:@"HelveticaNeue" size:48];
CGSize size = [watermark sizeWithAttributes:
    @{@"NSFontAttributeName": font}];
CGRect stringRect = RectCenteredInRect(
    SizeMakeRect(size), targetRect);

// Draw the string, using a blend mode
CGContextSetBlendMode(context, kCGBlendModeDifference);
[watermark drawInRect:stringRect withAttributes:
    @{@"NSFontAttributeName": font,
        NSForegroundColorAttributeName:[UIColor whiteColor]}];

// Retrieve the new image
UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
return image;

```

Retrieving Image Data

Although you can query an image for its PNG (`UIImagePNGRepresentation()`) or JPEG (`UIImageJPEGRepresentation ()`) representations, these functions return data suitable for storing images to file formats. They include file header and marker data, internal chunks, and compression. The data isn't meant for byte-by-byte operations. When you plan to perform image processing, you'll want to extract byte arrays from your contexts. Listing 3-6 shows how.

This function draws an image into a context and then uses `CGBitmapContextGetData()` to retrieve the source bytes. It copies those bytes into an `NSData` instance and returns that instance to the caller.

Wrapping the output data into an `NSData` object enables you to bypass issues regarding memory allocation, initialization, and management. Although you may end up using this data in C-based APIs like Accelerate, you're able to do so from an Objective-C viewpoint.

Note

The byte processing discussed here is not meant for use with Core Image, which has its own set of techniques and practices.

Creating Contexts

You've already met the `CGBitmapContextCreate()` function several times in this book. It requires seven arguments that define how the context should be created. For the most part, you can treat this as boilerplate, with very little variation. Here's a breakdown of the parameters and what values you supply:

- **`void *data`**—By passing `NULL` to the first argument, you ask Quartz to allocate memory on your behalf. Quartz then performs its own management on that memory, so you don't have to explicitly allocate or free it. You access the data by calling `CGBitmapContextGetData()`, which is what Listing 3-6 does in order to populate the `NSData` object it creates. As the `get` in the name suggests, this function reads the data but does not copy it or otherwise interfere with its memory management.
- **`size_t width` and `size_t height`**—The next two arguments are the image width and height. The `size_t` type is defined on iOS as `unsigned long`. Listing 3-6 passes `extents` it retrieves from the source image's size to `CGBitmapContextCreate()`.
- **`size_t bitsPerComponent`**—In UIKit, you work with 8-bit bytes (`uint_8`), so you just pass `8` unless you have a compelling reason to do otherwise. The Quartz 2D programming guide lists all supported pixel formats, which can include 5-, 16-, and 32-bit components. A *component* refers to a single channel of information. ARGB data uses four components per pixel. Grayscale data uses one (without alpha channel data) or two (with).
- **`size_t bytesPerRow`**—You multiply the size of the row by the bytes per component to calculate the number of bytes per row. Typically, you pass `width * 4` for ARGB images and just `width` for straight (non-alpha) grayscale. Take special note of this value. It's not just useful as a parameter; you also use it to calculate the (x, y) offset for any pixel in a byte array, namely `(y * bytesPerRow + x)`.
- **`CGColorSpaceRef colorspace`**—You pass the color space Quartz should use to create the bitmap context, typically device RGB or device gray.
- **`CGBitmapInfo bitmapInfo`**—This parameter specifies the style of alpha channel the bitmap uses. As a rule of thumb, use `kCGImageAlphaPremultipliedFirst` for color images and `kCGImageAlphaNone` for grayscale. If you're curious, refer to the Quartz 2D programming guide for a more complete list of options. In iOS 7 and later, make sure to cast any alpha settings to `(CGBitmapInfo)` to avoid compiler issues.

Listing 3-6 Extracting Bytes

```
NSData *BytesFromRGBImage(UIImage *sourceImage)
{
    if (!sourceImage) return nil;

    // Establish color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        NSLog(@"Error creating RGB color space");
        return nil;
    }

    // Establish context
    int width = sourceImage.size.width;
    int height = sourceImage.size.height;
    CGContextRef context = CGBitmapContextCreate(
        NULL, width, height,
        8, // bits per byte
        width * 4, // bytes per row
        colorSpace,
        (CGBitmapInfo) kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace);
    if (context == NULL)
    {
        NSLog(@"Error creating context");
        return nil;
    }

    // Draw source into context bytes
    CGRect rect = (CGRect){.size = sourceImage.size};
    CGContextDrawImage(context, rect, sourceImage.CGImage);

    // Create NSData from bytes
    NSData *data =
        [NSData dataWithBytes:CGBitmapContextGetData(context)
            length:(width * height * 4)]; // bytes per image
    CGContextRelease(context);
    return data;
}
```

Note

When responsiveness is of the essence, don't wait for `UIImage` instances and their underlying `CGImage` representations to decompress. Enable caching of decompressed images by loading your `CGImage` instances via `CGImageSource`. This implementation, which is part of the `ImageIO` framework, enables you to specify an option to cache decompressed data (`kCGImageSourceShouldCache`). This results in much faster drawing performance, albeit at the cost of extra storage. For more details, see www.cocoanetics.com/2011/10/avoiding-image-decompression-sickness.

Creating Images from Bytes

Listing 3-7 reverses the bytes-to-image scenario, producing images from the bytes you supply. Because of this, you pass those bytes to `CGBitmapContextCreate()` as the first argument. This tells Quartz not to allocate memory but to use the data you supply as the initial contents of the new context.

Beyond this one small change, the code in Listing 3-7 should look pretty familiar by now. It creates an image from the context, transforming the `CGImageRef` to a `UIImage`, and returns that new image instance.

Being able to move data in both directions—from image to data and from data to image—means that you can integrate image processing into your drawing routines and use the results in your `UIViews`.

Listing 3-7 Turning Bytes into Images

```
UIImage *ImageFromBytes(NSData *data, CGSize targetSize)
{
    // Check data
    int width = targetSize.width;
    int height = targetSize.height;
    if (data.length < (width * height * 4))
    {
        NSLog(@"Error: Got %d bytes. Expected %d bytes",
              data.length, width * height * 4);
        return nil;
    }

    // Create a color space
    CGColorSpaceRef colorSpace =
        CGColorSpaceCreateDeviceRGB();
```

```

if (colorSpace == NULL)
{
    NSLog(@"Error creating RGB colorspace");
    return nil;
}

// Create the bitmap context
Byte *bytes = (Byte *) data.bytes;
CGContextRef context = CGBitmapContextCreate(
    bytes, width, height,
    BITS_PER_COMPONENT, // 8 bits per component
    width * ARGB_COUNT, // 4 bytes in ARGB
    colorSpace,
    (CGBitmapInfo) kCGImageAlphaPremultipliedFirst);
CGColorSpaceRelease(colorSpace );
if (context == NULL)
{
    NSLog(@"Error. Could not create context");
    return nil;
}

// Convert to image
CGImageRef imageRef = CGBitmapContextCreateImage(context);
UIImage *image = [UIImage imageWithCGImage:imageRef];

// Clean up
CGContextRelease(context);
CFRelease(imageRef);

return image;
}

```

Drawing and Auto Layout

Under Auto Layout, the new constraint-based system in iOS and OS X, a view's content plays as important a role in its layout as its constraints. This is expressed through each view's *intrinsic content size*. This size describes the minimum space needed to express the full view content, without squeezing or clipping that data. It derives from the properties of the content that each view presents. In the case of images and drawings, it represents the "natural size" of an image in points.

When you include embellishments in your pictures such as shadows, sparks, and other items that extend beyond the image's core content, that natural size may no longer reflect the true way you want Auto Layout to handle layout. In Auto Layout, constraints determine

view size and placement, using a geometric element called an *alignment rectangle*. As you'll see, UIKit calls help control that placement.

Alignment Rectangles

As developers create complex views, they may introduce visual ornamentation such as shadows, exterior highlights, reflections, and engraving lines. As they do, these features are often drawn onto images. Unlike frames, a view's alignment rectangle should be limited to a core visual element. Its size should remain unaffected as new items are drawn onto the view. Consider Figure 3-6 (left). It depicts a view drawn with a shadow and a badge. When laying out this view, you want Auto Layout to focus on aligning just the core element—the blue rectangle—and not the ornamentation.

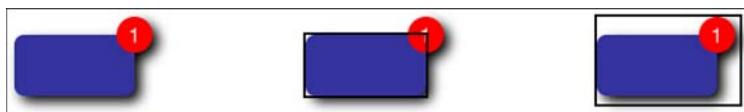


Figure 3-6 A view's alignment rectangle (center) refers strictly to the core visual element to be aligned, without embellishments.

The center image highlights the view's alignment rectangle. This rectangle excludes all ornamentation, such as the drop shadow and badge. It's the part of the view you want considered when Auto Layout does its work. Contrast this with the rectangle shown in the right image. This version includes all the visual ornamentation, extending the view's frame beyond the area that should be considered for alignment.

The right-hand rectangle encompasses all the view's visual elements. It encompasses the shadow and badge. These ornaments could potentially throw off a view's alignment features (for example, its center, bottom, and right) if they were considered during layout.

By working with alignment rectangles instead of frames, Auto Layout ensures that key information like a view's edges and center are properly considered during layout. In Figure 3-7, the adorned view is perfectly aligned on the background grid. Its badge and shadow are not considered during placement.

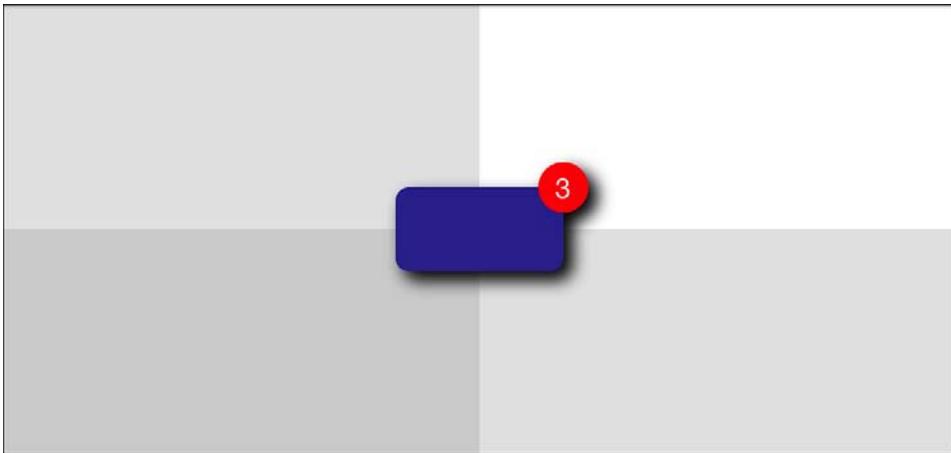


Figure 3-7 Auto Layout considers the view's alignment rectangle when laying it out as centered in its parent. The shadow and badge don't affect its placement.

Alignment Insets

Drawn art often contains hard-coded embellishments like highlights, shadows, and so forth. These items take up little memory and run efficiently. Therefore, many developers predraw effects because of their low overhead.

To accommodate extra visual elements, use `imageWithAlignmentRectInsets:`. You supply a `UIEdgeInsets` structure, and `UIImage` returns the inset-aware image. Insets define offsets from the top, left, bottom, and right of some rectangle. You use these to describe how far to move in (using positive values) or out (using negative values) from rectangle edges. These insets ensure that the alignment rectangle is correct, even when there are drawn embellishments placed within the image.

```
typedef struct {  
    CGFloat top, left, bottom, right;  
} UIEdgeInsets;
```

The following snippet accommodates a 20-point shadow by insetting the alignment rect on the bottom and right:

```
UIImage *image = [[UIImage imageNamed:@"Shadowed.png"]  
    imageWithAlignmentRectInsets:UIEdgeInsetsMake(0, 0, 20, 20)];  
UIImageView *imageView = [[UIImageView alloc] initWithImage:image];
```

It's a bit of a pain to construct these insets by hand, especially if you may later update your graphics. When you know the alignment rect and the overall image bounds, you can, instead, automatically calculate the edge insets you need to pass to this method. Listing 3-8 defines a simple inset builder. It determines how far the alignment rectangle lies from each

edge of the parent rectangle, and it returns a `UIEdgeInsets` structure representing those values. Use this function to build insets from the intrinsic geometry of your core visuals.

Listing 3-8 Building Edge Insets from Alignment Rectangles

```
UIEdgeInsets BuildInsets(
    CGRect alignmentRect, CGRect imageBounds)
{
    // Ensure alignment rect is fully within source
    CGRect targetRect =
        CGRectIntersection(alignmentRect, imageBounds);

    // Calculate insets
    UIEdgeInsets insets;
    insets.left = CGRectGetMinX(targetRect) -
        CGRectGetMinX(imageBounds);
    insets.right = CGRectGetMaxX(imageBounds) -
        CGRectGetMaxX(targetRect);
    insets.top = CGRectGetMinY(targetRect) -
        CGRectGetMinY(imageBounds);
    insets.bottom = CGRectGetMaxY(imageBounds) -
        CGRectGetMaxY(targetRect);

    return insets;
}
```

Drawing Images with Alignment Rects

Figure 3-8 demonstrates alignment rectangle layout in action. The image in the top view does not express alignment preferences. Therefore, the entire image (the translucent gray square, including the bunny, the shadow, and the star) is centered within the parent. The bottom screenshot uses alignment insets. These use just the bunny's bounding box (the inner outline) as reference. Here, the centering changes. It's now the bunny and not the overall image that's centered in the parent. The extra drawing area and other image details no longer contribute to that placement.

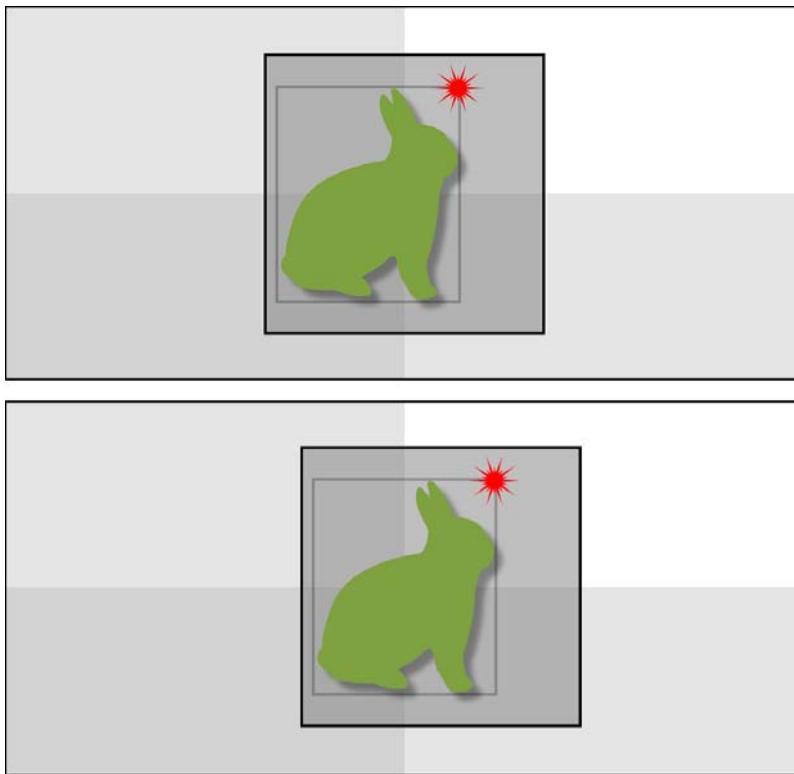


Figure 3-8 You can integrate alignment rectangle awareness into your drawing routines to ensure that Auto Layout aligns them properly. In the top image, the large square outlined in black is used for alignment. In the bottom, the inset gray rectangle around the bunny is the alignment rect.

Listing 3-9 details the drawing and alignment process behind the bottom screenshot. It creates an image with the gray background, the green bunny, the red badge, and the outline showing the bunny's bounds. As part of this process, it adds a shadow to the bunny and moves the badge to the bunny's top-right corner. Shadows and badges are fairly common items used in normal iOS visual elements, even in iOS 7's slimmed down, clean aesthetic.

In the end, however, the part that matters is specifying how to align the output image. To do that, this code retrieves the bounding box from the bunny's `UIBezierPath`. This path is independent of the badge, the background, and the drawn shadow. By applying the edge insets returned by Listing 3-8, Listing 3-9 creates an image that aligns around the bunny and only the bunny.

This is a really powerful way to draw ornamented graphics using Quartz 2D and UIKit that work seamlessly within Auto Layout.

Listing 3-9 Drawing with Alignment in Mind

```
UIBezierPath *path;

// Begin the image context
UIGraphicsBeginImageContextWithOptions(
    targetSize, NO, 0.0);
CGContextRef context = UIGraphicsGetCurrentContext();
CGRect targetRect = SizeMakeRect(targetSize);

// Fill the background of the image and outline it
[backgroundGrayColor setFill];
UIRectFill(targetRect);
path = [UIBezierPath bezierPathWithRect:targetRect];
[path strokeInside:2];

// Fit bunny into an inset, offset rectangle
CGRect destinationRect =
    RectInsetByPercent(SizeMakeRect(targetSize), 0.25);
destinationRect.origin.x = 0;
UIBezierPath *bunny =
    [[UIBezierPath bunnyPath]
        pathWithinRect:destinationRect];

// Add a shadow to the context and draw the bunny
CGContextSaveGState(context);
CGContextSetShadow(context, CGSizeMake(6,6), 4);
[greenColor setFill];
[bunny fill];
CGContextRestoreGState(context);

// Outline bunny's bounds, which are the alignment rect
CGRect alignmentRect = bunny.bounds;
path = [UIBezierPath bezierPathWithRect:alignmentRect];
[darkGrayColor setStroke];
[path strokeOutside:2];

// Add a red badge at the top-right corner
UIBezierPath *badge = [[UIBezierPath badgePath]
    pathWithinRect:CGRectMake(0, 0, 40, 40)];
badge = [badge pathMoveCenterToPoint:
    RectGetTopRight(bunny.bounds)];
[[UIColor redColor] setFill];
[badge fill];

// Retrieve the initial image
```

```
UIImage *initialImage =  
    UIGraphicsGetImageFromCurrentImageContext();  
UIGraphicsEndImageContext();  
  
// Build and apply the insets  
UIEdgeInsets insets =  
    BuildInsets(alignmentRect, targetRect);  
UIImage *image =  
    [initialImage imageWithAlignmentRectInsets:insets];  
  
// Return the updated image  
return image;
```

Building Stretchable Images

Resizable drawings enable you to create images whose edges are not scaled or resized when adjusted to fit a view. They preserve image details, ensuring that you resize only the middle of an image. Figure 3-9 shows an example of a stretched image. It acts as a button background, where its middle extent may grow or shrink, depending on the text assigned to the button. To ensure that only the middle is resized, a set of *cap insets* specify the off-limits edges. These caps ensure that the center region (the large purple expanse) can grow and shrink without affecting the rounded corners, with their lined artwork.



Figure 3-9 You can create stretchable images for use with buttons.

To get a sense of how cap insets work, compare Figure 3-9 with Figure 3-10. Figure 3-10 shows the same image assigned to the same button, but without the cap insets. Its edges and corners stretch along with the rest of the button image, producing a visually confusing result. The clean presentation in Figure 3-9 becomes a graphic mess in Figure 3-10.



Figure 3-10 Without caps, the button proportionately stretches all parts of the image.

Listing 3-10 shows the code behind the button image. It builds a 40-by-40 image context, draws two rounded rectangles into that context, and fills the background with a solid color. It retrieves this base image, but before returning it, Listing 3-10 calls `resizableImageWithCapInsets:`. This method creates a new version of the image that uses those cap insets. This one line makes the difference between the button you see in Figure 3-9 and the one you see in Figure 3-10.

Although iOS 7 introduced borderless, trimmed-away buttons, the traditional button design demonstrated in this listing continues to play a role in third party applications. This was emphasized in Apple's own WWDC sessions, where demos included many skeumorphic examples. You read more about button effects, including glosses, textures, and other un-Ive enhancements, in Chapter 7. The age of custom buttons is not dead; they just need more thought than they used to.

Note

Performance introduces another compelling reason for using stretchable images where possible. They are stretched on the GPU, providing a boost over manual stretching. The same is true for pattern colors, which you'll read about later in this chapter. Filling with pattern colors executes faster than manually drawing that pattern into a context.

Listing 3-10 Images and Stretching

```
CGSize targetSize = CGSizeMake(40, 40);
UIGraphicsBeginImageContextWithOptions(
    targetSize, NO, 0.0);

// Create the outer rounded rectangle
CGRect targetRect = CGRectMake(0, 0, targetSize.width, targetSize.height);
UIBezierPath *path =
    [UIBezierPath bezierPathWithRoundedRect:targetRect
        cornerRadius:12];

// Fill and stroke it
[bgColor setFill];
```

```
[path fill];
[path strokeInside:2];

// Create the inner rounded rectangle
UIBezierPath *innerPath =
    [UIBezierPath bezierPathWithRoundedRect:
        CGRectMakeInset(targetRect, 4, 4) cornerRadius:8];

// Stroke it
[innerPath strokeInside:1];

// Retrieve the initial image
UIImage *baseImage =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

// Create a resizable version, with respect to
// the primary corner radius
UIImage *image =
    [baseImage resizableImageWithCapInsets:
        UIEdgeInsetsMake(12, 12, 12, 12)];
return image;
```

Rendering a PDF

Figure 3-11 shows an image built by rendering a PDF page into an image context. This task is made a little complicated by the Core Graphics API, which does not offer a “draw page in rectangle” option.

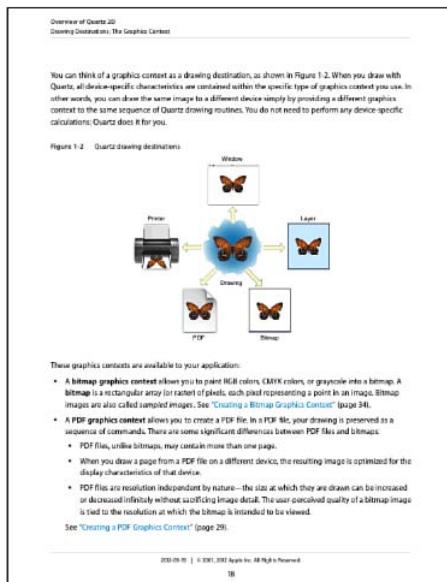


Figure 3-11 Rendering PDF pages into contexts requires a few tweaks.

Several stages are involved in the process, as detailed in the next listings. You start by opening a PDF document, as demonstrated in Listing 3-11. The `CGPDFDocumentCreateWithURL()` function returns a new document reference, which you can use to extract and draw pages.

When you have that document, follow these steps:

1. Check the number of pages in the document by calling `CGPDFDocumentGetNumberOfPages()`.
2. Retrieve each page by using `CGPDFDocumentGetPage()`. The page count starts with page 1, not page 0, as you might expect.
3. Make sure you release the document with `CGPDFDocumentRelease()` after you finish your work.

Listing 3-11 Opening a PDF Document

```
// Open PDF document
NSString *pdfPath = [[NSBundle mainBundle]
    pathForResource:@"drawingwithquartz2d" ofType:@"pdf"];
CGPDFDocumentRef pdfRef = CGPDFDocumentCreateWithURL(
    (_bridge CFURLRef)[NSURL fileURLWithPath:pdfPath]);
if (pdfRef == NULL)
{
```

```
    NSLog(@"Error loading PDF");
    return nil;
}

// ... use PDF document here

CGPDFDocumentRelease(pdfRef);
```

Upon grabbing `CGPDFPageRef` pages, Listing 3-12 enables you to draw each one into an image context, using a rectangle you specify. What's challenging is that the PDF functions draw using the Quartz coordinate system (with the origin at the bottom left), and the destinations you'll want to draw to are in the UIKit coordinate system (with the origin at the top left).

To handle this, you have to play a little hokey-pokey with your destination and with the context. First, you flip the entire context vertically, to ensure that the PDF page draws from the top down. Next, you transform your destination rectangle, so the page draws at the right place.

You might wonder why you need to perform this double transformation, and the answer is this: After you flip your coordinate system to enable top-to-bottom Quartz drawing, your destination rectangle that used to be, for example, at the top right, will now draw at the bottom right. That's because it's still living in a UIKit world. After you adjust the drawing context's transform, your rectangle must adapt to that transform, as it does about halfway down Listing 3-12. If you skip this step, your PDF output appears at the bottom right instead of the top right, as you intended.

I encourage you to try this out on your own by commenting out the rectangle transform step and testing various destination locations. What you'll discover is an important lesson in coordinate system conformance. Flipping the context doesn't just "fix" the Quartz drawing; it affects all position definitions. You'll see this same problem pop up in Chapter 7, when you draw text into UIKit paths. In that case, you won't be working with just rectangles. You must vertically mirror the entire path in the drawing destination.

When you've performed coordinate system adjustments, you calculate a proper-fitting rectangle for the page. As Chapter 2 discusses, fitting rectangles retain aspect while centering context within a destination. This requires one last context adjustment, so the drawing starts at the top left of that fitting rectangle. Finally, you draw. The `CGContextDrawPDFPage()` function renders page contents into the active context.

The `DrawPDFPageInRect()` function is meant only for drawing to UIKit images destinations. It cannot be used when drawing PDF pages into PDF contexts. It depends on retrieving a `UIImage` from the context in order to perform its geometric adjustments. To adapt this listing for more general use, you need to supply both a context parameter (rather than retrieve one from UIKit) and a context size, for the vertical transformation.

Listing 3-12 Drawing PDF Pages into Destination Rectangles

```
void DrawPDFPageInRect(CGPDFPageRef pageRef,
    CGRect destinationRect)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    CGContextSaveGState(context);
    UIImage *image =
        UIGraphicsGetImageFromCurrentImageContext();

    // Flip the context to Quartz space
    CGAffineTransform transform = CGAffineTransformIdentity;
    transform = CGAffineTransformScale(transform, 1.0f, -1.0f);
    transform = CGAffineTransformTranslate(
        transform, 0.0f, -image.size.height);
    CGContextConcatCTM(context, transform);

    // Flip the rect, which remains in UIKit space
    CGRect d = CGRectApplyAffineTransform(
        destinationRect, transform);

    // Calculate a rectangle to draw to
    // CGPDFPageGetBoxRect() returns a rectangle
    // representing the page's dimension
    CGRect pageRect =
        CGPDFPageGetBoxRect(pageRef, kCGPDFCropBox);
    CGFloat drawingAspect = AspectScaleFit(pageRect.size, d);
    CGRect drawingRect = RectByFittingInRect(pageRect, d);

    // Draw the page outline (optional)
    UIRectFrame(drawingRect);

    // Adjust the context to the page draws within
    // the fitting rectangle (drawingRect)
    CGContextTranslateCTM(context,
        drawingRect.origin.x, drawingRect.origin.y);
    CGContextScaleCTM(context, drawingAspect, drawingAspect);

    // Draw the page
    CGContextDrawPDFPage(context, pageRef);
```

```
CGContextRestoreGState(context);  
}
```

Bitmap Context Geometry

When working with bitmap contexts, you can retrieve certain details, like the context’s height and width, directly from the context reference. The `CGBitmapContextGetHeight()` and `CGBitmapContextGetWidth()` functions report integer dimensions. You cannot, however, retrieve a context’s scale—the way the number of pixels in the context relates to the number of points for the output image. That’s because scale lives a bit too high up in abstraction. As you’ve seen throughout this book, scale is set by UIKit’s `UIGraphicsBeginImageContextWithOptions()` and is not generally a feature associated with direct Quartz drawing.

For that reason, Listing 3-12 doesn’t use the bitmap context functions to retrieve size. This is important because transforms operate in points, not pixels. If you applied a Retina-scaled flip to your context, you’d be off, mathematically, by a factor of two. Your 200-point translation would use a 400-pixel value returned by `CGBitmapContextGetHeight()`.

Point-versus-pixel calculations aside, the context functions are not without use. They can retrieve the current context transformation matrix (`CGContextGetCTM()`), bytes per row (`CGBitmapContextGetBytesPerRow()`), and alpha level (`CGBitmapContextGetAlphaInfo()`), among other options. Search for “ContextGet” in the Xcode Documentation Organizer for more context-specific functions.

Building a Pattern Image

Pattern images are a lovely little UIKit gem. You craft a pattern in code or load an image from a file and assign it as a “color” to any view, as in this snippet:

```
self.view.backgroundColor =  
    [UIColor colorWithPatternImage:[self buildPattern]];
```

Figure 3-12 shows a simple pattern created by Listing 3-13. This listing uses common techniques: rotating and mirroring shapes and alternating major and minor sizes.

If you’d rather create your patterns externally and import them into apps, check out the wonderful design tool Patterno (\$19.99 at the Mac App Store). It enables you to build repeating textures that naturally tile.

Scalable Vector Graphics (SVG) patterns are widely available. SVG uses an XML standard to define vector-based graphics for the Web, providing another avenue for pattern design. Sites like <http://philbit.com/svgpatterns> offer simple but visually pleasing possibilities. PaintCode (\$99.99 at the Mac App Store) converts SVG snippets to standard UIKit drawing, providing an easy pathway from SVG sources to UIKit implementation. Just paste the SVG directives into TextEdit, rename to .svg, and import the result into PaintCode. PaintCode translates the SVG to Objective-C, which you can add to your Xcode projects.

Note

Unfamiliar with Clarus the DogCow? Google for Apple Technote 31 or visit <http://en.wikipedia.org/wiki/Dogcow>. *Moof!*

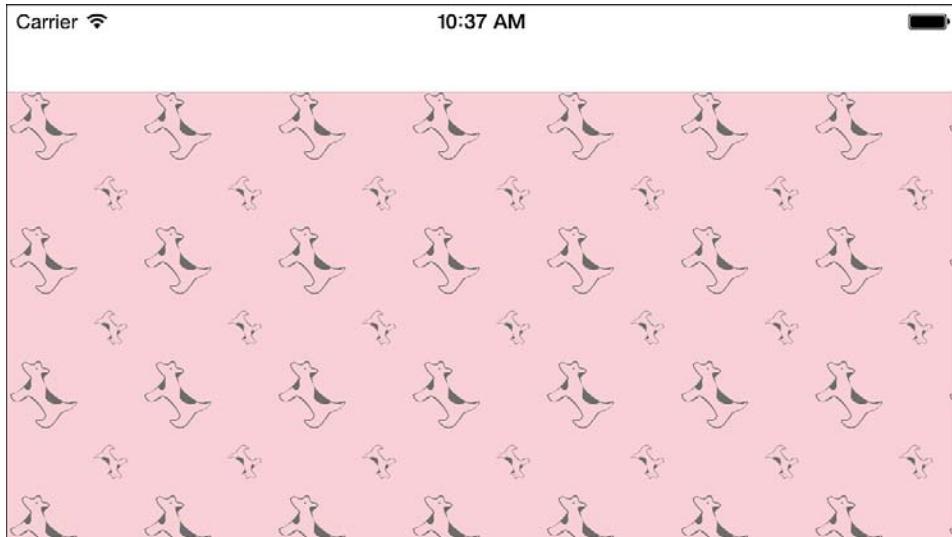


Figure 3-12 You can create `UIColor` instances built from pattern images.

Listing 3-13 Building Clarus the DogCow

```
- (UIImage *) buildPattern
{
    // Create a small tile
    CGSize targetSize = CGSizeMake(80, 80);
    CGRect targetRect = CGRectMake(targetSize);

    // Start a new image
    UIGraphicsBeginImageContextWithOptions(
        targetSize, NO, 0.0);
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Fill background with pink
    [customPinkColor set];
```

```

UIRectFill(targetRect);

// Draw a couple of dogcattle in gray
[[UIColor grayColor] set];

// First, bigger with interior detail in the top-left.
// Read more about Bezier path objects in Chapters 4 and 5
CGRect weeRect = CGRectMake(0, 0, 40, 40);
UIBezierPath *moof = BuildMoofPath();
FitPathToRect(moof, weeRect);
RotatePath(moof, M_PI_4);
[moof fill];

// Then smaller, flipped around, and offset down and right
RotatePath(moof, M_PI);
OffsetPath(moof, CGSizeMake(40, 40));
ScalePath(moof, 0.5, 0.5);
[moof fill];

// Retrieve and return the pattern image
UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
return image;
}

```

Summary

This chapter surveys several common challenges you encounter when drawing images to iOS contexts. You read about basic image generation, converting between bytes and images, adding inset adjustments, and performing PDF drawing. Before you move on, here are a few final thoughts about this chapter:

- Alignment insets are your friends. They enable you to build complex visual presentations while simplifying their integration with Auto Layout. They will save you a ton of detail work if you keep the core visuals in mind while doing your drawing. From there, all you have to do is add those images into standard image views and let Auto Layout do the rest of the work.
- When you pull the bytes out from a drawing or photograph, it's really easy to work with them directly or to move them to a digital signal processing (DSP) library to create visual effects. The routines you saw in this chapter (image to bytes and bytes to image) provide an important bridge between DSP processing and UIKit image presentation.

- When moving between UIKit drawing and Quartz drawing, don't overlook the difference in coordinate systems. The PDF example in this chapter avoids not one but two separate gotchas. To use UIKit-sourced rectangles in your Quartz-flipped drawing system, you must transform those rectangles to match the reversed coordinates.

This page intentionally left blank

4

Path Basics

Bezier paths are some of the most important tools for iOS drawing, enabling you to create and draw shapes, establish clipping paths, define animation paths, and more. Whether you're building custom view elements, adding Photoshop-like special effects, or performing ordinary tasks like drawing lines and circles, a firm grounding in the `UIBezierPath` class will make your development easier and more powerful.

Why Bezier

When building buttons or bevels, shadows or patterns, Bezier paths provide a powerful and flexible solution. Consider Figure 4-1. Both screenshots consist of elements sourced primarily from Bezier path drawing.

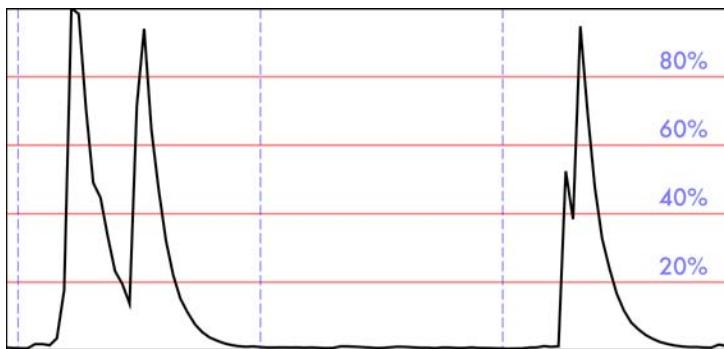


Figure 4-1 Despite the striking visual differences between these two screenshots, the `UIBezierPath` class is primarily responsible for the content in both of them.

The top screenshot is from an audio sampling app. Its blue and white graph and its black audio scan are simple path instances drawn as solid and dashed lines. These lines demonstrate a conventional use case, where each path traces some pattern.

The bottom screenshot uses more sophisticated effects. The bunny's geometry is stored in a single Bezier path. So is the white rounded rectangle frame that surrounds it. The gradient overlay on top of this button is clipped to a Bezier-based ellipse. Bezier paths also established the clipping paths that shape the outer edges and that define the limits for the subtle outer glow effect as well as the inner 3D beveling.

Although the results are strikingly different between the two screenshots, the class behind the drawing is the same. If you plan to incorporate UIKit drawing into your apps, you'll want to develop a close and comfortable relationship with the `UIBezierPath` class.

Class Convenience Methods

UIBezierPath class methods build rectangles, ovals, rounded rectangles, and arcs, offering single-call access to common path style elements:

- **Rectangles**—`bezierPathWithRect`: is used for any kind of view filling or rectangular elements you have in your interface. You can create any kind of rectangle with this method.
- **Ovals and circles**—`bezierPathWithOvalInRect`: provides a tool for building circles and ellipses of any kind.
- **Rounded rectangles**—`bezierPathWithRoundedRect:cornerRadius`: builds the rounded rectangles that iOS designers love so much. These paths are perfect for buttons, popup alerts, and many other view elements.
- **Corner-controlled rounded rectangles**—`bezierPathWithRoundedRect:byRoundingCorners:cornerRadii`: enables you to create rounded rectangles but choose which corners to round. This is especially helpful for creating alerts and panels that slide in from just one side of the screen, where you don't want to round all four corners.
- **Arcs**—`bezierPathWithArcCenter:radius:startAngle:endAngle:clockwise`: draws arcs from a starting and ending angle that you specify. Use these arcs to add visual embellishments, to create stylized visual dividers, and to soften your overall GUI.

These Bezier path class methods provide a basic starting point for your drawing, as demonstrated in Example 4-1. This code builds a new Bezier path by combining several shapes. At each step, it appends the shape to an ever-growing path. When stroked, this resulting path produces the “face” shown in Figure 4-2.

As you see in Example 4-1, Bezier path instances are inherently mutable. You can add to them and grow them by appending new shapes and elements. What's more, paths need not be contiguous. The path in Figure 4-2 consists of four distinct, nonintersecting subshapes, which can be painted as a single unit.

Note

This is the first part of the book that uses examples. As the name suggests, examples show samples of applying API calls as you draw. Listings, in contrast, offer utilities and useful snippets that you may bring back to your own projects.

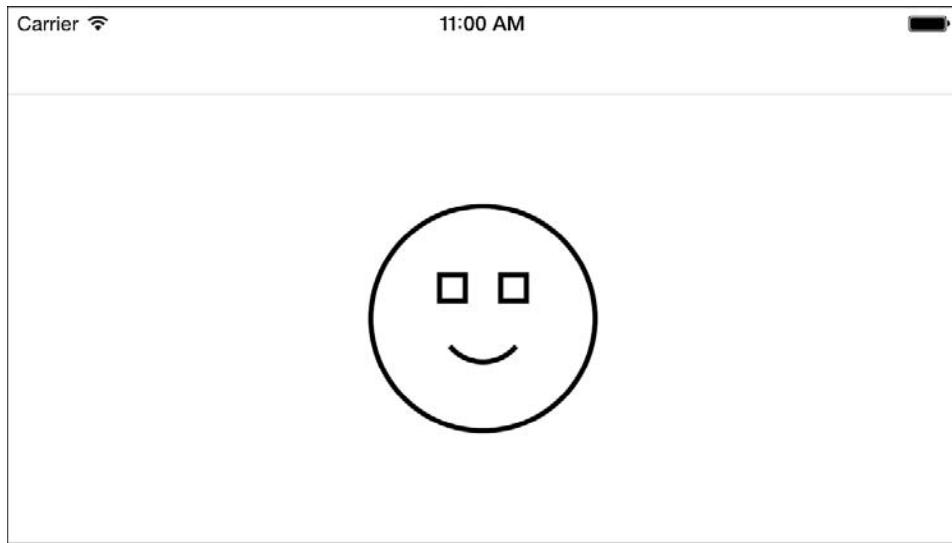


Figure 4-2 This face consists of a single Bezier path. It was constructed by appending class-supplied shapes, namely an oval (aka a circle), an arc, and two rectangles, to a path instance.

Although you will run into performance issues if your paths get *too* complex, this usually happens only with user-driven interaction. I represent drawings in my Draw app as paths. Kindergartners, with nothing better to do than draw continuously for 5 or 10 minutes at a time without ever lifting their finger, will cause slow-downs as paths become insanely long. You shouldn't encounter this issue for any kind of (sane) path you prebuild using normal vector drawing and incorporate into your own applications.

You will want to break down paths, however, for logical units. Paths are drawn at once. A stroke width and fill/stroke colors are applied to all elements. When you need to add multiple drawing styles, you should create individual paths, each of which represents a single unified operation.

Example 4-1 Building a Bezier Path

```
CGRect fullRect = (CGRect){.size = size};  
  
// Establish a new path  
UIBezierPath *bezierPath = [UIBezierPath bezierPath];  
  
// Create the face outline  
// and append it to the path  
CGRect inset = CGRectInset(fullRect, 32, 32);  
UIBezierPath *faceOutline =  
[UIBezierPath bezierPathWithOvalInRect:inset];
```

```
[bezierPath appendPath:faceOutline];

// Move in again, for the eyes and mouth
CGRect insetAgain = CGRectMakeInset(inset, 64, 64);

// Calculate a radius
CGPoint referencePoint =
    CGPointMake(CGRectGetMinX(insetAgain),
               CGRectGetMaxY(insetAgain));
CGPoint center = RectGetCenter(inset);
CGFloat radius = PointDistanceFromPoint(referencePoint, center);

// Add a smile from 40 degrees around to 140 degrees
UIBezierPath *smile =
    [UIBezierPath bezierPathWithArcCenter:center
        radius:radius startAngle:RadiansFromDegrees(140)
        endAngle:RadiansFromDegrees(40) clockwise:NO];
[bezierPath appendPath:smile];

// Build Eye 1
CGPoint p1 = CGPointMake(CGRectGetMinX(insetAgain),
    CGRectGetMinY(insetAgain));
CGRect eyeRect1 = RectAroundCenter(p1, CGSizeMake(20, 20));
UIBezierPath *eye1 =
    [UIBezierPath bezierPathWithRect:eyeRect1];
[bezierPath appendPath:eye1];

// And Eye 2
CGPoint p2 = CGPointMake(CGRectGetMaxX(insetAgain),
    CGRectGetMinY(insetAgain));
CGRect eyeRect2 = RectAroundCenter(p2, CGSizeMake(20, 20));
UIBezierPath *eye2 =
    [UIBezierPath bezierPathWithRect:eyeRect2];
[bezierPath appendPath:eye2];

// Draw the complete path
bezierPath.lineWidth = 4;
[bezierPath stroke];
```

Nesting Rounded Rectangles

Figure 4-3 shows two nested rounded rectangles. Building rounded rectangles is a common drawing task in iOS apps. Each path is filled and stroked: the outer rectangle with a 2-point line and the inner rectangle with a 1-point line. Figure 4-3 demonstrates a common error.

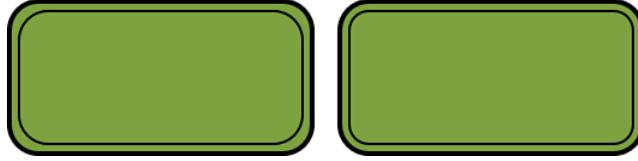


Figure 4-3 Nested rectangles with rounded corners.

Look carefully at the corners of each image. The inner path's corner radius differs between the first image and second image. The left image insets its inner path by 4 points and draws it using the same 12-point corner radius as the first path:

```
UIBezierPath *path2 = [UIBezierPath bezierPathWithRoundedRect:  
    CGRectMakeInset(destinationRect, 4, 4) cornerRadius:12];
```

The second performs the same 4-point inset but reduces the corner radius from 12 points to 8. Matching the inset to the corner radius creates a better fit between the two paths, ensuring that both curves start and end at parallel positions. It produces the more visually pleasing result you see in the right-hand image.

Building Paths

When system-supplied paths like rectangles and ovals are insufficient to your needs, you can iteratively build paths. You create paths by laying out items point-by-point, adding curves and lines as you go.

Each Bezier path can include a variety of geometric elements, including the following:

- Positioning requests established by `moveToPoint:`
- Straight lines, added by `addLineToPoint:`
- Cubic Bezier curve segments created by
`addCurveToPoint:controlPoint1:controlPoint2:`
- Quadratic Bezier curve segments built by
`addQuadCurveToPoint:controlPoint:`
- Arcs added with calls to `addArcToCenter:radius:startAngle:`
`endAngle:clockwise:`.

Figure 4-4 shows a star built from a series of cubic segments. Example 4-2 details the code behind the drawing. It establishes a new path, tells it to move to the starting point (`p0`), and then adds a series of cubic curves to build the star shape.

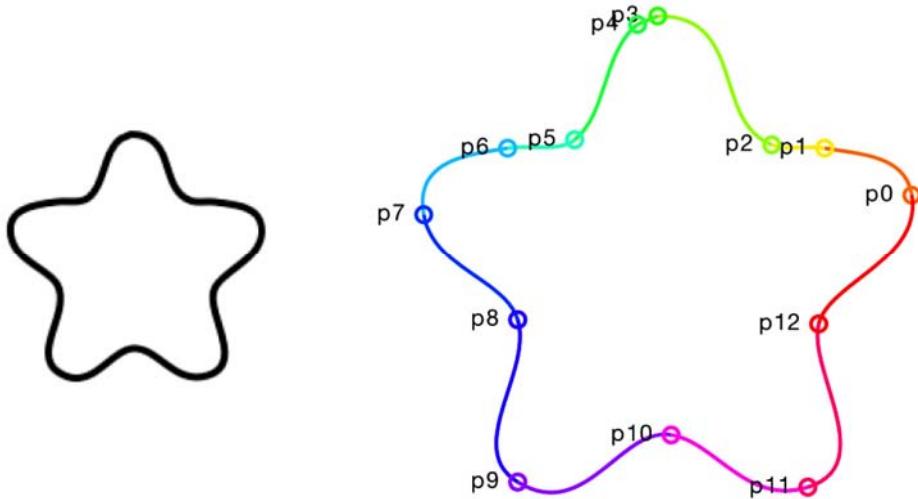


Figure 4-4 This shape started off in Photoshop. Pixel Cut's PaintCode transformed it into the code in Example 4-2.

If this code seems particularly opaque, well, it is. Constructing a Bezier curve is not a code-friendly process. In fact, this particular path began its life in Photoshop. I drew a shape and saved it to a PSD file. Then I used PaintCode (available from the Mac App Store for \$99 + \$20 in-app Photoshop file import purchase) to transform the vector art into a series of Objective-C calls.

When it comes to drawing, the tools best suited to expressing shapes often lie outside Xcode. After designing those shapes, however, there's a lot you can do from code to manage and develop that material for your app. For that reason, you needn't worry about the exact offsets, positions, and sizes you establish in Photoshop or other tools. As you'll discover, `UIBezierPath` instances express vector art. They can be scaled, translated, rotated, and more, all at your command, so you can tweak that material exactly as needed in your apps.

Example 4-2 Creating a Star Path

```
// Create new path, courtesy of PaintCode (paintcodeapp.com)
UIBezierPath* bezierPath = [UIBezierPath bezierPath];

// Move to the start of the path, at the right side
[bezierPath moveToPoint: CGPointMake(883.23, 430.54)]; // p0

// Add the cubic segments
[bezierPath addCurveToPoint: CGPointMake(749.25, 358.4) // p1
controlPoint1: CGPointMake(873.68, 370.91)]
```

```

controlPoint2: CGPointMake(809.43, 367.95)];
[bezierPath addCurveToPoint: CGPointMake(668.1, 353.25) // p2
controlPoint1: CGPointMake(721.92, 354.07)
controlPoint2: CGPointMake(690.4, 362.15)];
[bezierPath addCurveToPoint: CGPointMake(492.9, 156.15) // p3
controlPoint1: CGPointMake(575.39, 316.25)
controlPoint2: CGPointMake(629.21, 155.47)];
[bezierPath addCurveToPoint: CGPointMake(461.98, 169.03) // p4
controlPoint1: CGPointMake(482.59, 160.45)
controlPoint2: CGPointMake(472.29, 164.74)];
[bezierPath addCurveToPoint: CGPointMake(365.36, 345.52) // p5
controlPoint1: CGPointMake(409.88, 207.98)
controlPoint2: CGPointMake(415.22, 305.32)];
[bezierPath addCurveToPoint: CGPointMake(262.31, 358.4) // p6
controlPoint1: CGPointMake(341.9, 364.44)
controlPoint2: CGPointMake(300.41, 352.37)];
[bezierPath addCurveToPoint: CGPointMake(133.48, 460.17) // p7
controlPoint1: CGPointMake(200.89, 368.12)
controlPoint2: CGPointMake(118.62, 376.61)];
[bezierPath addCurveToPoint: CGPointMake(277.77, 622.49) // p8
controlPoint1: CGPointMake(148.46, 544.36)
controlPoint2: CGPointMake(258.55, 560.05)];
[bezierPath addCurveToPoint: CGPointMake(277.77, 871.12) // p9
controlPoint1: CGPointMake(301.89, 700.9)
controlPoint2: CGPointMake(193.24, 819.76)];
[bezierPath addCurveToPoint: CGPointMake(513.51, 798.97) // p10
controlPoint1: CGPointMake(382.76, 934.9)
controlPoint2: CGPointMake(435.24, 786.06)];
[bezierPath addCurveToPoint: CGPointMake(723.49, 878.84) // p11
controlPoint1: CGPointMake(582.42, 810.35)
controlPoint2: CGPointMake(628.93, 907.89)];
[bezierPath addCurveToPoint: CGPointMake(740.24, 628.93) // p12
controlPoint1: CGPointMake(834.7, 844.69)
controlPoint2: CGPointMake(722.44, 699.2)];
[bezierPath addCurveToPoint: CGPointMake(883.23, 430.54) // p0
controlPoint1: CGPointMake(756.58, 564.39)
controlPoint2: CGPointMake(899.19, 530.23)];

```

Drawing Bezier Paths

After you create Bezier path instances, you draw them to a context by applying `fill` or `stroke`. Filling paints a path, adding color to all the areas inside the path. Stroking outlines a path, painting just the edges, using the line width stored in the path's `lineWidth` property. A typical drawing pattern might go like this:

```
myPath.lineWidth = 4.0f;
[[UIColor blackColor] setStroke];
[[UIColor redColor] setFill];
[myPath fill];
[myPath stroke];
```

This sequence assigns a line width to the path, sets the fill and stroke colors for the current context, and then fills and strokes the path.

Listing 4-1 introduces what I consider to be a more convenient approach. The methods define a class category that adds two new ways to draw. This category condenses the drawing sequence as follows:

```
[myPath fill:[UIColor redColor]];
[myPath stroke:4 color:[UIColor blackColor]];
```

The colors and stroke width become parameters of a single drawing request. These functions enable you to create a series of drawing operations without having to update context or instance properties between requests.

All drawing occurs within the graphics state (GState) save and restore requests introduced in Chapter 1. This ensures that the passed parameters don't persist past the method call, so you can return to exactly the same context and path state you left.

Note

When working with production code (that is, when not writing a book and trying to offer easy-to-read examples), please make sure to namespace your categories. Add custom prefixes that guarantee your extensions won't overlap with any possible future Apple class updates.

`esStroke:color:` isn't as pretty as `stroke:color:`, but it offers long-term protection for your code.

Listing 4-1 Drawing Utilities

```
@implementation UIBezierPath (HandyUtilities)
// Draw with width
- (void) stroke: (CGFloat) width color: (UIColor *) color
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    CGContextSaveGState(context);
```

```

// Set the color
if (color) [color setStroke];

// Store the width
CGFloat holdWidth = self.lineWidth;
self.lineWidth = width;

// Draw
[self stroke];

// Restore the width
self.lineWidth = holdWidth;
CGContextRestoreGState(context);
}

// Fill with supplied color
- (void) fill: (UIColor *) fillColor
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }
    CGContextSaveGState(context);
    [fillColor set];
    [self fill];
    CGContextRestoreGState(context);
}

```

Drawing Inside Paths

In UIKit, the `UIRectFrame()` function draws a single line inside a rectangle you supply as a parameter. This function provides particularly clean and appealing results. It was my inspiration for Listing 4-2.

Listing 4-2 performs the same stroke operation that you saw in Figure 4-3 but with two slight changes. First, it doubles the size of the stroke. Second, it clips the drawing using `addClip`.

Normally, a stroke operation draws the stroke in the center of the path's edge. This creates a stroke that's half on one side of that edge and half on the other. Doubling the size ensures that the inside half of the stroke uses exactly the size you specified.

As you discovered in Chapter 1, clipping creates a mask. It excludes material from being added to the context outside the clipped bounds. In Listing 4-2, clipping prevents the stroke from continuing past the path’s edge, so all drawing occurs inside the path. The result is a stroke of a fixed size that’s drawn fully within the path.

Listing 4-2 Stroking Inside a Path

```
- (void) strokeInside: (CGFloat) width color: (UIColor *) color
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    CGContextSaveGState(context);
    [self addClip];
    [self stroke:width * 2 color:color]; // Listing 4-1
    CGContextRestoreGState(context);
}
```

Filling Paths and the Even/Odd Fill Rule

In vector graphics, *winding rules* establish whether areas are inside or outside a path. These rules affect whether areas, like those shown in Figure 4-5, are colored or not when you fill a path. Quartz uses an even/odd rule, which is an algorithm that determines the degree to which any area is “inside” a path.

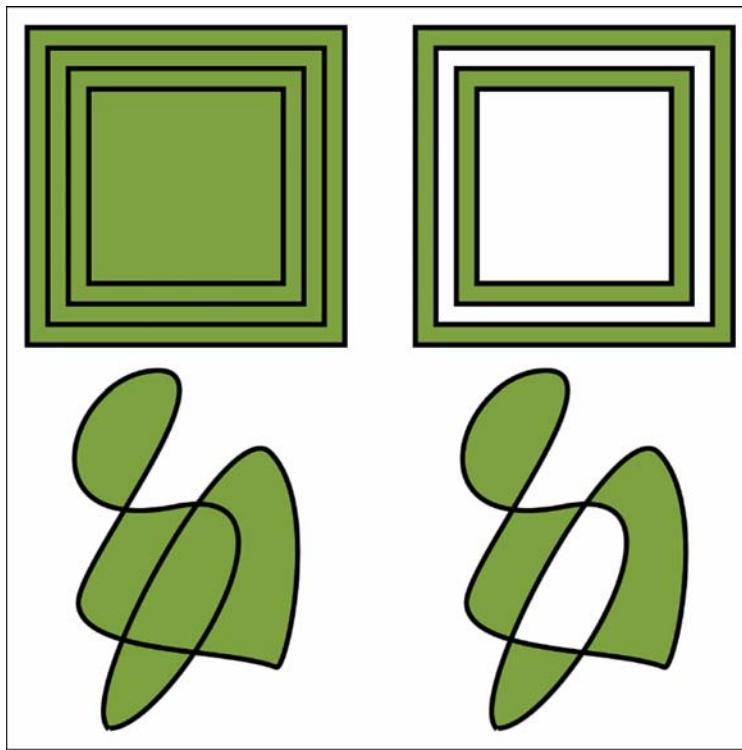


Figure 4-5 An even/odd fill rule establishes whether Quartz fills the entire inner path or only “inside” areas. The shapes on the left side use the default fill rule. The shapes on the right apply the even/odd fill rule.

You use this rule in many drawing scenarios. For example, the even/odd fill rule enables you to draw complex borders. It offers a way to cut areas out of your text layouts so you can insert images. It provides a basis for inverting selections, enabling you to create positive and negative space and more.

The algorithm tests containment by projecting a ray (a line with one fixed end, pointing in a given direction) from points within the path to a distant point outside it. The algorithm counts the number of times that ray crosses any line. If the ray passes through an even number of intersections, the point is outside the shape; if odd, inside.

In the nested square example at the left of Figure 4-5, a point in the very center passes through four subsequent lines before passing out of the path. A point lying just past that box, between the third and fourth inner squares, would pass through only three lines on the way out. Therefore, according to the even/odd fill rule, the first point is “outside” the shape because it passes through an even number of lines. The second point is “inside” the shape because it passes through an odd number.

These even and odd numbers have consequence, which you can see in Figure 4-5. When you enable a path's `usesEvenOddFillRule` property, UIKit uses this calculation to determine which areas lay inside the shape and should be filled and what areas are outside and should not.

A special version of the context fill function in Quartz, `CGContextEOFillPath()`, performs a context fill by applying the even/odd fill rule. When you use the normal version of the function, `CGContextFillPath()`, the rule is not applied.

Retrieving Path Bounds and Centers

The `UIBezierPath` class includes a `bounds` property. It returns a rectangle that fully encloses all points in the path, including control points. Think of this property as a path's frame. The origin almost never is at zero; instead, it represents the top-most and left-most point, or the control point used to construct the path.

Bounds are useful because they're fast to calculate and provide a good enough approximation for many drawing tasks. Figure 4-6 demonstrates an inherent drawback. When you're working with an exacting layout, bounds do not take the true extent of a figure's curves into account. For that reason, bounds almost always exceed actual boundary extents.

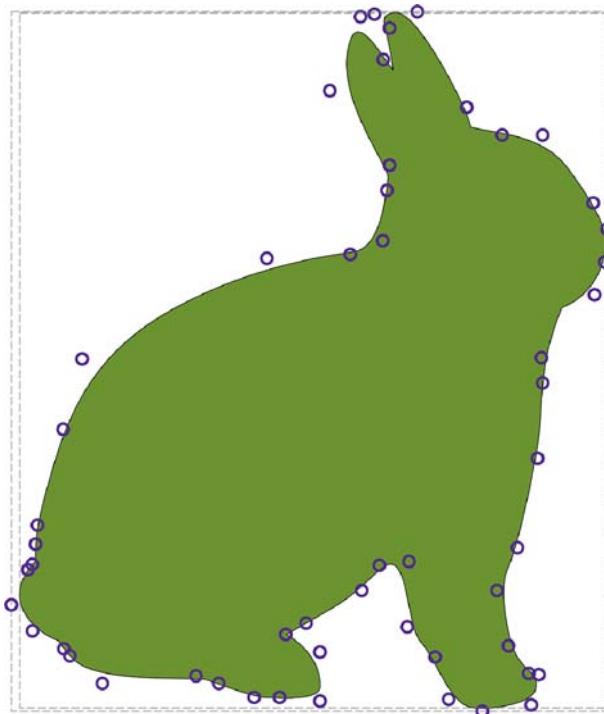


Figure 4-6 The green rendering shows the path. Purple circles are the control points used to construct that path. The dashed rectangles indicate the path’s bounds property (outer rectangle) and the true boundary (inner rectangle). Although close in size, they are not identical, and that difference can be important when you’re creating perfect drawings.

Listing 4-3 turns to the Quartz `CGPathGetPathBoundingBox()` function to retrieve better path bounds. This approach demands more calculation as the function must interpolate between control points to establish the points along each curve. The results are far superior. You trade accuracy for the overhead of time and processor demands.

The `PathBoundingBox()` function returns the closer bounds shown in Figure 4-6’s inner rectangle. A related function, `PathBoundingBoxWithLineWidth()`, extends those bounds to take a path’s `lineWidth` property into account. Strokes are drawn over a path’s edges, so the path generally expands by half that width in each direction.

Listing 4-3 also includes functions to calculate a path’s center, using both the general (`PathCenter()`) and precise (`PathBoundingCenter()`) results. The latter version offers a better solution for transformations, where precision matters. (Slight errors can be magnified by affine transforms.) Using the path bounding box versions of these measurements ensures a more consistent end result.

Note

In my own development, I've implemented the routines in Listing 4-3 as a `UIBezierPath` category.

Listing 4-3 A Path's Calculated Bounds

```
// Return calculated bounds
CGRect PathBoundingBox(UIBezierPath *path)
{
    return CGPathGetPathBoundingBox(path.CGPath);
}

// Return calculated bounds taking line width into account
CGRect PathBoundingBoxWithLineWidth(UIBezierPath *path)
{
    CGRect bounds = PathBoundingBox(path);
    return CGRectInset(bounds,
        -path.lineWidth / 2.0f, -path.lineWidth / 2.0f);
}

// Return the calculated center point
CGPoint PathBoundingCenter(UIBezierPath *path)
{
    return RectGetCenter(PathBoundingBox(path));
}

// Return the center point for the bounds property
CGPoint PathCenter(UIBezierPath *path)
{
    return RectGetCenter(path.bounds);
}
```

Transforming Paths

The Bezier path's `applyTransform:` method transforms all of a path's points and control points, by applying the affine transform matrix passed as the method argument. This change happens in place and will update the calling path. For example, after you apply the following scale transform, the entire `myPath` path shrinks:

```
[myPath applyTransform:CGAffineTransformMakeScale(0.5f, 0.5f)];
```

If you'd rather preserve the original path, create a copy and apply the transform to the copy. This enables you to perform multiple changes that leave the original item intact:

```
UIBezierPath *pathCopy = [myPath copy];
[pathCopy applyTransform: CGAffineTransformMakeScale(0.5f, 0.5f)];
```

You cannot "revert" a path by applying an identity transform. It produces a "no op" result (the path stays the same) rather than a reversion (the path returns to what it originally was).

The Origin Problem

Apply transforms don't always produce the results you expect. Take Figure 4-7, for example. I created a path and applied a rotation to it:

```
[path applyTransform:CGAffineTransformMakeRotation(M_PI / 9)];
```

The left image in Figure 4-7 shows what you might expect to happen, where the image rotates 20 degrees around its center. The right image shows what actually happens when you do not control a transform's point of origin. The path rotates around the origin of the current coordinate system.



Figure 4-7 A rotation whose origin is not explicitly set to a path's center produces unexpected results.

Fortunately, it's relatively easy to ensure that rotation and scaling occur the way you anticipate. Listing 4-4 details `ApplyCenteredPathTransform()`, a function that bookends an affine transform with translations that set and then, afterward, reset the coordinate system. These extra steps produce the controlled results you're looking for.

Listing 4-4 Rotating a Path Around Its Center

```
// Translate path's origin to its center before applying the transform
void ApplyCenteredPathTransform(
    UIBezierPath *path, CGAffineTransform transform)
{
    CGPoint center = PathBoundingCenter(path);
    CGAffineTransform t = CGAffineTransformIdentity;
    t = CGAffineTransformTranslate(t, center.x, center.y);
    t = CGAffineTransformConcat(transform, t);
    t = CGAffineTransformTranslate(t, -center.x, -center.y);
    [path applyTransform:t];
}

// Rotate path around its center
void RotatePath(UIBezierPath *path, CGFloat theta)
{
    CGAffineTransform t =
        CGAffineTransformMakeRotation(theta);
    ApplyCenteredPathTransform(path, t);
}
```

Other Transformations

As with rotation, if you want to scale an object in place, make sure you apply your transform at the center of a path's bounds. Figure 4-8 shows a transform that scales a path by 85% in each dimension. The function that created this scaling is shown in Listing 4-5, along with a number of other convenient transformations.

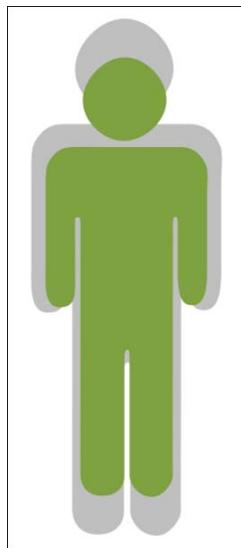


Figure 4-8 The smaller green image is scaled at (0.85, 0.85) of the original.

Listings 4-4 and 4-5 implement many of the most common transforms you need for Quartz drawing tasks. In addition to the rotation and scaling options, these functions enable you to move a path to various positions and to mirror the item in place. Centered origins ensure that each transform occurs in the most conventionally expected way.

Listing 4-5 Applying Transforms

```
// Scale path to sx, sy
void ScalePath(UIBezierPath *path, CGFloat sx, CGFloat sy)
{
    CGAffineTransform t = CGAffineTransformMakeScale(sx, sy);
    ApplyCenteredPathTransform(path, t);
}

// Offset a path
void OffsetPath(UIBezierPath *path, CGSize offset)
{
    CGAffineTransform t =
        CGAffineTransformMakeTranslation(
            offset.width, offset.height);
    ApplyCenteredPathTransform(path, t);
}

// Move path to a new origin
```

```

void MovePathToPoint(UIBezierPath *path, CGPoint destPoint)
{
    CGRect bounds = PathBoundingBox(path);
    CGSize vector =
        PointsMakeVector(bounds.origin, destPoint);
    OffsetPath(path, vector);
}

// Center path around a new point
void MovePathCenterToPoint(
    UIBezierPath *path, CGPoint destPoint)
{
    CGRect bounds = PathBoundingBoxCenter(path);
    CGSize vector = PointsMakeVector(bounds.origin, destPoint);
    vector.width -= bounds.size.width / 2.0f;
    vector.height -= bounds.size.height / 2.0f;
    OffsetPath(path, vector);
}

// Flip horizontally
void MirrorPathHorizontally(UIBezierPath *path)
{
    CGAffineTransform t = CGAffineTransformMakeScale(-1, 1);
    ApplyCenteredPathTransform(path, t);
}

// Flip vertically
void MirrorPathVertically(UIBezierPath *path)
{
    CGAffineTransform t = CGAffineTransformMakeScale(1, -1);
    ApplyCenteredPathTransform(path, t);
}

```

Fitting Bezier Paths

Listing 4-6 tackles one of the most important tasks for working with Bezier paths: working with a path created at an arbitrary point and scale, and drawing it into a specific rectangle. Fitting vector-based art ensures that you can reliably place items into your drawing with expected dimensions and destinations.

Listing 4-6 uses the same fitting approach you've seen used several times already in this book. The difference is that this version uses the scale and fitting rectangle to apply transforms to the path. The path is moved to its new center and then scaled in place down to the proper size.

Listing 4-6 Fitting a Path

```
void FitPathToRect(UIBezierPath *path, CGRect destRect)
{
    CGRect bounds = PathBoundingBox(path);
    CGRect fitRect = RectByFittingRect(bounds, destRect);
    CGFloat scale = AspectScaleFit(bounds.size, destRect);

    CGPoint newCenter = RectGetCenter(fitRect);
    MovePathCenterToPoint(path, newCenter);
    ScalePath(path, scale, scale);
}
```

Creating Bezier Paths from Strings

Core Text simplifies the process of transforming strings to Bezier paths. Listing 4-7 provides a simple conversion function. It transforms its string into individual Core Text glyphs, represented as individual CGPath items. The function adds each letter path to a resulting Bezier path, offsetting itself after each letter by the size of that letter.

After all the letters are added, the path is mirrored vertically. This converts the Quartz-oriented output into a UIKit-appropriate layout. You can treat these string paths just like any others, setting their line widths, filling them with colors and patterns, and transforming them however you like. Figure 4-9 shows a path created from a bold Baskerville font and filled with a green pattern.



Figure 4-9 This is a filled and stroked Bezier path created by Listing 4-7 from an NSString instance.

Here's the snippet that created that path:

```
UIFont *font = [UIFont fontWithName:@"Baskerville-Bold" size:16];
UIBezierPath *path = BezierPathFromString(@"Hello World", font);
FitPathToRect(path, targetRect);
[path fill:GreenStripesColor()];
[path strokeInside:4];
```

Interestingly, font size doesn't play a role in this particular drawing. The path is proportionately scaled to the destination rectangle, so you can use almost any font to create the source.

If you want the path to look like normal typeset words, just fill the returned path with a black fill color without stroking. This green-filled example used an inside stroke to ensure that the edges of the type path remained crisp.

Listing 4-7 Creating Bezier Paths from Strings

```
UIBezierPath *BezierPathFromString(
    NSString *string, UIFont *font)
{
    // Initialize path
    UIBezierPath *path = [UIBezierPath bezierPath];
    if (!string.length) return path;

    // Create font ref
    CTFontRef fontRef = CTFontCreateWithName(
        (__bridge CFStringRef)font.fontName,
        font.pointSize, NULL);
    if (fontRef == NULL)
    {
        NSLog(@"Error retrieving CTFontRef from UIFont");
        return nil;
    }

    // Create glyphs (that is, individual letter shapes)
    CGGlyph *glyphs = malloc(sizeof(CGGlyph) * string.length);
    const unichar *chars = (const unichar *)[string
        cStringUsingEncoding:NSUTFStringEncoding];
    BOOL success = CTFontGetGlyphsForCharacters(
        fontRef, chars, glyphs, string.length);
    if (!success)
    {
        NSLog(@"Error retrieving string glyphs");
        CFRelease(fontRef);
        free(glyphs);
        return nil;
    }

    // Draw each char into path
    for (int i = 0; i < string.length; i++)
    {
        // Glyph to CGPath
        CGGlyph glyph = glyphs[i];
        CGPathRef pathRef =
            CTFontCreatePathForGlyph(fontRef, glyph, NULL);

        // Append CGPath
    }
}
```

```
[path appendPath:[UIBezierPath  
bezierPathWithCGPath:pathRef]]  
  
    // Offset by size  
    CGSize size =  
        [[string substringWithRange: NSMakeRange(i, 1)]  
        sizeWithAttributes:@{NSFontAttributeName:font}];  
    OffsetPath(path, CGSizeMake(-size.width, 0));  
}  
  
// Clean up  
free(glyphs); CFRelease(fontRef);  
  
// Return the path to the UIKit coordinate system  
MirrorPathVertically(path);  
return path;  
}
```

Adding Dashes

UIKit makes it easy to add dashes to a Bezier path. For example, you might add a simple repeating pattern, like this:

```
CGFloat dashes[] = {6, 2};  
[path setLineDash:dashes count:2 phase:0];
```

The array specifies, in points, the on/off patterns used when drawing a stroke. This example draws lines of 6 points followed by spaces of 2 points. Your dash patterns will naturally vary by the scale of the drawing context and any compression used by the view. That said, there are many ways to approach dashes. Table 4-1 showcases some basic options.

The phase of a dash indicates the amount it is offset from the beginning of the drawing. In this example, the entire pattern is 8 points long. Moving the phase iteratively from 0 up to 7 enables you to animate the dashes, creating a marching ants result that surrounds a Bezier path. You will read about how to accomplish this effect in Chapter 7.

Table 4-1 Dash Patterns



Paths default to solid lines without dashes.



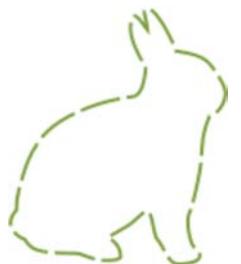
Use a (1, 1) dash to create a dot-like pattern.



Increasing the size of the dashes and spaces extends the repeating pattern. This stroke uses a (2, 2) dash.



This path uses a (6, 2) stroke. This offers a dashed pattern suitable for most developer needs.



When you increase the path to (12, 2), the spaces become less noticeable, although the path is still clearly dashed.



Alternate long and short dashes for another take on dash style. This pattern uses (8, 2, 4, 2) strokes.



Patterns can be more complex. This pattern uses a (8, 2, 4, 2, 4, 2) stroke. It combines one long stroke with two short ones.

This path uses the same approach but reduces the size of each item. The result is a dot-and-line mix, stroked using (6, 1, 1, 1, 1).

This final pattern uses a (2, 1, 4, 1, 8, 1, 4, 1, 2, 1) stroke. The lines stretch and reduce by powers of 2.

Building a Polygon Path

Although `UIBezierPath` offers easy ovals and rectangles, it does not provide a simple *N*-gon generator. Listing 4-8 fills that gap, returning Bezier paths with the number of sides you request. Some basic shapes it creates are shown in Figure 4-10.

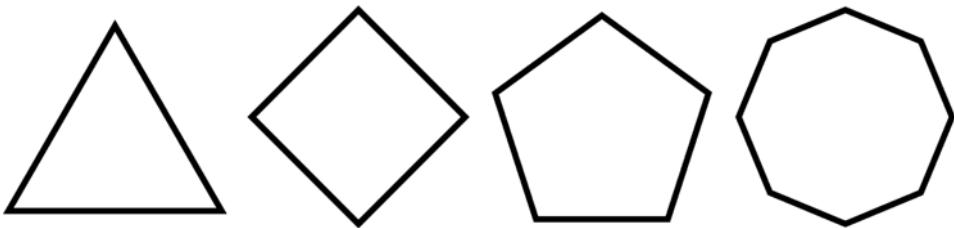


Figure 4-10 These polygon Bezier paths were generated by Listing 4-8. The shapes have, respectively, 3, 4, 5, and 8 sides.

This function creates its shapes by dividing a circle (2π radians) into the requested number of sides and then drawing lines from one destination point to the next. The final segment closes the path, avoiding drawing artifacts at the starting point of the shape. Figure 4-11 shows the kind of effect you'll encounter when you don't properly close your shape. Quartz treats that corner as two line segments instead of a proper join.

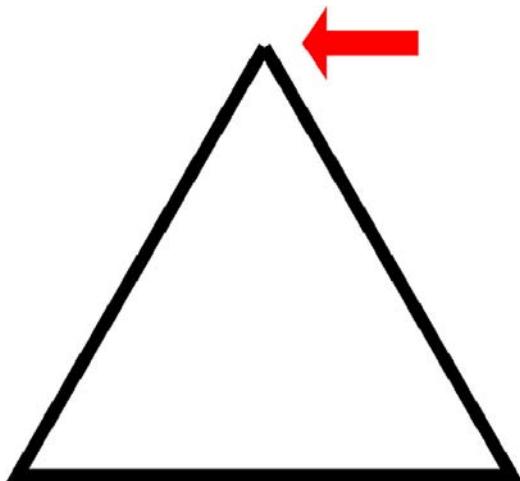


Figure 4-11 Avoid stroking gaps by closing the path from the last point to the origin.

All returned paths use unit sizing—that is, they fit within the $\{0, 0, 1, 1\}$ rectangle. You size the path as needed. Also, the first point is always placed here at the top of the shape, so if you intend to return a square instead of a diamond, make sure to rotate by 90 degrees.

Listing 4-8 Generating Polygons

```
UIBezierPath *BezierPolygon(NSUInteger numberOfSides)
{
    if (numberOfSides < 3)
    {
        NSLog(@"Error: Please supply at least 3 sides");
        return nil;
    }

    UIBezierPath *path = [UIBezierPath bezierPath];

    // Use a unit rectangle as the destination
    CGRect destinationRect = CGRectMake(0, 0, 1, 1);
    CGPoint center = RectGetCenter(destinationRect);
    CGFloat r = 0.5f; // radius

    BOOL firstPoint = YES;
    for (int i = 0; i < (numberOfSides - 1); i++)
    {
```

```

CGFloat theta = M_PI + i * TWO_PI / numberOfSides;
CGFloat dTheta = TWO_PI / numberOfSides;

CGPoint p;
if (firstPoint)
{
    p.x = center.x + r * sin(theta);
    p.y = center.y + r * cos(theta);
    [path moveToPoint:p];
    firstPoint = NO;
}

p.x = center.x + rx * sin(theta + dTheta);
p.y = center.y + ry * cos(theta + dTheta);
[path addLineToPoint:p];
}

[path closePath];

return path;
}

```

Line Joins and Caps

A path's `lineJoinStyle` property establishes how to draw the point at which each line meets another. Quartz offers three styles for you to work with, which are shown in Figure 4-12. The default, `kCGLineJoinMiter`, creates a sharp angle. You round those edges by choosing `kCGLineJoinRound` instead. The final style is `kCGLineJoinBevel`. It produces flat endcaps with squared-off ends.

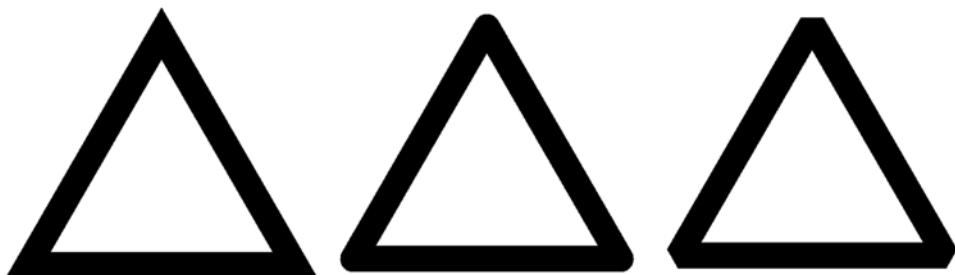


Figure 4-12 From left to right: mitered, rounded, and beveled joins.

Lines points that don't join other lines have their own styles, called *caps*. Figure 4-13 shows the three possible cap styles. I've added gray vertical lines to indicate the natural endpoint of each line. The `kCGLineCapButt` style ends exactly with the line. In two of the three, however, the line ornament extends *beyond* the final point. This additional distance for `kCGLineCapSquare` and `kCGLineCapRound` is half the line's width. The wider the line grows, the longer the line cap extends.

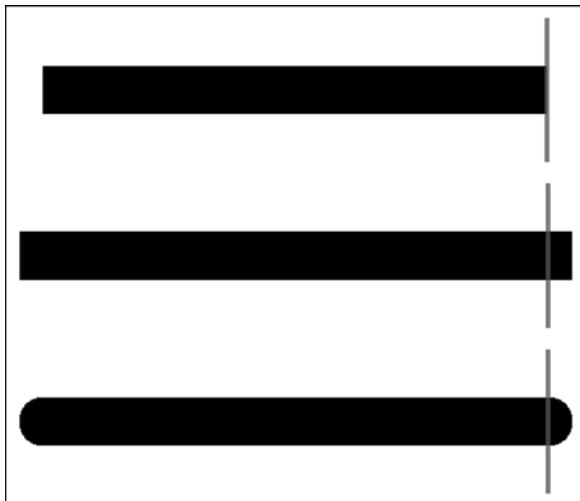


Figure 4-13 From top to bottom: butt, square, and round line caps.

Miter Limits

Miter limits restrict the pointiness of a shape, as you see in Figure 4-14. When the diagonal length of a miter—that is, the triangular join between two lines—exceeds a path's limit, Quartz converts those points into bevel joins instead.

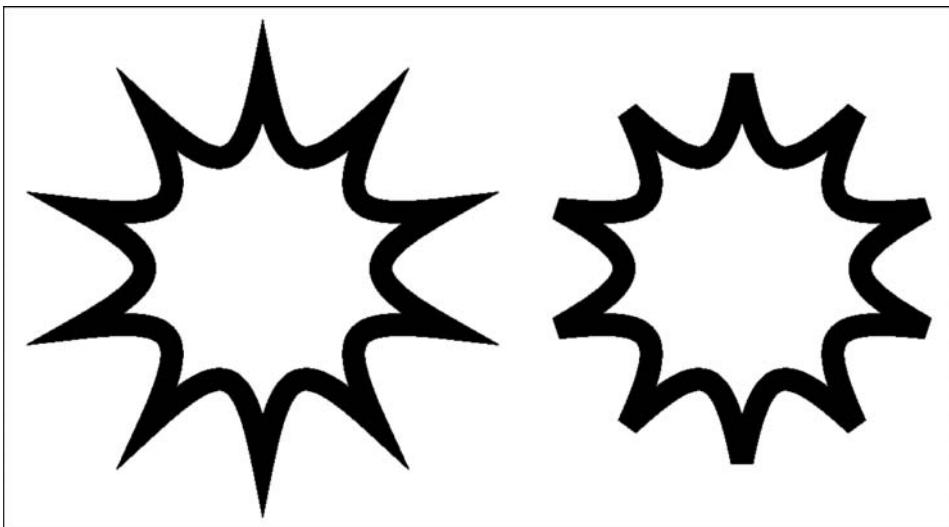


Figure 4-14 When shapes reach their miter limit, angled joins convert to flat bevels.

In the left image, the shape has sharp angles. Its pointy extensions have not reached its path's `miterLimit`. Lowering the limit below the actual length of those points, as in the right image, forces those joins to be clipped.

The default Bezier path miter limit is 10 points. This affects any angle under 11 degrees. As you change the miter limit, the default cutoff point adjusts. For example, at 30 degrees, you'd have to lower the limit below 3.8 points in order to see any effect. Miters start growing most extremely in the 0- to 11-degree range. The natural miter that's just 10.4 points long for an 11-degree angle becomes 23 points at 5 degrees and almost 60 points at 2 degrees. Adding reasonable limits ensures that as your angles grow small, miter lengths won't overwhelm your drawings.

Appendix B offers a more in-depth look at the math behind the miters.

Inflected Shapes

Figure 4-14 shows an example of a simple shape built by adding cubic curves around a center point. It was created by using the same function that built the shapes you see in Figure 4-15.

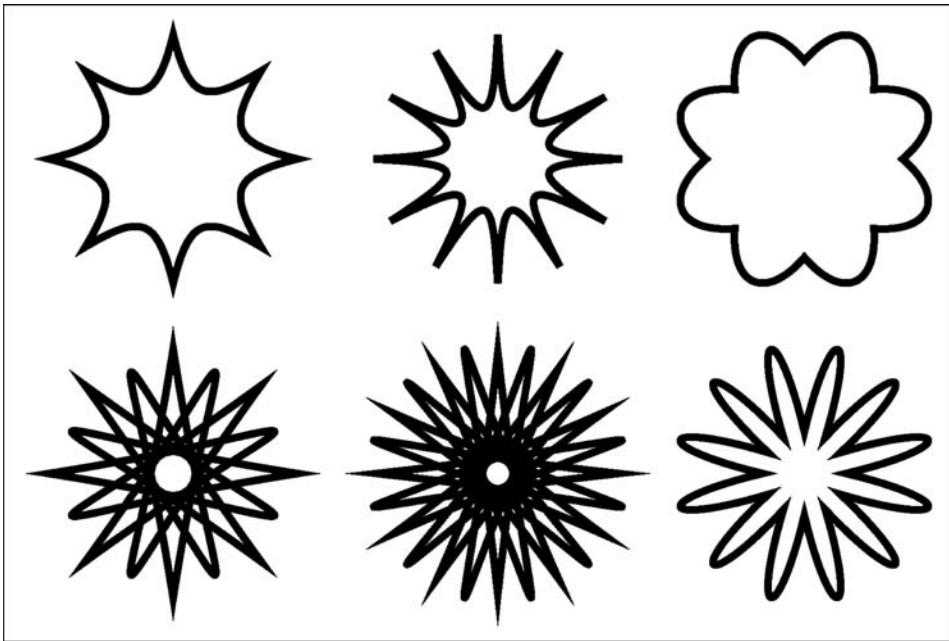


Figure 4-15 Inflection variations: The fourth and fifth shapes shown here use large negative inflections that cross the center. Top row: 8 points and -0.5 inflection, 12 points and -0.75 inflection, 8 points and 0.5 inflection. Bottom row: 8 points and -2.5 inflection, 12 points and -2.5 inflection, 12 points and 2.5 inflection.

If you think the code in Listing 4-9 looks very similar to the polygon generator in Listing 4-8, you’re right. The difference between the two functions is that Listing 4-9 creates a curve between its points instead of drawing a straight line. You specify how many curves to create.

The inflection of that curve is established by two control points. These points are set by the `percentInflection` parameter you pass to the function. Positive inflections move further away from the center, building lobes around the shape. Negative inflections move toward the center—or even past the center—creating the spikes and loops you see on the other figures.

Listing 4-9 Generating Inflected Shapes

```
UIBezierPath *BezierInflectedShape (
   NSUInteger numberOfInflections,
    CGFloat percentInflection)
{
    if (numberOfInflections < 3)
    {
```

```
    NSLog(@"Error: Please supply at least 3 inflections");
    return nil;
}

UIBezierPath *path = [UIBezierPath bezierPath];
CGRect destinationRect = CGRectMake(0, 0, 1, 1);
CGPoint center = RectGetCenter(destinationRect);
CGFloat r = 0.5;
CGFloat rr = r * (1.0 + percentInflection);

BOOL firstPoint = YES;
for (int i = 0; i < number_of_inflections; i++)
{
    CGFloat theta = i * TWO_PI / number_of_inflections;
    CGFloat dTheta = TWO_PI / number_of_inflections;

    if (firstPoint)
    {
        CGFloat xa = center.x + r * sin(theta);
        CGFloat ya = center.y + r * cos(theta);
        CGPoint pa = CGPointMake(xa, ya);
        [path moveToPoint:pa];
        firstPoint = NO;
    }

    CGFloat cp1x = center.x + rr * sin(theta + dTheta / 3);
    CGFloat cp1y = center.y + rr * cos(theta + dTheta / 3);
    CGPoint cp1 = CGPointMake(cp1x, cp1y);

    CGFloat cp2x = center.x + rr *
        sin(theta + 2 * dTheta / 3);
    CGFloat cp2y = center.y + rr *
        cos(theta + 2 * dTheta / 3);
    CGPoint cp2 = CGPointMake(cp2x, cp2y);

    CGFloat xb = center.x + r * sin(theta + dTheta);
    CGFloat yb = center.y + r * cos(theta + dTheta);
    CGPoint pb = CGPointMake(xb, yb);

    [path addCurveToPoint:pb
        controlPoint1:cp1 controlPoint2:cp2];
}

[path closePath];
return path;
}
```

By tweaking Listing 4-9 to draw lines instead of curves, you create star shapes instead. In Listing 4-10, a single control point placed halfway between each point provides a destination for the angles of a star. Figure 4-16 highlights some shapes you can produce using Listing 4-10.

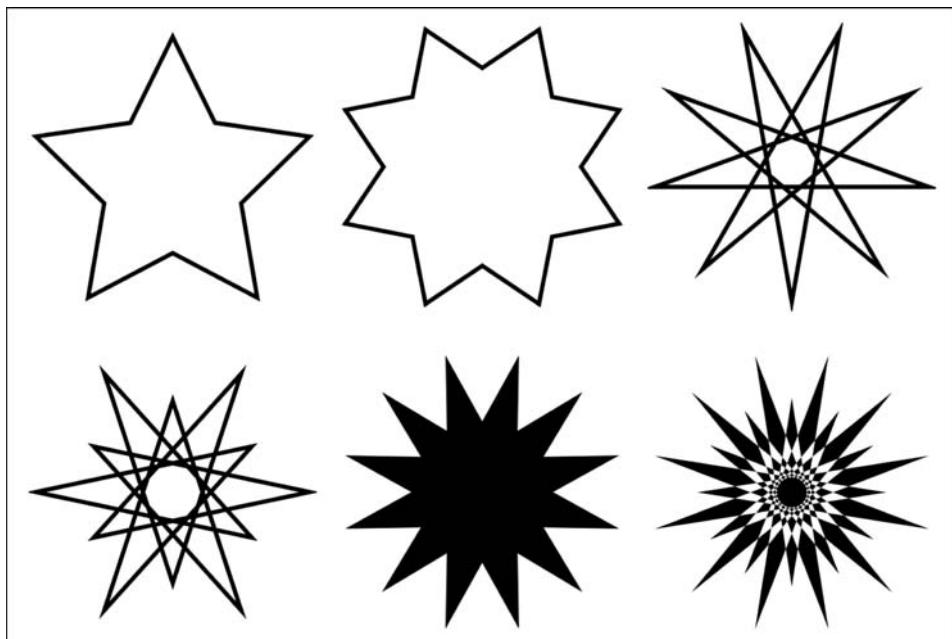


Figure 4-16 An assortment of stars built with Listing 4-10. Top row: 5 points and 0.75 inflection, 8 points and 0.5 inflection, 9 points and -2 inflection. Bottom row: 6 points and -1.5 inflection, 12 points and 0.75 inflection (filled). The last item, which uses the even/odd fill rule, was built with `BezierStarShape(12, -2.5)`.

Listing 4-10 Generating Star Shapes

```
UIBezierPath *BezierStarShape(
    NSUInteger number_of_inflections, CGFloat percent_inflection)
{
    if (number_of_inflections < 3)
    {
        NSLog(@"Error: Please supply at least 3 inflections");
        return nil;
    }

    UIBezierPath *path = [UIBezierPath bezierPath];
    CGRect destination_rect = CGRectMake(0, 0, 1, 1);
```

```

CGPoint center = RectGetCenter(destinationRect);
CGFloat r = 0.5;
CGFloat rr = r * (1.0 + percentInflection);

BOOL firstPoint = YES;
for (int i = 0; i < numberOfInflections; i++)
{
    CGFloat theta = i * TWO_PI / numberOfInflections;
    CGFloat dTheta = TWO_PI / numberOfInflections;

    if (firstPoint)
    {
        CGFloat xa = center.x + r * sin(theta);
        CGFloat ya = center.y + r * cos(theta);
        CGPoint pa = CGPointMake(xa, ya);
        [path moveToPoint:pa];
        firstPoint = NO;
    }

    CGFloat cplx = center.x + rr *
        sin(theta + dTheta / 2);
    CGFloat cply = center.y + rr *
        cos(theta + dTheta / 2);
    CGPoint cp1 = CGPointMake(cplx, cply);

    CGFloat xb = center.x + r * sin(theta + dTheta);
    CGFloat yb = center.y + r * cos(theta + dTheta);
    CGPoint pb = CGPointMake(xb, yb);

    [path addLineToPoint:cp1];
    [path addLineToPoint:pb];
}

[path closePath];
return path;
}

```

Summary

This chapter introduces the `UIBezierPath` class. It provides a basic overview of the class's role and the properties that you can adjust. You saw how to fill and stroke paths, as well as how to construct complex shapes. You read about building new instances in both conventional and unexpected ways. Before moving on to more advanced path topics, here are some final thoughts to take from this chapter:

- Paths provide a powerful tool for both representing geometric information and drawing that material into your apps. They create a vector-based encapsulation of a shape that can be used at any position, at any size, and at any rotation you need. Their resolution independence means they retain sharpness and detail, no matter what scale they're drawn at.
- When you turn strings into paths, you can have a lot of fun with text. From filling text with colors to playing with individual character shapes, there's a wealth of expressive possibilities for you to take advantage of.
- Take care when using bounding box calculations to estimate drawing bounds. Items like miters and line caps may extend beyond a path's base bounds when drawn.

This page intentionally left blank

5

Paths in Depth

Exposing path internals amps up the way you work with the `UIBezierPath` class. In particular, it enables you to leverage the `CGPathElement` data structures stored in each instance's underlying `CGPath` to produce solutions for many common iOS drawing challenges. Want to place objects along a path's curves? Want to divide the path into subpaths and color them individually? Element-based solutions enable you to do that.

This chapter starts off with rather a lot of didactic implementation details. It finishes with solutions that leverage these implementations, creating compelling drawing results for your iOS projects.

Path Elements

The `UIBezierPath` class supports three kinds of Bezier elements: line segments, quadratic curves, and cubic curves, which you see in Figure 5-1. Each element represents one of five operations that describe the way the path is laid out:

- `kCGPathElementMoveToPoint`—This element sets the path's position to a new location but does not add any lines or curves.
- `kCGPathElementAddLineToPoint`—This element adds a line segment from the previous point to the element's location. You see a line segment in Figure 5-1, in the top drawing.
- `kCGPathElementAddQuadCurveToPoint`—This element adds a quadratic curve from the previous point to the element's location. The middle drawing in Figure 5-1 is built with a quad curve, using a single control point.
- `kCGPathElementAddCurveToPoint`—This element adds a cubic curve from the previous point to the element's location. The bottom drawing in Figure 5-1 shows a cubic curve, with its two control points.

- **kCGPathElementCloseSubpath**—This element closes the subpath, drawing a line from the current point to the start of the most recent subpath. This start location is always set using a “move to point” element.

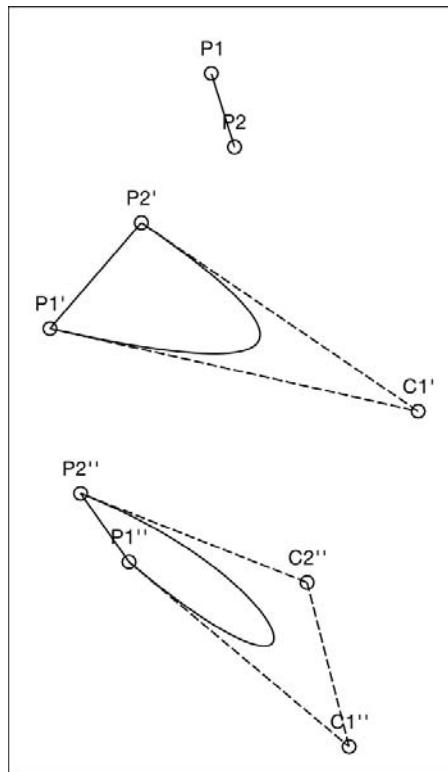


Figure 5-1 You can construct Bezier paths from lines (top), quadratic curves with a single control point (middle), and cubic curves with two control points (bottom).

Each `CGPathElement` stores an element type (one of the five types you just saw) and an array of `CGPoint` items:

```
struct CGPathElement {
    CGPathElementType type;
    CGPoint *points;
};
```

An element's `points` array may store zero, one, two, or three points. The number depends on the element's role in the path. Close-path elements don't define any points. The move-

to-point and line-to-point elements use one, which specifies the destination. The destination tells the element where to move or where to add a line.

Both quad and cubic curves require control points in addition to the destination. A quad curve stores a destination and one control point. A cubic curve stores a destination and two control points. What's more, the order of these points varies by the point type.

Wrapping these path elements in an Objective-C class, as shown in Listing 5-1, simplifies their use. This class hides the intricacies of the point array with its implementation details, such as which item is the destination point and which items are the control points. Each element-based object expresses a consistent set of properties, forming a stepping-stone for many handy `UIBezierPath` utilities.

Listing 5-1 An Objective-C Wrapper for `CGPathElement`

```
#define NULLPOINT CGRectNull.origin

@interface BezierElement : NSObject <NSCopying>
@property (nonatomic, assign) CGPathElementType elementType;
@property (nonatomic, assign) CGPoint point;
@property (nonatomic, assign) CGPoint controlPoint1;
@property (nonatomic, assign) CGPoint controlPoint2;
@end

@implementation BezierElement
- (instancetype) init
{
    self = [super init];
    if (self)
    {
        _elementType = kCGPathElementMoveToPoint;
        _point = NULLPOINT;
        _controlPoint1 = NULLPOINT;
        _controlPoint2 = NULLPOINT;
    }
    return self;
}

// Create a BezierElement object that represents
// the data stored in the passed element
+ (instancetype) elementWithPathElement:
    (CGPathElement) element
{
    BezierElement *newElement = [[self alloc] init];
    newElement.elementType = element.type;

    switch (newElement.elementType)
```

```

{
    case kCGPathElementCloseSubpath:
        break;
    case kCGPathElementMoveToPoint:
    case kCGPathElementAddLineToPoint:
    {
        newElement.point = element.points[0];
        break;
    }
    case kCGPathElementAddQuadCurveToPoint:
    {
        newElement.point = element.points[1];
        newElement.controlPoint1 = element.points[0];
        break;
    }
    case kCGPathElementAddCurveToPoint:
    {
        newElement.point = element.points[2];
        newElement.controlPoint1 = element.points[0];
        newElement.controlPoint2 = element.points[1];
        break;
    }
    default:
        break;
    }

    return newElement;
}

```

Converting Bezier Paths into Element Arrays

Quartz's `CGPathApply()` function iterates across all the elements that comprise a path. As Listing 5-2 demonstrates, this enables you to convert a `UIBezierPath` into an array of its components. This listing converts and collects those path elements along the way. The result is an `NSArray` of Objective-C `BezierElement` objects, each representing an original path element.

Listing 5-2 Extracting Element Arrays

```

// Convert one element to BezierElement and save to array
void GetBezierElements(void *info, const CGPathElement *element)
{
    NSMutableArray *bezierElements =
        (__bridge NSMutableArray *)info;

```

```

    if (element)
        [bezierElements addObject:
            [BezierElement elementWithPathElement:*element]];
}

// Retrieve array of component elements
- (NSArray *) elements
{
    NSMutableArray *elements = [NSMutableArray array];
    CGPathApply(self.CGPath, (__bridge void *)elements,
        GetBezierElements);
    return elements;
}

```

Extracting elements from Bezier paths enables you to perform geometric operations on the components. Afterward, you rebuild Bezier paths from the updated components. Listing 5-3 shows this reassembly. Its `BezierPathWithElements()` constructor function creates a new path and then iteratively calls `addToPath:` on each element. As you can see from the implementation, this method applies each element's point values and type to build the new path.

Listing 5-3 Reassembling Bezier Paths

```

// Construct a Bezier path from an element array
UIBezierPath *BezierPathWithElements(NSArray *elements)
{
    UIBezierPath *path = [UIBezierPath bezierPath];
    for (BezierElement *element in elements)
        [element addToPath:path];
    return path;
}

// This is a BezierElement method. The element
// adds itself to the path passed as the argument
- (void) addToPath: (UIBezierPath *) path
{
    switch (self.elementType)
    {
        case kCGPathElementCloseSubpath:
            [path closePath];
            break;
        case kCGPathElementMoveToPoint:
            [path moveToPoint:self.point];
            break;
        case kCGPathElementAddLineToPoint:

```

```

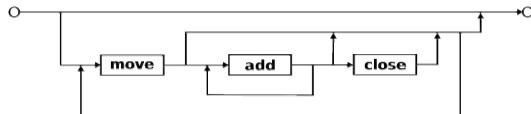
        [path addLineToPoint:self.point];
        break;
    case kCGPathElementAddQuadCurveToPoint:
        [path addQuadCurveToPoint:self.point
            controlPoint:self.controlPoint1];
        break;
    case kCGPathElementAddCurveToPoint:
        [path addCurveToPoint:self.point
            controlPoint1:self.controlPoint1
            controlPoint2:self.controlPoint2];
        break;
    default:
        break;
    }
}

```

Legal and Illegal Paths

A legal path always begins with a move operation followed by a series of lines and curves, and it ends with an optional close operation. You repeat as desired. Here's a summary of what makes a legal path:

path :- \emptyset | {move • (add)* • {add • close | \emptyset } }*



A path can be entirely empty, like this:

```
UIBezierPath *path = [UIBezierPath bezierPath]
```

Or a path can consist of a move operation followed by zero or more add elements. The following snippet creates a path by moving to point p1, adds a quad curve from point p1 to point p2 (using control point c1), and then closes that path with a line from point p2 to point p1:

```
[path moveToPoint:p1];
[path addQuadCurveToPoint:p2 controlPoint:c1];
[path closePath];
```

Figure 5-1 (middle) shows an example of what this path with its move, curve, and close elements might look like when drawn.

A close element, if used, should always follow an add line or add curve element. Here, it tells the path to add a line from the current point (p2) to the first point (p1).

Path State

Bezier operations depend on the path storing (or, more accurately, being able to derive) two key pieces of state. These are where the first point of the path was placed and where the current point of the path is placed.

For example, when you add a line to a point, the full line segment is *[path's current point, new requested point]*. When you close a path, the segment you create is *[path's current point, path's first point]*. These values are established as the path progresses. At each add operation, the new requested point becomes the current point:

```
[path moveToPoint:p1]; // First and current points are p1  
[path addQuadCurveToPoint:p2  
    controlPoint:c1]; // Current point is p2  
[path closePath]; // No current point
```

These state items are important because each add request requires at least one extra state that isn't specified by the method invocation. You don't say `addLineFrom:to:`. You either add a line or curve to some point or `closePath`. The path understands where the current point is and where the first point was. If it cannot establish these states, any new operation that refers to them is illegal.

Just because Xcode compiles your code without complaint doesn't mean the path you're building is legal. Example 5-1 shows an illegal path construction sequence. It starts by closing a path and then adds a quad curve.

Example 5-1 Constructing a Path with Multiple Closed Subpaths

```
UIBezierPath *path = [UIBezierPath bezierPath;  
[p closePath];  
[p addQuadCurveToPoint:p1 controlPoint:p2];  
[p stroke];
```

Executing this code produces the console errors you see here:

```
<Error>: void CGPathCloseSubpath(CGMutablePathRef) : no current point.  
<Error>: void CGPathAddQuadCurveToPoint(CGMutablePathRef,  
    const CGAffineTransform *, CGFloat, CGFloat, CGFloat, CGFloat):  
    no current point.
```

If you cut and paste this into a project, you'll discover that no exceptions are raised, and the app continues running. The path cannot be drawn, however. The current context remains unchanged, even after you apply the `stroke` command in the last line of Example 5-1.

Although this particular example demonstrates a robustness of the underlying drawing system, not all operations are so lucky. Some drawing errors *can* raise exceptions that interfere with application execution. Apple's buggy path-reversing

`bezierPathByReversingPath` method is one example of a method that has crashed apps on me.

Compound Paths

The `UIBezierPath` class enables you to add multiple paths together to build compound paths. As you can see in Figure 5-2, a path need not be continuous. It can contain multiple disjoint sections. Figure 5-2's path includes six subpaths, consisting of the four outer parts of the letters and the two holes (inside the p and a).

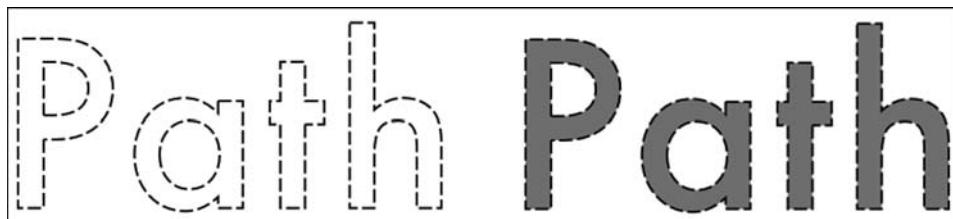


Figure 5-2 This compound path consists of six subpaths.

Consider an example of creating multiple subpaths in code. Example 5-2 builds a new path through a series of move, add, and close statements. The example uses two distinct move–add–close sequences, establishing a single path with two closed subpaths.

Example 5-2 Constructing a Path with Multiple Closed Subpaths

```
UIBezierPath *path = [UIBezierPath bezierPath];  
  
// Begin subpath 1  
[path moveToPoint:p1];  
[path addLineToPoint:p2];  
[path addCurveToPoint:p3 controlPoint1:p4 controlPoint2:p5];  
[path closePath];  
  
// Begin subpath 2  
[path moveToPoint:p6];  
[path addQuadCurveToPoint:p7 controlPoint:p8];  
[path addQuadCurveToPoint:p9 controlPoint:p0];  
[path closePath];
```

You can append entire paths by using `appendPath:`. This option is particularly helpful when you are assembling drawing items from ready-built path components, as is done in Example 5-3.

Example 5-3 Constructing a Path with Multiple Closed Subpaths

```
UIBezierPath *path = [UIBezierPath bezierPath];
[path appendPath:[UIBezierPath
    bezierPathWithRect:rect1]];
[path appendPath:[UIBezierPath
    bezierPathWithOvalInRect:rect2]];
```

Compound paths enable you to build complex drawings with holes and standalone elements. In Figure 5-2, you saw a path with six distinct subpaths: the outer hulls of each letter and the holes in the p and a. The even/odd rule introduced in Chapter 4 ensures that the interiors of these two holes remain empty when you fill the path, as shown in Figure 5-2 (bottom).

Math Behind the Path

As you've now discovered, the `UIBezierPath` class supports three kinds of Bezier segments: linear, quadratic, and cubic. Each of these participates in the class to create complex shapes for UIKit and Quartz drawing routines. Retrieving component lines and curves enables you to calculate their lengths and interpolate along their paths. With this knowledge, you can apply drawing functions at computed points, as shown in Figure 5-3.

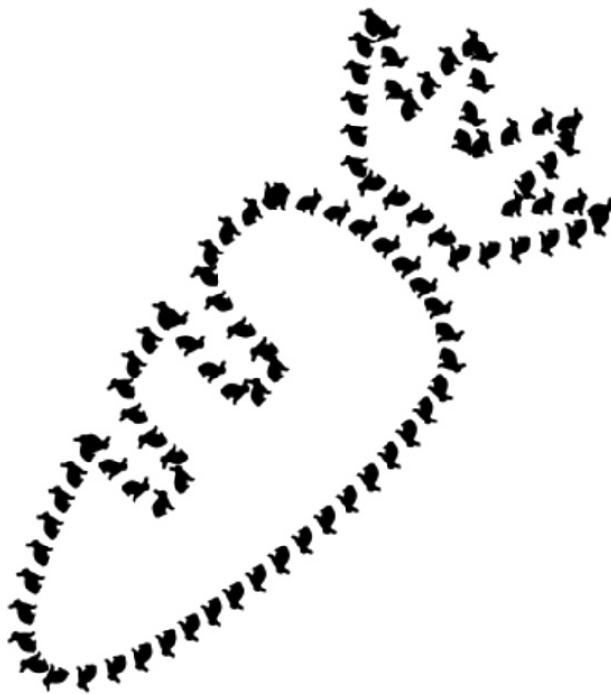


Figure 5-3 This carrot is drawn with Bezier bunny images placed at equal intervals along the carrot path.

Unfortunately, the `CGPathElement` structure doesn't offer you much to work with. It provides an element type that might be one of the following: move to point, add line to point, add curve to point, add quad curve to point, or close path. So how do you calculate the intermediate points between one point and the next?

With linear items, it's easy. You calculate the vector from one point to the next and scale it to the percentage of progress. Listing 5-4 shows an interpolating function.

Listing 5-4 Interpolating on Line Segments

```
// Interpolate between p1 and p2
CGPoint InterpolateLineSegment(
    CGPoint p1, CGPoint p2,
    CGFloat percent, CGPoint *slope)
{
    CGFloat dx = p2.x - p1.x;
    CGFloat dy = p2.y - p1.y;
```

```

    if (slope)
        *slope = CGPointMake(dx, dy);

    CGFloat px = p1.x + dx * percent;
    CGFloat py = p1.y + dy * percent;

    return CGPointMake(px, py);
}

```

But given a curve, how do you interpolate? Fortunately, you can apply the same curve math that the `UIBezierPath` class uses. Listing 5-5 offers functions for cubic (two control points) and quadratic (one control point) interpolation, calculating those values. You supply the percentage of progress (from 0 to 1), the start value, the end value, and the one or two control values. The functions return the interpolated points.

Listing 5-5 Interpolating Bezier Curves

```

// Calculate Cubic curve
CGFloat CubicBezier(CGFloat t, CGFloat start,
                     CGFloat c1, CGFloat c2, CGFloat end)
{
    CGFloat t_ = (1.0 - t);
    CGFloat tt_ = t_ * t_;
    CGFloat ttt_ = t_ * t_ * t_;
    CGFloat tt = t * t;
    CGFloat ttt = t * t * t;

    return start * ttt_
    + 3.0 * c1 * tt_ * t
    + 3.0 * c2 * t_ * tt
    + end * ttt;
}

// Calculate quad curve
CGFloat QuadBezier(CGFloat t, CGFloat start,
                    CGFloat c1, CGFloat end)
{
    CGFloat t_ = (1.0 - t);
    CGFloat tt_ = t_ * t_;
    CGFloat tt = t * t;

    return start * tt_
    + 2.0 * c1 * t_ * t
    + end * tt;
}

```

```

// Interpolate points on a cubic curve
CGPoint CubicBezierPoint(CGFloat t, CGPoint start,
    CGPoint c1, CGPoint c2, CGPoint end)
{
    CGPoint result;
    result.x = CubicBezier(t, start.x, c1.x, c2.x, end.x);
    result.y = CubicBezier(t, start.y, c1.y, c2.y, end.y);
    return result;
}

// Interpolate points on a quad curve
CGPoint QuadBezierPoint(CGFloat t,
    CGPoint start, CGPoint c1, CGPoint end)
{
    CGPoint result;
    result.x = QuadBezier(t, start.x, c1.x, end.x);
    result.y = QuadBezier(t, start.y, c1.y, end.y);
    return result;
}

```

Calculating Path Distance

Before you can say “move 35% along a path,” you must be able to evaluate a path’s length. That’s what the functions and method in Listing 5-6 provide for you. They return a value, in points, that represents the path’s extent at its current scale.

The `pathLength` method calculates a Bezier path’s length by iteratively applying the `ElementDistanceFromPoint()` function to each of its elements. This function uses path state (specifically the current and first points) to return the distance for each successive path element.

This depends on three functions that calculate linear distance and distance along a cubic or a quad Bezier curve. Curves are sampled N times; you specify what that sampling number is. In this listing, it is six. This turns out to be a decent approximation for most curves. Some implementations reduce that number to three to increase the overall efficiency. The trade-off is this: The fewer samples you take, the less accurate your distance measures will be.

Figure 5-4 shows a real-world example, where I calculated the differences between the three-point and six-point values. The results for each curve equal the sum of the linear distances between the sample points. In this case, the three-point sample was approximately 6% shorter than the six-point sample. As the curvature increased, so did the sampling differences, reaching up to 10%–15% for highly curved cubic sections.

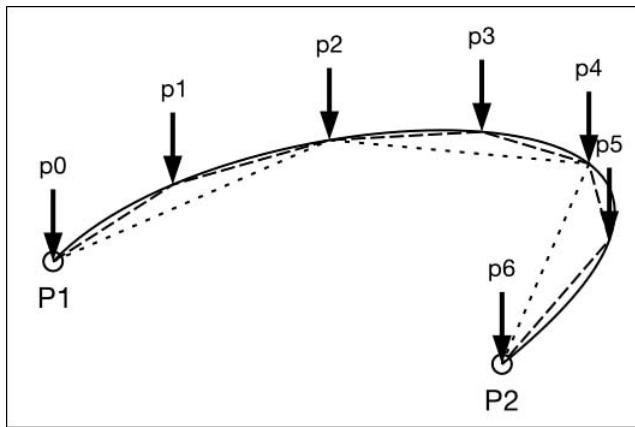


Figure 5-4 As the number of sample segments increases, path length measurements grow more accurate. The solid line is the original curve. The dashed line uses six samples to approximate the curve. The dotted line uses three samples.

Of course, there are trade-offs. As you raise the number of samples, the time to calculate the curve approximation increases, as does the accuracy of the measure—to a point. Too many samples, and you’re just spinning your CPU’s wheels (metaphorically), without substantial mathematical improvement in your measurement.

Listing 5-6 Element Distance

```
// Distance from p1 to p2
CGFloat PointDistanceFromPoint(CGPoint p1, CGPoint p2)
{
    CGFloat dx = p2.x - p1.x;
    CGFloat dy = p2.y - p1.y;
    return sqrt(dx*dx + dy*dy);
}

// How many points to interpolate
#define NUMBER_OF_BEZIER_SAMPLES 6

// Cubic length
float CubicBezierLength(
    CGPoint start, CGPoint c1, CGPoint c2, CGPoint end)
{
    int steps = NUMBER_OF_BEZIER_SAMPLES;
    CGPoint current;
    CGPoint previous;
    float length = 0.0;
```

```

        for (int i = 0; i <= steps; i++)
    {
        float t = (float) i / (float) steps;
        current = CubicBezierPoint(t, start, c1, c2, end);
        if (i > 0)
            length += PointDistanceFromPoint(
                current, previous);
        previous = current;
    }

    return length;
}

// Quad length
float QuadBezierLength(
    CGPoint start, CGPoint c1, CGPoint end)
{
    int steps = NUMBER_OF_BEZIER_SAMPLES;
    CGPoint current;
    CGPoint previous;
    float length = 0.0;

    for (int i = 0; i <= steps; i++)
    {
        float t = (float) i / (float) steps;
        current = QuadBezierPoint(t, start, c1, end);
        if (i > 0)
            length += PointDistanceFromPoint(
                current, previous);
        previous = current;
    }

    return length;
}

// Calculate element-to-element distance
CGFloat ElementDistanceFromPoint(
    BezierElement *element, CGPoint point, CGPoint startPoint)
{
    CGFloat distance = 0.0f;
    switch (element.elementType)
    {
        case kCGPathElementMoveToPoint:
            return 0.0f;
        case kCGPathElementCloseSubpath:

```

```

        return PointDistanceFromPoint(point, startPoint);
    case kCGPathElementAddLineToPoint:
        return PointDistanceFromPoint(point, element.point);
    case kCGPathElementAddCurveToPoint:
        return CubicBezierLength(point,
            element.controlPoint1, element.controlPoint2,
            element.point);
    case kCGPathElementAddQuadCurveToPoint:
        return QuadBezierLength(point, element.controlPoint1,
            element.point);
    }
    return distance;
}

// Bezier pathLength property
- (CGFloat) pathLength
{
    NSArray *elements = self.elements;
    CGPoint current = NULLPOINT;
    CGPoint firstPoint = NULLPOINT;
    float totalPointLength = 0.0f;

    for (BezierElement *element in elements)
    {
        totalPointLength += ElementDistanceFromPoint(
            element, current, firstPoint);
        if (element.elementType == kCGPathElementMoveToPoint)
            firstPoint = element.point;
        else if (element.elementType ==
            kCGPathElementCloseSubpath)
            firstPoint = NULLPOINT;

        if (element.elementType != kCGPathElementCloseSubpath)
            current = element.point;
    }

    return totalPointLength;
}

```

Interpolating Paths

Listing 5-7 enables you to interpolate a path instance to find a point that's some percentage along it. The code is extensive because of two things. First, all cases must be considered—linear, cubic curves, and quadratic curves. That involves a bunch of switch statements to consider and calculate each possibility.

Second, this method returns an optional slope—that's the `CGPoint` address passed as the last parameter. Calculating a slope requires about as much code as calculating the actual point in question. You want a slope value because it expresses the tangent to the path's curve at the point in question. This enables you to do things like have all your shapes orient themselves toward the inside of the curve, as you saw in Figure 5-3.

Note that reversing the path reverses the slope. Items at each point that used to represent the tangent on the outside of the curve flip to the inside and vice versa. That's because the tangent function is symmetrical across the Y-axis. Moving from point `p2` to point `p1` instead of from `p1` to `p2` produces a tangent value, and a resulting angle, that's the negative of the original.

Using this percentage method is expensive in terms of calculation. Where possible, you may want to precompute these interpolation values and use a cached percentage-to-point array to speed up animations and layout.

Listing 5-7 Path Interpolation

```
// Calculate a point that's a given percentage along a path
- (CGPoint) pointAtPercent: (CGFloat) percent
    withSlope: (CGPoint *) slope
{
    // Retrieve path elements
    NSArray *elements = self.elements;

    if (percent == 0.0f)
    {
        BezierElement *first = [elements objectAtIndex:0];
        return first.point;
    }

    // Retrieve the full path distance
    float pathLength = self.pathLength; // see Listing 5-6
    float totalDistance = 0.0f;

    // Establish the current and firstPoint states
    CGPoint current = NULLPOINT;
    CGPoint firstPoint = NULLPOINT;

    // Iterate through elements until the percentage
    // no longer overshoots
    for (BezierElement *element in elements)
    {
        float distance = ElementDistanceFromPoint(
            element, current, firstPoint);
        CGFloat proposedTotalDistance =
            totalDistance + distance;
```

```

CGFloat proposedPercent =
    proposedTotalDistance / pathLength;

if (proposedPercent < percent)
{
    // Consume and continue
    totalDistance = proposedTotalDistance;

    if (element.elementType ==
        kCGPathElementMoveToPoint)
        firstPoint = element.point;

    current = element.point;

    continue;
}

// What percent between p1 and p2?
CGFloat currentPercent = totalDistance / pathLength;
CGFloat dPercent = percent - currentPercent;
CGFloat percentDistance = dPercent * pathLength;
CGFloat targetPercent = percentDistance / distance;

// Return result
CGPoint point = InterpolatePointFromElement(
    element, current, firstPoint,
    targetPercent, slope);
return point;
}

return NULLPOINT;
}

// Interpolate individual distances along element
CGPoint InterpolatePointFromElement(
    BezierElement *element, CGPoint point,
    CGPoint startPoint, CGFloat percent,
    CGPoint *slope)
{
    switch (element.elementType)
    {
        case kCGPathElementMoveToPoint:
        {
            // No distance
            if (slope)
                *slope = CGPointMake(INFINITY, INFINITY);
        }
    }
}

```

```

        return point;
    }

case kCGPathElementCloseSubpath:
{
    // from self.point to firstPoint
    CGPoint p = InterpolateLineSegment(
        point, startPoint, percent, slope);
    return p;
}

case kCGPathElementAddLineToPoint:
{
    // From point to self.point
    CGPoint p = InterpolateLineSegment(
        point, element.point, percent, slope);
    return p;
}

case kCGPathElementAddQuadCurveToPoint:
{
    // From point to self.point
    CGPoint p = QuadBezierPoint(percent, point,
        element.controlPoint1, element.point);

    // Calculate slope by moving back slightly
    CGFloat dx = p.x - QuadBezier(percent * 0.9,
        point.x, element.controlPoint1.x,
        element.point.x);
    CGFloat dy = p.y - QuadBezier(percent * 0.9,
        point.y, element.controlPoint1.y,
        element.point.y);
    if (slope)
        *slope = CGPointMake(dx, dy);
    return p;
}

case kCGPathElementAddCurveToPoint:
{
    // From point to self.point
    CGPoint p = CubicBezierPoint(percent, point,
        element.controlPoint1, element.controlPoint2,
        element.point);

    // Calculate slope by moving back slightly
    CGFloat dx = p.x - CubicBezier(percent * 0.9,

```

```
    point.x, element.controlPoint1.x,
    element.controlPoint2.x, element.point.x);
CGFloat dy = p.y - CubicBezier(percent * 0.9,
    point.y, element.controlPoint1.y,
    element.controlPoint2.y, element.point.y);
if (slope)
    *slope = CGPointMake(dx, dy);
return p;
}
}

// Element could not be interpolated
return NULLPOINT;
}
```

Retrieving Subpaths

Bezier paths may contain one or more subpaths, consisting of either open components, like arcs and lines, or closed components, such as the ovals in Figure 5-5. Each oval in this figure was created as an individual subpath and appended to a composite parent.

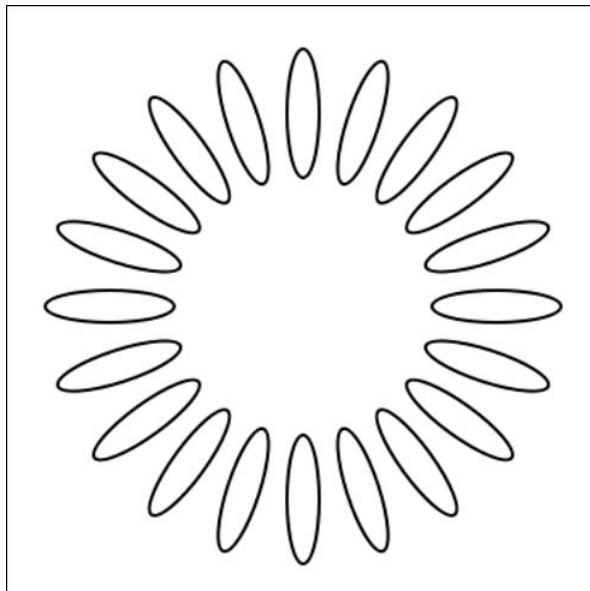


Figure 5-5 This Bezier path consists of multiple oval subpaths.

Many useful drawing operations are based on decomposing a compound path. For example, you might want to color each subpath with a different hue, as is done in Figure 5-6. Iterating through subpaths enables you to apply custom drawing operations based on the order of each path in the parent.

Note

You iterate through the color wheel by using hue component values. Figure 5-6 created these with `colorWithHue:saturation:brightness:alpha:`. Its hue parameter varies from 0.0 (red) to 1.0 (red again, but after passing through all the other possible hues). Each subpath represents a 5% progression along the color wheel.



Figure 5-6 A unique hue fills each subpath in this shape.

Listing 5-8's `subpaths` method decomposes a path. It walks through a path's elements, starting new subpaths whenever it encounters a move operation. Each subpath becomes a standalone `UIBezierPath` instance. The method stores these to a results array and returns that array to the caller.

Listing 5-8 Building a Subpath Array

```
// Return array of component subpaths
- (NSMutableArray *) subpaths
{
    NSMutableArray *results = [NSMutableArray array];
    UIBezierPath *current = nil;
    NSArray *elements = self.elements;

    for (BezierElement *element in elements)
    {
        // Close the subpath and add to the results
        if (element.elementType ==
            kCGPathElementCloseSubpath)
        {
            [current closePath];
            [results addObject:current];
            current = nil;
            continue;
        }

        // Begin new paths on move-to-point
        if (element.elementType ==
            kCGPathElementMoveToPoint)
        {
            if (current)
                [results addObject:current];

            current = [UIBezierPath bezierPath];
            [current moveToPoint:element.point];
            continue;
        }

        if (current)
            [element addToPath:current];
        else
        {
            NSLog(@"Cannot add to nil path: %@", element.stringValue);
            continue;
        }
    }

    if (current)
        [results addObject:current];
```

```
    return results;  
}
```

Inverting Paths

Figure 5-7 shows three takes on one Bezier path. Consisting of a series of ovals rotated and placed into a circle, the leftmost image is simply filled. The middle image inverts that path, filling all areas that lie *outside* the path. The right image offers a third take. It inverts the path but limits that inverse to the path's natural bounds.

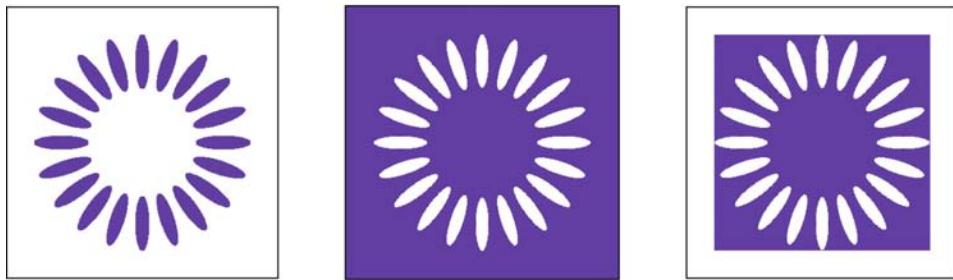


Figure 5-7 Left: Original path. Middle: Inverted path. Right: Inversion within the path bounds.

Each inverse operation leverages the even/odd fill rule. That rule, as you'll remember from Chapter 4, basically says that any point inside the path must pass through an odd number of edges on its way toward infinity. Adding one more rectangle outside the original path flips the even and odd values of every point to produce an inverted selection. The middle image (via the `inverse` method in Listing 5-9) adds an infinite rectangle to the path. This establishes one more boundary for all the path points, flipping their insided-ness to outsided-ness. The even/odd rule uses the new “inside”s, which now fall entirely outside the original path, to determine what to fill.

The right image does the same kind of thing—adding another rectangle so the path insides will flip to the outside—but it does this by using the original path bounds. The `boundedInverse` method in Listing 5-9 is responsible. Any point outside the path's bounds remains “outside” the inverted path and won't be filled.

Inverting paths is a powerful tool for many important drawing operations, such as inner and outer glows and inner and outer shadows. These glows and shadows, which you'll read about later in this chapter, form the basis for many Photoshop-style primitives, which produce outstanding visuals.

Listing 5-9 Inverting Paths

```
- (UIBezierPath *) inverseInRect: (CGRect) rect
{
    UIBezierPath *path = [UIBezierPath bezierPath];
    [path appendPath:self];
    [path appendPath:[UIBezierPath bezierPathWithRect:rect]];
    path.usesEvenOddFillRule = YES;
    return path;
}

- (UIBezierPath *) inverse
{
    return [self inverseInRect:CGRectInfinite];
}

- (UIBezierPath *) boundedInverse
{
    return [self inverseInRect:self.bounds];
}
```

Drawing Shadows

Quartz contexts support shadow drawing as an optional context-specific feature. Painted at an offset from your path that you specify, shadow drawing mimics the effect of a light source on a physical object. Figure 5-8 demonstrates what a shadow might look like when drawn into your context. Shadows can be computationally intense, but they add beautiful details to your interfaces.

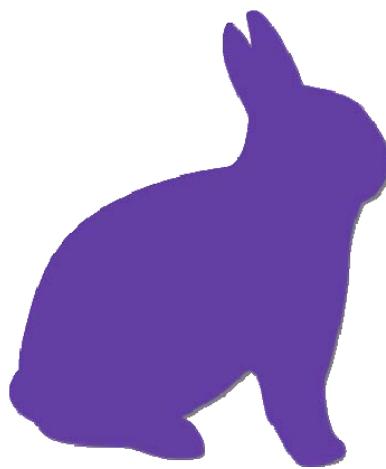


Figure 5-8 You can add shadows to a context.

Shadows, like all other context state changes, affect any subsequent drawing operations. If you want to reset state after applying a shadow, make sure to save and restore the context graphic state (`GState`). Setting the shadow color to clear (`[UIColor clearColor].CGColor`) “disables” shadow application.

Listing 5-10 wraps the `CGContextSetShadowWithColor()` function with an Objective-C color parameter. You specify a color, an offset (size), and a blur amount. The function updates the context state, applying those values. Here are a few points to know about shadows:

- Each shadow is added at an x- and y-offset from any drawing operations. You specify this via `CGSize`.
- The floating-point blur parameter indicates how hard (0) or soft (greater than 0) to make the edge.
- You can skip a color value by calling `CGContextSetShadow()` instead. This function defaults to a semi-translucent black color, with a 0.33 alpha value. Listing 5-10 uses this if the `SetShadow()` function is called with a nil color.

Listing 5-10 Specifying a Context Shadow

```
void SetShadow(UIColor *color,
    CGSize size, CGFloat blur)
{
    CGContextRef context =
        UIGraphicsGetCurrentContext();
```

```

if (context == NULL)
{
    NSLog(@"Error: No context to draw into");
    return;
}

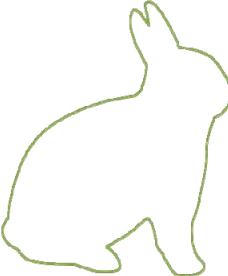
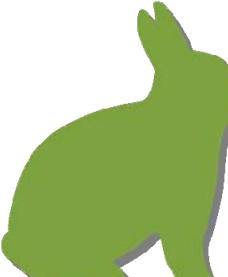
if (color)
    CGContextSetShadowWithColor(context,
        size, blur, color.CGColor);
else
    CGContextSetShadow(context, size, blur);

}

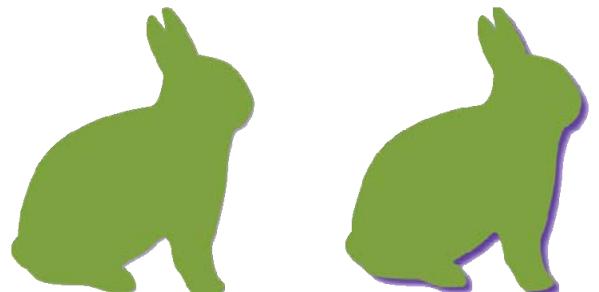
```

Table 5-1 demonstrates the effects that shadow parameters have on drawing.

Table 5-1 Setting Shadows

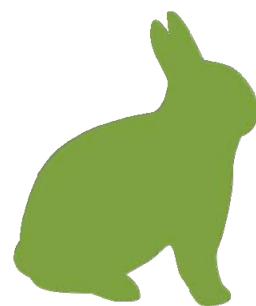
Application	Result
When stored to the context state, shadows apply to any drawing operation. You create shadows both by filling (left) and stroking (right).	 
Increasing the blur radius softens the blur. The left image uses a blur radius of 4. Compare it to the hard blur on the right, which is set at 0.	 

You can apply any shadow color. The examples here are purple. The left image uses an alpha level of 0.5, the right an alpha of 1.0.



Adjusting offsets moves the shadow.

The left image switches the shadow of previous images from (4, 4) to (-4, 4). The shadow moves left but remains under the shape.



The right image uses (-4, -4). The shadow appears to the left and top of the shape.

The Cost of Shadows

There's no getting around the fact that shadows place a high computational load on your drawing. Although visually gorgeous, they're not necessarily a feature you want to use for real-time high-performance interface elements.

Whenever possible, profile your drawing operations during development to get a sense of their cost. Here's a quick-and-dirty approach to use as you build your methods, to track elapsed time:

```
NSDate *start = [NSDate date];
// Perform drawing operations
NSLog(@"%@", [[NSDate date] timeIntervalSinceDate:start]);
```

Drawing an Inner Shadow

Figure 5-9 shows an image created by filling a Bezier path with an added inner shadow. An inner shadow, as the name suggests, adds a shadow drawn within the bounds of the path. Depending on how your brain is processing it at the moment, the shadow either looks as if the edges outside the shape are casting a shadow or, if you can make your mind do a flip-flop, the shape expresses an embossed edge.

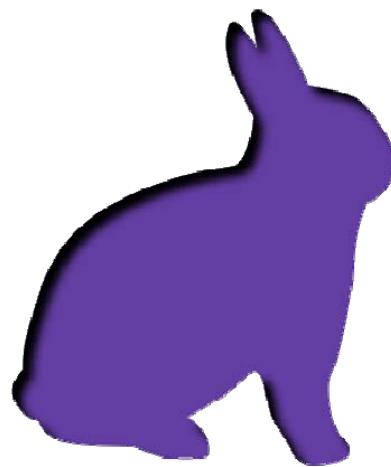


Figure 5-9 You can create a path with an inner shadow.

Figure 5-10 shows the drawing operations you combine to create the inner shadow. The first operation is a matte fill operation. The second operation builds the shadow, which is masked inside the shape.

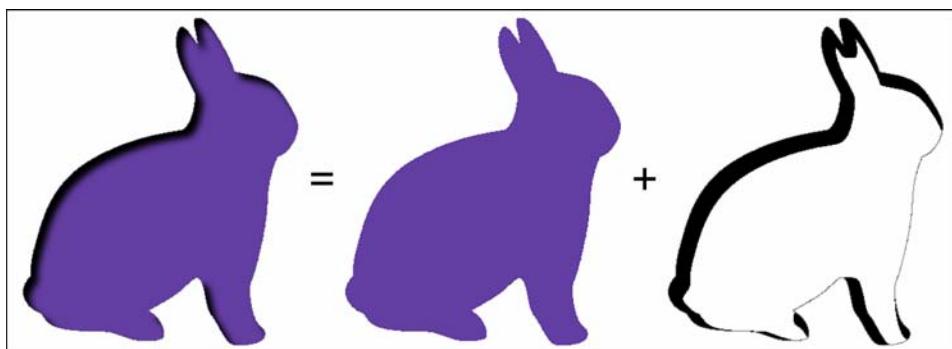


Figure 5-10 Left: Original path. Middle: Inverted path. Right: Inversion within the path bounds.

Building the shadow requires you to invert the path, as you did in Listing 5-9. You allow the context to draw a shadow for that inverted shape, which naturally falls within the remaining portion of the original path. Clipping the context ensures that nothing is drawn outside that path.

Listing 5-11 shows the steps that draw the inner shadow. First you set the context shadow state. This function applies this within a `GState` stack, ensuring that the state restores after the drawing operation. It also clips the drawing area to the path passed as the parameter. This ensures that all drawing takes place within path bounds.

Next, it sets the drawing color. Since all drawing is done outside the clipping region, theoretically any non-clear color would work. In practice, however, you'll encounter slight clipping errors where the path border experiences high curvature. This is a known issue. Using the shadow color avoids visual discontinuities.

Finally, this function fills the inverse of the path with that color. Due to the clip, the only drawing that's created is the shadow, which is shown on the right in Figure 5-10.

Listing 5-11 Drawing an Inner Shadow

```
void DrawInnerShadow(UIBezierPath *path,
    UIColor *color, CGSize size, CGFloat blur)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    // Build shadow
    CGContextSaveGState(context);
    SetShadow(color,
        CGSizeMake(size.width, size.height), blur);

    // Clip to the original path
    [path addClip];

    // Fill the inverted path
    [path.inverse fill:color];

    CGContextRestoreGState(context);
}
```

Listing 5-12 offers another take on the inner shadow. I call this approach the “PaintCode” solution because it was originally inspired by code exported by PixelCut (<http://pixelcut.com>) from their PaintCode application. It’s fussier than the solution from Listing 5-11, but it avoids the edge case that blends tiny bits of the inverted path’s fill color with your drawing at highly inflected curves. It works by tweaking the inverted path ever so

slightly, to clip a tiny bit away from that edge. This results in a cleaner shadow presentation.

Listing 5-12 Drawing a (Better) Inner Shadow

```
- (void) drawInnerShadow: (UIColor *) color
    size: (CGSize) size blur: (CGFloat) radius
{
    if (!color)
        COMPLAIN_AND_BAIL(@"Color cannot be nil", nil);
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
        COMPLAIN_AND_BAIL(@"No context to draw into", nil);

    CGContextSaveGState(context);

    // Originally inspired by the PaintCode guys
    // http://paintcodeapp.com

    // Establish initial offsets
    CGFloat xOffset = size.width;
    CGFloat yOffset = size.height;

    // Adjust the border
    CGRect borderRect =
        CGRectMakeInset(self.bounds, -radius, -radius);
    borderRect =
        CGRectMakeOffset(borderRect, -xOffset, -yOffset);
    CGRect unionRect =
        CGRectMakeUnion(borderRect, self.bounds);
    borderRect = CGRectMakeInset(unionRect, -1.0, -1.0);

    // Tweak the size a tiny bit
    xOffset += round(borderRect.size.width);
    CGSize tweakSize = CGSizeMake(
        xOffset + copysign(0.1, xOffset),
        yOffset + copysign(0.1, yOffset));

    // Set the shadow and clip
    CGContextSetShadowWithColor(context, tweakSize,
        radius, color.CGColor);
    [self addClip];

    // Apply transform
    CGAffineTransform transform =
        CGAffineTransformMakeTranslation(
```

```
    -round(borderRect.size.width), 0);
UIBezierPath *negativePath =
    [self inverseInRect:borderRect];
[negativePath applyTransform:transform];

// Any color would do, use red for testing
[negativePath fill;color];

CGContextRestoreGState(context);
}
```

Embossing a Shape

Applying both a dark inner shadow to the bottom left and a light inner shadow to the top right creates the embossed effect you see in Figure 5-11. This effect uses a soft blur, to create smooth transitions:

```
DrawInnerShadow(path, [[UIColor whiteColor]
    colorWithAlphaComponent:0.3f],
    CGSizeMake(-4, 4), 4);
DrawInnerShadow(path, [[UIColor blackColor]
    colorWithAlphaComponent:0.3f],
    CGSizeMake(4, -4), 4);
```

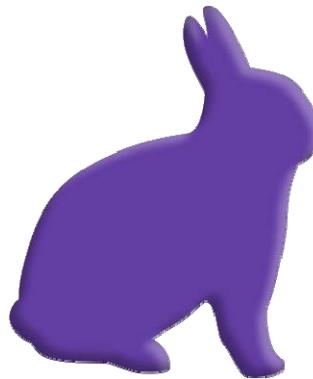


Figure 5-11 You can combine light and dark inner shadows to “emboss” a Bezier path.

You can combine a softer inner shadow with a sharp outer shadow to create a bevel effect for your shape, like the one in Figure 5-12:

```
DrawInnerShadow(bunny, WHITE_LEVEL(0, 0.5),  
    CGSizeMake(-4, 4), 2);  
DrawShadow(bunny, WHITE_LEVEL(0, 0.5),  
    CGSizeMake(2, -2), 0);
```

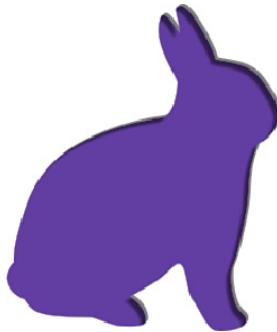


Figure 5-12 You can “bevel” a path by combining an inner shadow and outer shadow with sharp edges.

Drawing Inner and Outer Glows

Unlike shadows, *glows* have no direction. They surround a path on all sides, as in Figure 5-13, where a soft green outer glow highlights the edges of the purple bunny.

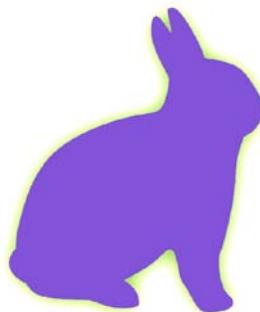


Figure 5-13 A green outer glow is applied outside the path.

With all shadows, there are several ways you can achieve a glow effect; the simplest involves using Quartz shadows, as in Listing 5-13. You draw a shadow directly over the path with an offset of (0, 0). The blur parameter spreads the shadow for you, equally on all sides.

You can use path clipping to draw the shadow outside or inside a path. When clipped inside, the glow is called *inner*. A simple black inner glow creates the 3D feel you see in Figure 5-14.

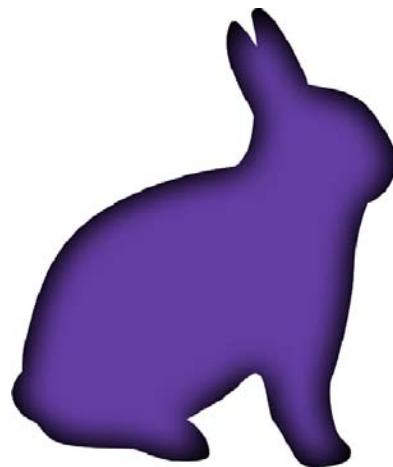


Figure 5-14 Applying an inner glow.

Listing 5-13's results tend to be fairly light in coverage, even when used with fully opaque colors. For this reason, you may want to apply the effect more than once, like this:

```
[path fill:purpleColor];
[path drawInnerGlow:BLACKCOLOR withRadius:20];
[path drawInnerGlow:BLACKCOLOR withRadius:20];
```

Listing 5-13 Drawing Inner and Outer Glows

```
- (void) drawOuterGlow: (UIColor *) fillColor
    withRadius: (CGFloat) radius
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }
```

```
}

CGContextSaveGState(context);
[self.inverse clipToPath];
CGContextSetShadowWithColor(context,
    CGSizeZero, radius, fillColor.CGColor);
[self fill:[UIColor grayColor]];
CGContextRestoreGState(context);
}

- (void) drawInnerGlow: (UIColor *) fillColor
    withRadius: (CGFloat) radius
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    CGContextSaveGState(context);
    [self clipToPath];
    CGContextSetShadowWithColor(context,
        CGSizeZero, radius, fillColor.CGColor);
    [self.inverse fill:[UIColor grayColor]];
    CGContextRestoreGState(context);
}
```

Combining Glows

Inner and outer glows lend impact to your path drawings, especially when combined, as in Figure 5-15. Together, they form a basis for many successful interface element designs by providing dimensional depth. As you'll discover in Chapter 6, adding gradient overlays and underlying textures adds even further realism.

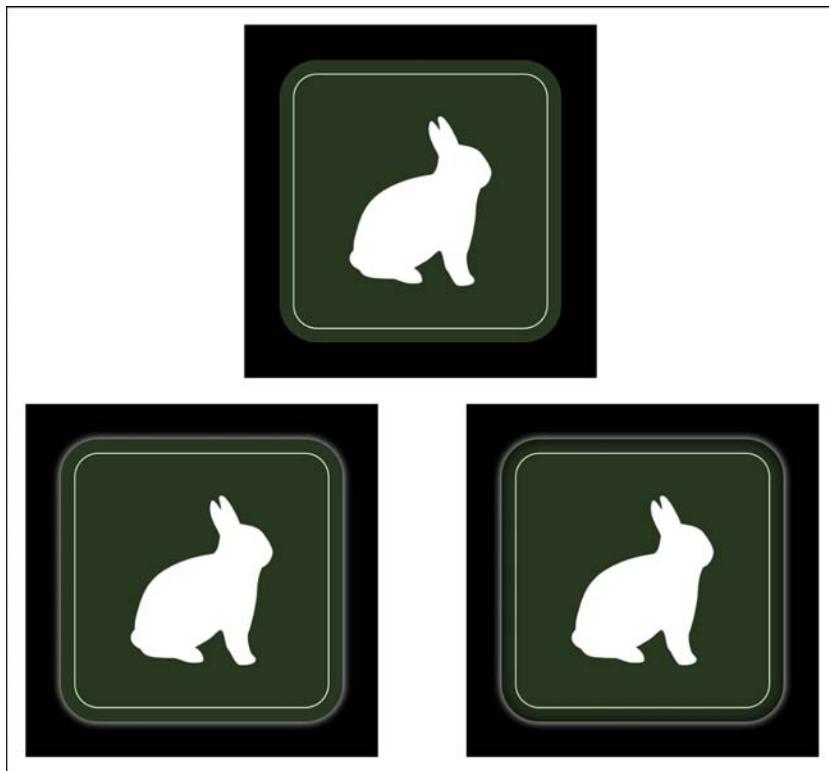


Figure 5-15 Original (top). Added outer glow (bottom left). Added inner and outer glow (bottom right).

Reversing a Path

Unfortunately, reversing a Bezier path—especially a complex one—using `bezierPathByReversingPath` is broken in the iOS 6.x and early release iOS 7 SDKs. A solution depends on decomposing each path into first, its subpaths (see Listing 5-8; each subpath begins with a move operation, and may or may not end with a close operation), and then into the component elements (see Listing 5-2).

In general, you want to reverse a path when you’re applying animation or a drawing effect to the path representation instead of just filling or stroking it. For example, if you’re using a Bezier path to lay out text along the path, reversing the order enables you to change the way drawing proceeds.

Reversing paths turns out to be a bit trickier than you might think. That's because when you reverse items, you have to take close points into account. That means a reversed path often looks like this:

- Move to the destination point of the second-to-last element (the one that comes before the close command).
- Add a line to the first element of the path.
- Reverse each line and curve, using the destination for each item as the start point and the start element as the destination.
- If you had a close command in the original path, apply that close to avoid odd line cap artifacts.

Listing 5-14 shows the lengthy solution. It consists of two methods. The public `reversed` method decomposes the path into subpaths, reversing the order of those subpaths. It then calls the private `reverseSubpath:` method, which flips each individual subpath.

Listing 5-14 Creating a Reversed Bezier Path

```
- (UIBezierPath *) reversed
{
    (UIBezierPath *) subpath
{
    NSArray *elements = subpath.elements;
    NSArray *reversedElements =
        [[elements reverseObjectEnumerator] allObjects];

    UIBezierPath *newPath = [UIBezierPath bezierPath];
    BOOL closesSubpath = NO;

    // Locate the element with the first point
    BezierElement *firstElement;
    for (BezierElement *e in elements)
    {
        if (!POINT_IS_NULL(e.point))
        {
            firstElement = e;
            break;
        }
    }

    // Locate the element with the last point
    BezierElement *lastElement;
    for (BezierElement *e in reversedElements)
    {
        if (!POINT_IS_NULL(e.point))
```

```

    {
        lastElement = e;
        break;
    }
}

// Check for path closure
BezierElement *element = [elements lastObject];
if (element.elementType == kCGPathElementCloseSubpath)
{
    if (firstElement)
        [newPath moveToPoint:firstElement.point];

    if (lastElement)
        [newPath addLineToPoint:lastElement.point];

    closesSubpath = YES;
}
else
{
    [newPath moveToPoint:lastElement.point];
}

// Iterate backwards and reconstruct the path
CFIndex i = 0;
for (BezierElement *element in reversedElements)
{
    i++;
    BezierElement *nextElement = nil;
    BezierElement *workElement = [element copy];

    if (element.elementType == kCGPathElementCloseSubpath)
        continue;

    if (element == firstElement)
    {
        if (closesSubpath) [newPath closePath];
        continue;
    }

    if (i < reversedElements.count)
    {
        nextElement = reversedElements[i];
        if (!POINT_IS_NULL(workElement.controlPoint2))
        {
            CGPoint tmp = workElement.controlPoint1;

```

```

        workElement.controlPoint1 =
            workElement.controlPoint2;
        workElement.controlPoint2 = tmp;
    }
    workElement.point = nextElement.point;
}

if (element.elementType == kCGPathElementMoveToPoint)
    workElement.elementType =
        kCGPathElementAddLineToPoint;

[workElement addToPath:newPath];
}

return newPath;
}

// Reverse the entire path
- (UIBezierPath *) reversed
{
    // [self bezierPathByReversingPath] does not work
    // the way you expect. Radars are filed.

    UIBezierPath *reversed = [UIBezierPath bezierPath];
    NSArray *reversedSubpaths =
        [[self.subpaths reverseObjectEnumerator] allObjects];

    for (UIBezierPath *subpath in reversedSubpaths)
    {
        UIBezierPath *p = [self reverseSubpath:subpath];
        if (p)
            [reversed appendPath:p];
    }
    return reversed;
}

```

Visualizing Path Order

Listing 5-15 enables you to trace a path, displaying the progress of the path from beginning (dark) to end (light). Although I developed this routine specifically for debugging, I've found it handy to use in a variety of circumstances.

For example, the max percentage parameter helps power custom animations. If you eliminate the color phases, the function works as a progress pattern, filling to a point you

specify. If you draw lines instead of dots between iterative points, you can also build an angular version of the entire path with phased color. Playing with a color's white levels (as in Figure 5-16) or hues (as in Figure 5-5) provides a natural progression: from black to white or around the color wheel.

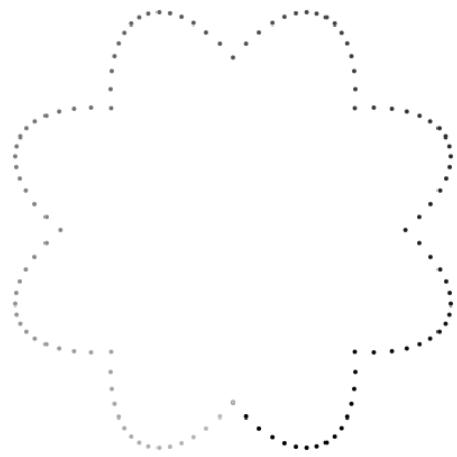


Figure 5-16 You can view the progress of a path from start to end.

Listing 5-15 Tracing a Path Progression

```
void ShowPathProgression(
    UIBezierPath *path, CGFloat maxPercent)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }

    // Bound the percent value
    CGFloat maximumPercent =
        fmaxf(fminf(maxPercent, 1.0f), 0.0f);

    CGContextSaveGState(context);

    // One sample every six points
    CGFloat distance = path.pathLength;
```

```

int samples = distance / 6;

// Change in white level for each sample
float dLevel = 0.75 / (CGFloat) samples;

UIBezierPath *marker;
for (int i = 0; i <= samples * maximumPercent; i++)
{
    // Calculate progress and color
    CGFloat percent = (CGFloat) i / (CGFloat) samples;
    CGPoint point = [path pointAtPercent:percent
        withSlope:NULL];
    UIColor *color =
        [UIColor colorWithWhite:i * dLevel alpha:1];

    // Draw marker
    CGRect r = RectAroundCenter(point, CGSizeMake(2, 2));
    marker = [UIBezierPath bezierPathWithOvalInRect:r];
    [marker fill;color];
}
CGContextRestoreGState(context);
}

```

Summary

This chapter introduces Bezier paths in depth. You saw how to decompose, reconstruct, and paint paths to create a variety of drawing effects. Here are a few final thoughts for this chapter:

- When testing drawing in the simulator, you can actually write from the simulated iOS app to your normal OS X file system. For example, you might want to save a copy of the image you just created in your context by adding a line like this:

```
[UIImagePNGRepresentation(image)
    writeToFile: @"/Users/ericasadun/Desktop/figure.png"
    atomically: YES].
```

- Pay attention to the changes you make in a context. Drawing with shadows *will* change the context state. Embed those updates in `GState` save and restore calls so you won't inadvertently affect other parts of your drawing.
- Be really careful when trying to replicate Photoshop-style effects in your app. Use Instruments to profile your drawing and check for expensive, repeated operations. It's often a lot faster to import images created in other applications and render them into your contexts than to calculate effects directly. Instead of calculating shines and shadows, you can composite them.

- Calculating progress along a path provides a wealth of implementation opportunities. You can draw a path over time, lay out type on that path, animate an object along the path, and more. This chapter shows you a few possibilities, but there's far more you can discover on your own.
- When applying advanced drawing techniques, you might want to increase the size of the drawing space to limit edge effects. This trades off calculation space and speed against better-quality output, which can be scaled back in image views.

6

Drawing Gradients

In iOS, a gradient describes a progression between colors. Used to shade drawings and simulate real-world lighting in computer-generated graphics, gradients are an important component for many drawing tasks and can be leveraged for powerful visual effects. This chapter introduces iOS gradients and demonstrates how to use them to add pizzazz to applications.

Gradients

Gradient progression always involves at least two colors. Colors are associated with starting and ending points that range between 0 and 1. Beyond that, gradients can be as simple or as complex as you'd like to make them. Figure 6-1 demonstrates this range. The top image in Figure 6-1 shows the simplest possible gradient. It goes from white (at 0) to black (at 1). The bottom image shows a gradient constructed from 24 individual hues, with the reference colors deposited at equidistant points. This complex gradient goes from red to orange to yellow to green and so forth.

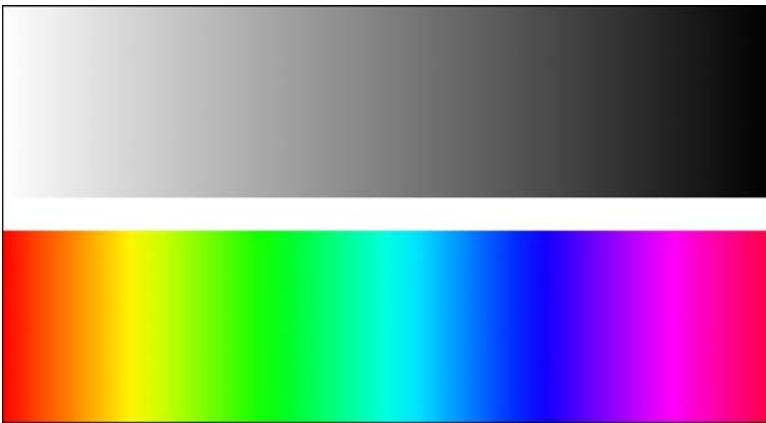


Figure 6-1 A simple white-to-black gradient and a color wheel–inspired gradient.

If you've worked with gradients before, you'll know that you can draw linear and radial output. In case you have not, Figure 6-2 introduces these two styles. On the left, the linear gradient is created by moving from the white (at the bottom) to the black (at the top). Linear gradients draw their colors along an axis that you specify.

In contrast, radial gradients vary the width of their drawing as they progress from start to end. On the right, the radial gradient starts at the white (in the center) and extends out to the black near the edge. In this example, the radius starts at 0 in the middle of the image and ends at the extent of the right edge. As the radius grows, the color darkens, producing the “sphere” look you see here.

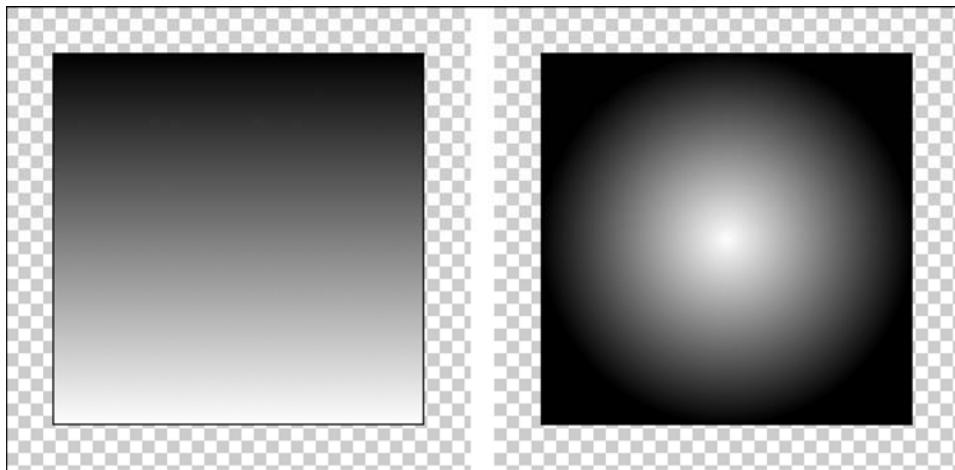


Figure 6-2 Linear (left) and radial (right) gradient drawings.

You may not realize that both images in Figure 6-2 use the same source gradient—the one shown at the top of Figure 6-1. Gradients don’t have a shape, a position, or any geometric properties. They simply describe how colors progress. The way a gradient is drawn depends entirely on you and the Core Graphics function calls you use.

Wrapping the `CGGradientRef` Class

A `CGGradientRef` is a Core Foundation type that stores an arbitrary number of colors and starting points across a range from 0.0 to 1.0. You build the gradient by passing two arrays to it—a set of colors and their locations, as in this example:

```
CGGradientRef CGGradientCreateWithColors(  
    CGColorSpaceRef space,  
    CFArrayRef colors,  
    const CGFloat locations[]  
) ;
```

Before looking any further at the Core Graphics implementation, I want to take a break to introduce an Objective-C workaround that really helps when you use this class.

On the whole, I find it far easier to use gradients through an Objective-C wrapper than to worry about the memory management and mixed C- and Core Foundation-style elements, such as the two arrays used here. As there’s no UIKit-supplied gradient wrapper and no toll-bridged equivalent, I built an Objective-C wrapper. This is where the workaround comes into play.

I'm helped by a little property trick that enables ARC to manage a Core Foundation reference as if it were a normal Cocoa Touch object. Here's how the <http://llvm.org/> website describes this feature:

GCC introduces `__attribute__((NSObject))` on structure pointers to mean "this is an object." This is useful because many low level data structures are declared as opaque structure pointers, e.g. `CFStringRef`, `CFArrrayRef`, etc.

You use this trick to establish a derived type. Here's the type definition I use for Quartz gradients:

```
typedef __attribute__((NSObject)) CGGradientRef GradientObject;
```

This declaration enables you to establish a Core Foundation-powered class property type outside the bounds of toll-free bridging while using ARC memory management. This is important because, as a rule, Quartz classes aren't toll-free bridged into UIKit. You use the derived type to build a property using ARC-style strong management:

```
@property (nonatomic, strong) GradientObject storedGradient;
```

When the `Gradient` instance created in Listing 6-1 is released, so is the underlying `CGGradientRef`. You don't have to build special `dealloc` methods to handle the Core Foundation objects. What you get is a class that handles Core Graphics gradients with Objective-C interfaces. You work with `NSArrays` of `UIColor` colors and `NSNumber` locations.

Caution

This attribute approach requires explicit type definitions, as you saw here. Avoid general use with other language features, like `__typeof`. Consult the LLVM docs for further details and cautions. I feel pretty comfortable using and recommending this approach because Apple engineers introduced me to it.

Listing 6-1 Creating an Objective-C Gradient Class

```
@interface Gradient ()  
@property (nonatomic, strong) GradientObject storedGradient;  
@end  
  
@implementation Gradient  
- (CGGradientRef) gradient  
{  
    // Expose the internal GradientObject property  
    // as a CGGradientRef to the outside world on demand  
    return _storedGradient;  
}
```

```

// Primary entry point for the class. Construct a gradient
// with the supplied colors and locations
+ (instancetype) gradientWithColors: (NSArray *) colorsArray
    locations: (NSArray *) locationArray
{
    // Establish color space
    CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
    if (space == NULL)
    {
        NSLog(@"Error: Unable to create RGB color space");
        return nil;
    }

    // Convert NSNumber *locations array to CGFloat *
    CGFloat locations[locationArray.count];
    for (int i = 0; i < locationArray.count; i++)
        locations[i] = fminf(fmaxf([locationArray[i] floatValue], 0), 1);

    // Convert colors array to (id) CGColorRef
    NSMutableArray *colorRefArray = [NSMutableArray array];
    for (UIColor *color in colorsArray)
        [colorRefArray addObject:(id)color.CGColor];

    // Build the internal gradient
    CGGradientRef gradientRef = CGGradientCreateWithColors(
        space, (_bridge CFArrayRef) colorRefArray, locations);
    CGColorSpaceRelease(space);
    if (gradientRef == NULL)
    {
        NSLog(@"Error: Unable to construct CGGradientRef");
        return nil;
    }

    // Build the wrapper, store the gradient, and return
    Gradient *gradient = [[self alloc] init];
    gradient.storedGradient = gradientRef;
    CGGradientRelease(gradientRef);
    return gradient;
}

+ (instancetype) gradientFrom: (UIColor *) color1
    to: (UIColor *) color2
{
    return [self
        gradientWithColors:@[color1, color2]
        locations:@[@(0.0f), @(1.0f)]];
}

```

```
}
```

```
@end
```

Drawing Gradients

Quartz offers two ways to draw gradients: *linear* and *radial*. The `CGContextDrawLinearGradient()` and `CGContextDrawRadialGradient()` functions paint a gradient between the start and end points you specify. The figures in this section all use an identical purple-to-green gradient, as well as common start and end points. What changes are the functions and parameters used to draw the gradient to the context.

Painting Linear Gradients

Figure 6-3 shows a basic gradient painted by the following linear gradient drawing function:

```
void CGContextDrawLinearGradient(
    CGContextRef context,
    CGGradientRef gradient,
    CGPoint startPoint,
    CGPoint endPoint,
    CGGradientDrawingOptions options
);
```

The green-to-purple gradient is painted from the top left down to the bottom right.

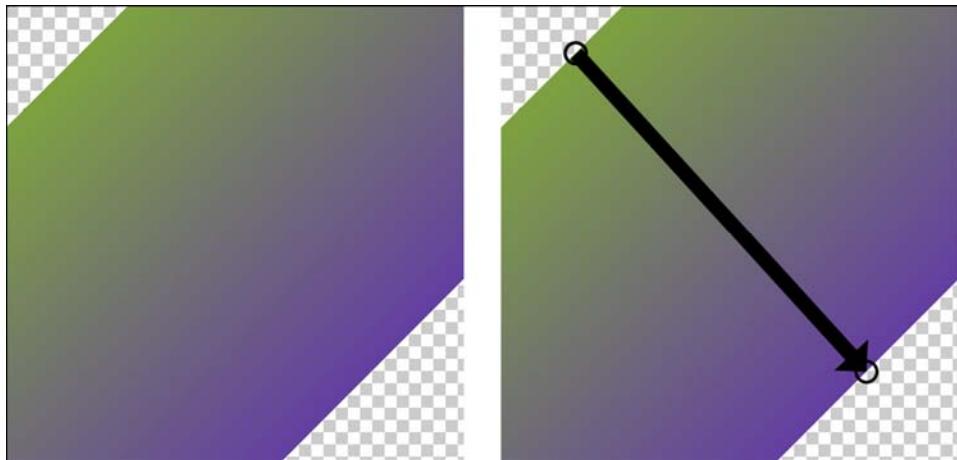


Figure 6-3 A basic linear gradient.

The last parameter of this drawing function is `options`. You use it to specify whether the gradient should extend beyond its start and end point parameters. You supply either 0 (no options, Figure 6-3) or a mask of `kCGGradientDrawsBeforeStartLocation` and `kCGGradientDrawsAfterEndLocation` choices. Figure 6-4 shows these options in use.



Figure 6-4 Using the `kCGGradientDrawsBeforeStartLocation` (left) and `kCGGradientDrawsAfterEndLocation` (middle) mask options enables you to continue drawing beyond the start and end point locations. The right image shows both masks used at once, by OR-ing them together.

Painting Radial Gradients

The radial drawing function adds two parameters beyond the linear function. These parameters specify the radius at the start and the end of the drawing. Figure 6-5 shows the green-to-purple gradient drawn with an initial radius of 20 and a final radius of 50. The left version uses no options, so the drawing stops at the start and end circles. The right version continues drawing both before and after the start and end locations. The underlying circles are clipped by the bounds of the drawing rectangle:

```
void CGContextDrawRadialGradient(  
    CGContextRef context,  
    CGGradientRef gradient,  
    CGPoint startCenter,  
    CGFloat startRadius,  
    CGPoint endCenter,  
    CGFloat endRadius,  
    CGGradientDrawingOptions options  
) ;
```

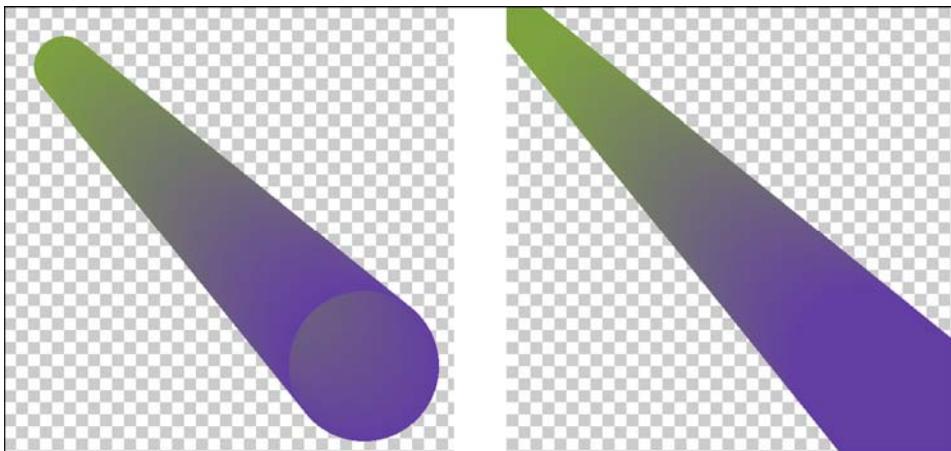


Figure 6-5 The radial gradient on the left is drawn with a starting radius of 20 and an ending radius of 50. The gradient on the right applies flags that continue drawing past the start and end points.

Listing 6-2 shows my Objective-C wrapper for linear and radial gradient drawing operations. These implementations form part of the custom `Gradient` class defined in Listing 6-1. They establish a simple way to paint the Core Graphics gradients embedded in class instances into the active UIKit drawing context.

Listing 6-2 Drawing Gradients

```
// Draw a linear gradient between the two points
- (void) drawFrom: (CGPoint) p1
    toPoint: (CGPoint) p2 style: (int) mask
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }
    CGContextDrawLinearGradient(
        context, self.gradient, p1, p2, mask);
}

// Draw a radial gradient between the two points
- (void) drawRadialFrom:(CGPoint) p1
    toPoint: (CGPoint) p2 radii: (CGPoint) radii
    style: (int) mask
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }
    CGContextDrawRadialGradient(context, self.gradient, p1,
        radii.x, p2, radii.y, mask);
}
```

Building Gradients

Every gradient consists of two collections of values:

- An ordered series of colors
- The locations at which color changes occur

For example, you might define a gradient that proceeds from red to green to blue, at 0.0, 0.5, and 1.0, along the way of the gradient. A gradient interpolates any value between these reference points. One-third of the way along the gradient, at 0.33, the color is approximately 66% of the way from red to green. Or, for example, imagine a simple black-to-white gradient. A medium gray color appears about halfway between the start and end of the drawn gradient.

You can supply any kind of color and location sequence, as long as the colors use either the RGB or grayscale color space. (You can't draw gradients using pattern colors.) Locations fall between 0.0 and 1.0. If you supply values outside this range, the creation function returns `NULL` at runtime.

The most commonly used gradients are white-to-black, white-to-clear, and black-to-clear. Because you apply these with varying alpha levels, I find it handy to define the following macro:

```
#define WHITE_LEVEL(_amt_, _alpha_) \
    [UIColor colorWithWhite:(_amt_) alpha:(_alpha_)]
```

This macro returns a grayscale color at the white and alpha levels you specify. White levels may range from 0 (black) to 1 (white), alpha levels from 0 (clear) to 1 (opaque).

Many developers use a default interpolation between colors to shade their gradients, as in Example 6-1. This example creates a clear-to-black gradient and draws it from points 70% to 100% across the green shape beneath it. You see the result in Figure 6-6, at the top-left corner. Contrast this with the other gradient drawings in Figure 6-6. As you will discover, images were built using gradient *easing*.

Example 6-1 Drawing the Linear Gradient

```
Gradient *gradient = [Gradient
    gradientFrom:WHITE_LEVEL(0, 0) to:WHITE_LEVEL(0, 1)];

// Calculate the points
CGPoint p1 = RectGetPointAtPercents(path.bounds, 0.7, 0.5);
CGPoint p2 = RectGetPointAtPercents(path.bounds, 1.0, 0.5);

// Draw a green background
[path fill:greenColor];

// Draw the gradient across the green background
[path addClip];
[gradient drawFrom:p1 toPoint:p2];
```

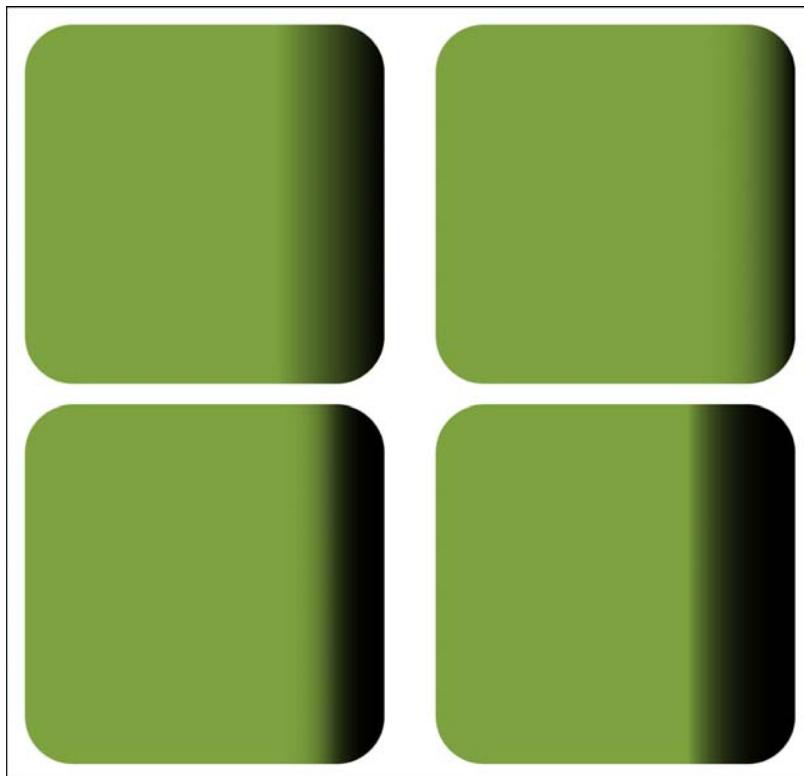


Figure 6-6 Tweaking your gradients affects drawn output. The top-left image shows standard gradient interpolation. The top-right image demonstrates ease-in interpolation. The bottom-left image uses ease-in-out, and the bottom-right image uses ease-out. Each gradient is drawn onto a solid green rounded-rectangle backdrop.

Easing

Easing functions vary the rate at which gradient shading changes. Depending on the function you select, they provide gentler transitions into and out of the gradient. I am most fond of the ease-in and ease-in-out gradients shown, respectively, at the top right and bottom left of Figure 6-6. As you can see, these two approaches avoid abrupt end transitions. Those harsh transitions are created by *perceptual banding*, also called *illusory mach bands*.

Mach bands are an optical illusion first noted by physicist Ernst Mach. Caused by natural pattern processing in our brains, they appear when slightly different shades of gray appear

along boundaries. They happen in computer graphics because drawing stops where an algorithm tells it to stop.

In Figure 6-6, you see these effects at the edges of the gradient's drawing area in the shots at the top left and the bottom right. By using ease-in-out drawing, you stretch the transition between the underlying color and the gradient overlay, avoiding the bands.

Figure 6-7 displays the easing functions for the gradients in Figure 6-6. You see a set of functions: linear (top-left), ease-in (top-right), ease-in-out (bottom left), and ease-out (bottom right). Easing affects the start (for “in”) or end (“out”) of a function to establish more gradual changes. These functions are used with many drawing and animation algorithms.

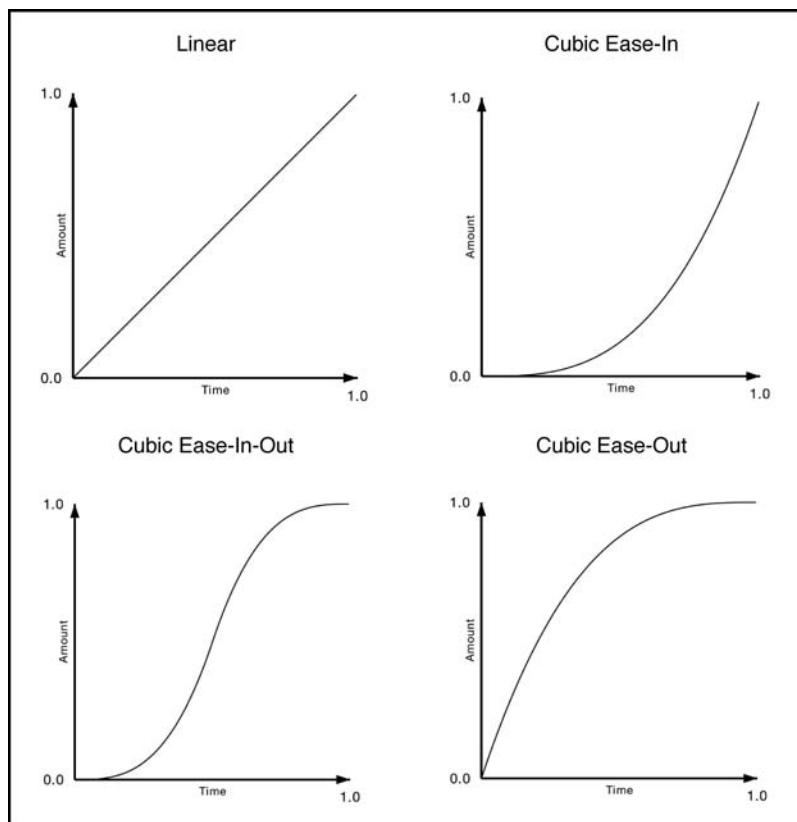


Figure 6-7 Unlike linear interpolation, where values change equally over time, easing functions vary the rate of change at the start (ease-in) and/or the end (ease-out). They provide more gradual transitions that better reflect natural effects in light and movement.

Listing 6-3 defines a `Gradient` class method that builds gradients from functions you supply. You pass a block that accepts an input percentage (the time axis) and returns a value (the amount axis) to apply to the start and end color values. The method interpolates the colors and adds the values to the gradient.

The three standard easing functions use two arguments: elapsed time and an exponent. The exponent you pass determines the type of easing produced. For standard cubic easing, you pass 3 as the second parameter, for quadratic easing, 2. Passing 1 produces a linear function without easing.

You may apply any function you like in the interpolation block. The following snippet builds a gradient using in-out cubic easing:

```
Gradient *gradient = [Gradient gradientUsingInterpolationBlock:  
    ^CGFloat (CGFloat percent) {return EaseInOut(percent, 3);};  
    between: WHITE_LEVEL(0, 0) and: WHITE_LEVEL(0, 1)];
```

Listing 6-3 Applying Functions to Create Custom Gradients

```
typedef CGFloat (^InterpolationBlock)(CGFloat percent);  
  
// Build a custom gradient using the supplied block to  
// interpolate between the start and end colors  
+ (instancetype) gradientUsingInterpolationBlock:  
    (InterpolationBlock) block  
    between: (UIColor *) c1 and: (UIColor *) c2;  
{  
    if (!block)  
    {  
        NSLog(@"Error: No interpolation block");  
        return nil;  
    }  
  
    NSMutableArray *colors = [NSMutableArray array];  
    NSMutableArray *locations = [NSMutableArray array];  
    int numberOfSamples = 24;  
    for (int i = 0; i <= numberOfSamples; i++)  
    {  
        CGFloat amt = (CGFloat) i / (CGFloat) numberOfSamples;  
        CGFloat percentage = fmin(fmax(0.0, block(amt)), 1.0);  
        [colors addObject:  
            InterpolateBetweenColors(c1, c2, percentage)];  
        [locations addObject:@(amt)];  
    }  
  
    return [Gradient gradientWithColors:colors  
        locations:locations];
```

```

}

// Return an interpolated color
UIColor *InterpolateBetweenColors(
    UIColor *c1, UIColor *c2, CGFloat amt)
{
    CGFloat r1, g1, b1, a1;
    CGFloat r2, g2, b2, a2;

    if (CGColorGetNumberOfComponents(c1.CGColor) == 4)
        [c1 getRed:&r1 green:&g1 blue:&b1 alpha:&a1];
    else
    {
        [c1 getWhite:&r1 alpha:&a1];
        g1 = r1; b1 = r1;
    }

    if (CGColorGetNumberOfComponents(c2.CGColor) == 4)
        [c2 getRed:&r2 green:&g2 blue:&b2 alpha:&a2];
    else
    {
        [c2 getWhite:&r2 alpha:&a2];
        g2 = r2; b2 = r2;
    }

    CGFloat r = (r2 * amt) + (r1 * (1.0 - amt));
    CGFloat g = (g2 * amt) + (g1 * (1.0 - amt));
    CGFloat b = (b2 * amt) + (b1 * (1.0 - amt));
    CGFloat a = (a2 * amt) + (a1 * (1.0 - amt));
    return [UIColor colorWithRed:r green:g blue:b alpha:a];
}

#pragma mark - Easing Functions

// Ease only the beginning
CGFloat EaseIn(CGFloat currentTime, int factor)
{
    return powf(currentTime, factor);
}

// Ease only the end
CGFloat EaseOut(CGFloat currentTime, int factor)
{
    return 1 - powf((1 - currentTime), factor);
}

```

```

// Ease both beginning and end
CGFloat EaseInOut(CGFloat currentTime, int factor)
{
    currentTime = currentTime * 2.0;
    if (currentTime < 1)
        return (0.5 * pow(currentTime, factor));
    currentTime -= 2.0;
    if (factor % 2)
        return 0.5 * (pow(currentTime, factor) + 2.0);
    return 0.5 * (2.0 - pow(currentTime, factor));
}

```

Adding Edge Effects

Radial gradients enable you to draw intriguing edge effects in circles. Consider the effect shown in Figure 6-8. It's a gradient expressing a sine wave. However, it's drawn only at the circle's edge, with the center of the path remaining untouched.

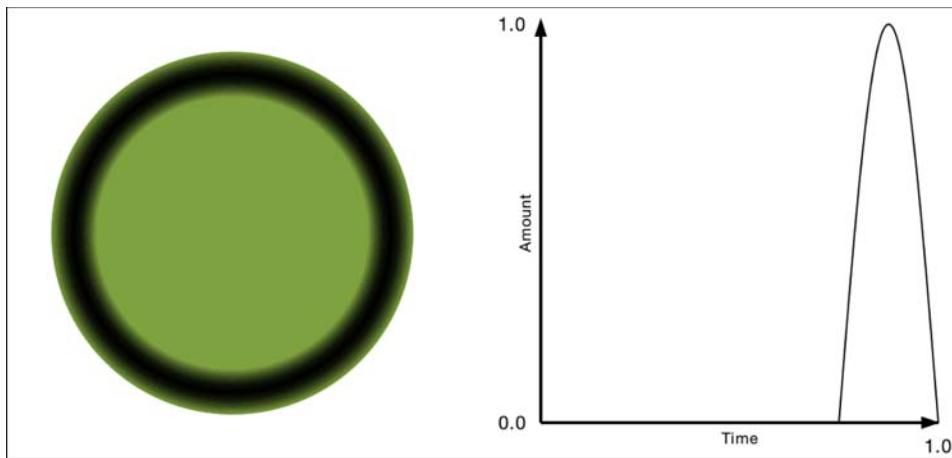


Figure 6-8 A sine-based gradient applied just at the circle's edge.

Example 6-2 uses a nonintuitive approach to accomplish this effect, demonstrating an interesting way to apply gradients. The sine function is compressed to just the last 25% of the gradient. Because the gradient is drawn radially from the center out, a shading effect appears only at the edge.

Example 6-2 Drawing a Delayed Radial Gradient

```
InterpolationBlock block = ^CGFloat (CGFloat percent)
{
    CGFloat skippingPercent = 0.75;
    if (percent < skippingPercent) return 0;
    CGFloat scaled = (percent - skippingPercent) *
        (1 / (1 - skippingPercent));
    return sinf(scaled * M_PI);
};

Gradient *gradient =
[Gradient gradientUsingInterpolationBlock: block
    between: WHITE_LEVEL(0, 0) and: WHITE_LEVEL(0, 1)];

CGContextDrawRadialGradient(UIGraphicsGetCurrentContext(),
    gradient.gradient, center, 0, center, dx, 0);
```

You can use this effect to apply easing just at the edges, as shown in Figure 6-9. The interpolation block compresses the easing function, applying it only after a certain percentage has passed—in this case, 50% of the radial distance:

```
InterpolationBlock block = ^CGFloat (CGFloat percent)
{
    CGFloat skippingPercent = 0.5;
    if (percent < skippingPercent) return 0;
    CGFloat scaled = (percent - skippingPercent) *
        (1 / (1 - skippingPercent));
    return EaseIn(scaled, 3);
};
```

The delayed easing appears in the graph on the right in Figure 6-9. The ascent begins after 0.5. As you can see, that ascent is quite gradual. You don't really see a darkening effect until about 0.75 in.

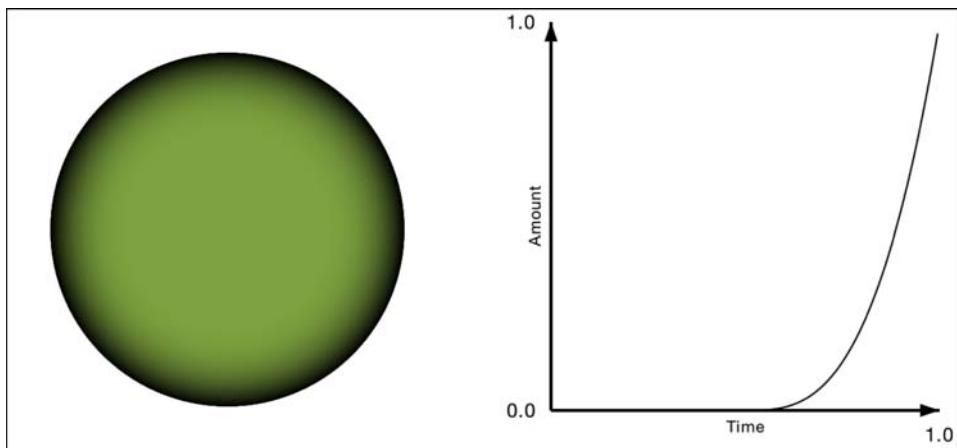


Figure 6-9 This clear-to-black radial gradient is drawn starting from about halfway out from the center. The graph of the interpolation function shows that the easing does not begin until half the radial distance has passed, with most of the color being added after about 75% of the radial distance.

Basic Easing Background

Say that you’re looking for a nice round button effect. The base radial result shown in Figure 6-2 may be too spherical for you, and the delayed effect in Figure 6-9 may be too flat. The easing function, as it turns out, produces a really nice button base, as shown in Figure 6-10.

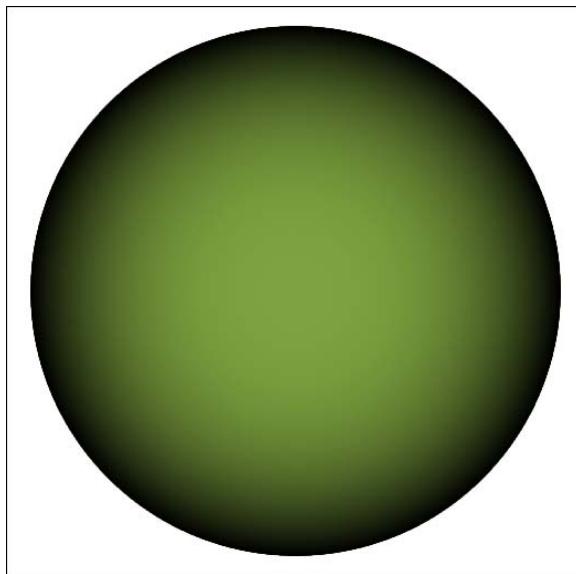


Figure 6-10 This ease-in radial function is applied from the center to the edges to provide an easy basis for round buttons.

The interpolation block is simply as follows:

```
InterpolationBlock block = ^CGFloat (CGFloat percent)
{
    return EaseIn(percent, 3);
};
```

You'll see this approach used again later in this chapter for building a "glowing" center onto a button base.

State and Transparency Layers

Before continuing further into gradients, this chapter needs to step back and cover an important Quartz drawing feature. This feature is used in the examples that follow in this chapter and deserve an explanation.

If you've worked with Photoshop (or similar image-composition and editing apps), you're probably familiar with layers. Layers encapsulate drawing into distinct individual containers. You stack these layers to build complex drawings, and you apply layer effects to add shadows, highlights, and other ornamentation to the contents of each layer. Importantly, these effects apply to entire layers at a time, regardless of the individual drawing operations that created the layer contents.

Quartz offers a similar feature, called *transparency layers*. These layers enable you to combine multiple drawing operations into a single buffer. Figure 6-11 demonstrates why you want to use layers in your apps.

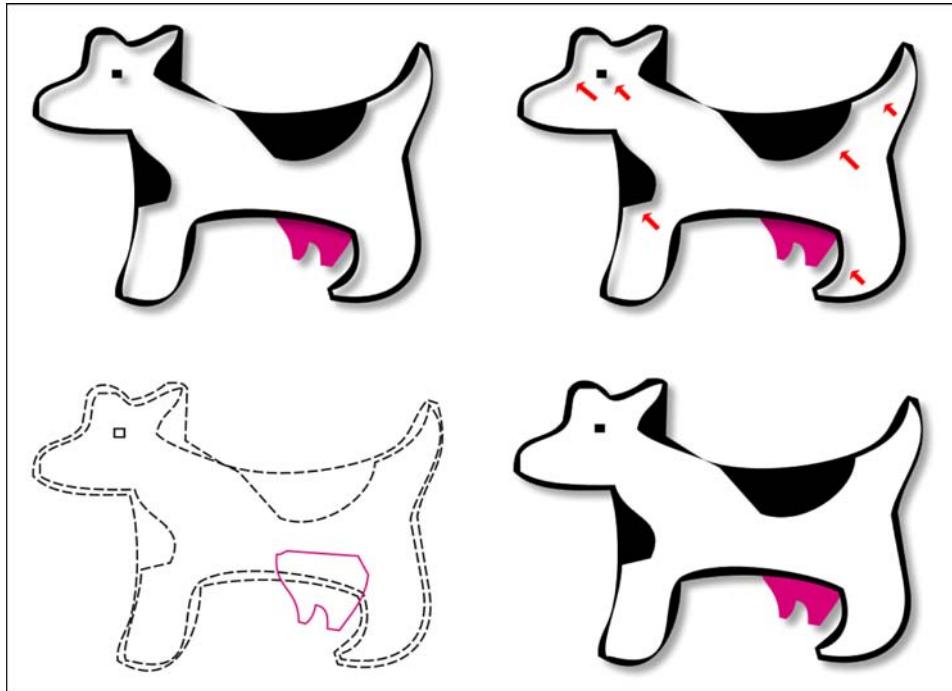


Figure 6-11 Using transparency layers ensures that drawing effects are applied to an entire collection of drawing operations at once rather than to individual drawing requests. Clarus the DogCow is sourced from Apple’s TechNote 31.

This drawing was rendered into a context where shadows were enabled. In the top images, the shadows appear under all portions of the drawing, including the “insides” of the DogCow. That’s because this picture was created using three Bezier fill operations:

- The first operation filled the pink DogCow udder. (Clarus purists, please forgive the heresy. I wanted a more complex shape to work with for this example.)
- The second filled the white background within the figure.
- The third drew the spots, the eye, and the outline on top of that background.

The bottom-left image shows the outlines of the Bezier paths used for these drawing tasks. When these paths are performed as three operations, the context applies shadows to each drawing request. To create a single compound drawing, as in the bottom right of Figure

6-11, you use Quartz transparency layers instead. The shadow is applied only at the edges of the compound drawing, not at the edges of the components.

Transparency layers group drawing requests into a distinct buffer, separate from your drawing context. On starting a layer (by calling `CGContextBeginTransparencyLayer()`), this buffer is initialized with a fully transparent background. Its shadows are disabled, and the global alpha is set to 1. Only after you finish drawing (by calling `CGContextEndTransparencyLayer()`) are the layer's contents rendered to the parent context.

Transparency Blocks

As with most other Quartz and UIKit drawing requests, layer declarations quickly become messy: hard to follow, challenging to read, and difficult to maintain. Consider, instead, Example 6-3, which presents the code that created the final DogCow in Figure 6-11. The block passed to `PushLayerDraw()` ensures that the shadow, which was set before the drawing, applies to the group as a whole.

Example 6-3 Drawing Transparency Layers Using Blocks

```
SetShadow(shadowColor, CGSizeMake(4, 4), 4);
PushLayerDraw:^{
    [udder fill:pinkColor];
    [interior fill:whiteColor];
    [baseMoof fill:blackColor];
});
```

Listing 6-4 presents the `PushLayerDraw()` function. It executes a block of drawing operations within a transparency layer. This approach enables you to group drawings within an easy-to-use block that ensures layer-based rendering.

Listing 6-4 Drawing with Transparency Layers

```
typedef void (^DrawingStateBlock)();

void PushLayerDraw(DrawingStateBlock block)
{
    if (!block) return; // Nothing to do

    CGContextRef context =
        UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
```

```
        return;
    }

CGContextBeginTransparencyLayer(context, NULL);
block();
CGContextEndTransparencyLayer(context);
}
```

The virtues of transparency layers are obvious: They enable you to treat drawing operations as groups. The drawback is that they can be memory hogs due to the extra drawing buffer. You mitigate this by clipping your context *before* working with layers. If you know that your group will be drawing to only a portion of your context, add that clipping before beginning the layer. This forces the layers to draw only to the clipped regions, reducing the buffer sizes and the associated memory overhead.

Be careful, however, with shadows. Add a shadow allowance to your clipping region, as the shadow will be drawn as soon as the transparency layer concludes. As a rule of thumb, you want to allow for the shadow size plus the blur. So for a shadow with an offset of (2, 4) and a blur of 4, add at least (6, 8) points to your clipping region.

State Blocks

Whenever you work with temporary clipping or any other context-specific state, you can make your life a lot easier by using a blocks-based approach, as in Listing 6-5. Similarly to Listing 6-4, this `PushDraw()` function executes a block between calls that save and restore a context's state.

Listing 6-5 Drawing with State Blocks

```
void PushDraw(DrawingStateBlock block)
{
    if (!block) return; // Nothing to do

    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
    {
        NSLog(@"Error: No context to draw to");
        return;
    }
    CGContextSaveGState(context);
    block();
    CGContextRestoreGState(context);
}
```

Example 6-4 uses the functions from Listings 6-4 and 6-5 to show the full sequence used to create the final image in Figure 6-11. It performs a context clip, sets the context shadow, and draws all three Bezier paths as a group. After this block executes, the context returns entirely to its predrawing conditions. Neither the clipping nor the shadow state changes persist beyond this example.

Example 6-4 Using State and Transparency Blocks with Clipping

```
CGRect clipRect = CGRectMakeInset(destinationRect, -8, -8);
PushDraw:^{
    // Clip path to bounds union with shadow allowance
    // to improve drawing performance
    [[UIBezierPath bezierPathWithRect:clipRect] addClip];

    // Set shadow
    SetShadow(shadowColor, CGSizeMake(4, 4), 4);

    // Draw as group
    PushLayerDraw:^{
        [udder fill:pinkColor];
        [interior fill:whiteColor];
        [baseMoof fill:blackColor];
    });
});
```

Flipping Gradients

Gradients naturally emulate light. When inverted, they establish visual hollows. These are areas that would be indented in a physical world to catch inverted light patterns. Drawing gradients first one way and then, after insetting, the other builds the effects you see in Figure 6-12.

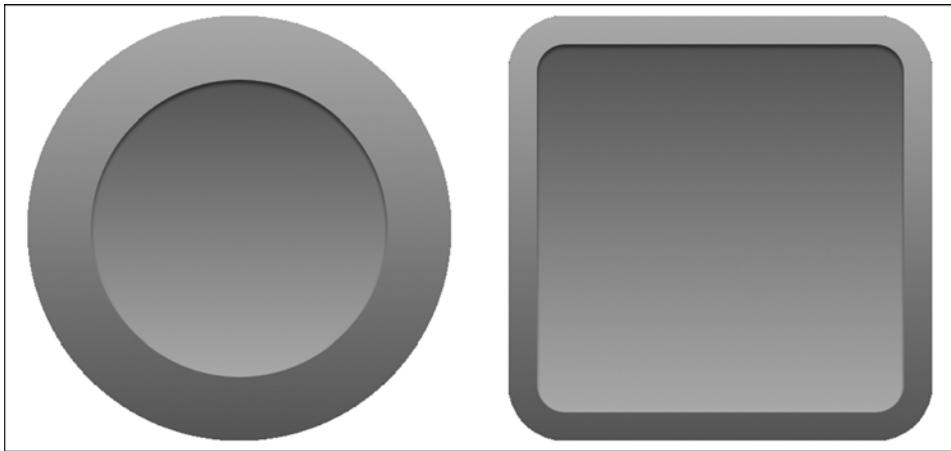


Figure 6-12 Reversing a gradient creates a 3D inset effect.

Example 6-5 shows the code that built the circular sample on the left. It creates a gradient from light gray to dark gray and draws it first from top to bottom in the larger shape. Then it reverses direction, drawing again in the other direction, using the smaller shape.

Example 6-5 adds a finishing touch in drawing a slight black inner shadow (see Chapter 5) at the top of the smaller shape. This shadow emphasizes the point of differentiation between the two drawings but is otherwise completely optional.

Example 6-5 Drawing Gradients in Opposing Directions

```
UIBezierPath *outerPath =
    [UIBezierPath bezierPathWithOvalInRect:outerRect];
UIBezierPath *innerPath =
    [UIBezierPath bezierPathWithOvalInRect:innerRect];

Gradient *gradient =
    [Gradient gradientFrom:WHITE_LEVEL(0.66, 1)
        to:WHITE_LEVEL(0.33, 1)];

PushDraw:^{
    [outerPath addClip];
    [gradient drawTopToBottom:outerRect];
};

PushDraw:^{
    [innerPath addClip];
    [gradient drawBottomToTop:innerRect];
```

```
});  
  
DrawInnerShadow(innerPath, WHITE_LEVEL(0.0, 0.5f),  
    CGSizeMake(0, 2), 2);
```

Mixing Linear and Radial Gradients

There's no reason you can't mix linear and radial effects in your drawings. For example, Example 6-6 draws a blue radial gradient over the base built by Example 6-5. This produces the eye-pleasing glowing button effect you see in Figure 6-13.

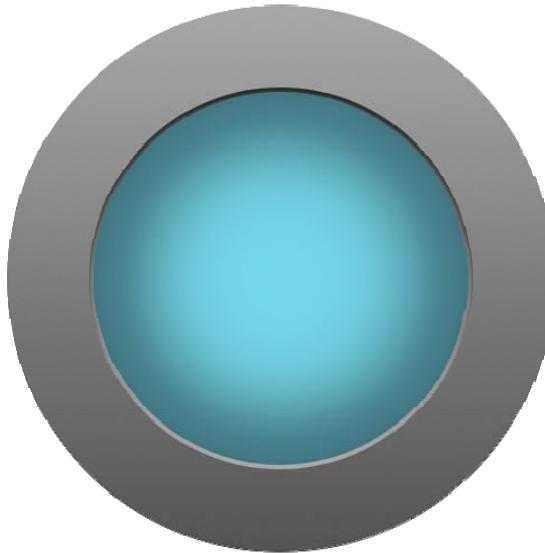


Figure 6-13 Combining radial and linear gradients.

Example 6-6 Drawing a Radial Gradient Using Ease-In-Out

```
CGRect insetRect = CGRectInset(innerRect, 2, 2);  
UIBezierPath *bluePath =  
    [UIBezierPath bezierPathWithOvalInRect:insetRect];  
  
// Produce an ease-in-out gradient, as in Listing 6-5  
Gradient *blueGradient = [Gradient  
    easeInOutGradientBetween:skyColor and:darkSkyColor];
```

```

// Draw the radial gradient
CGPoint center = RectGetCenter(insetRect);
CGPoint topRight = RectGetTopRight(insetRect);
CGFloat width = PointDistanceFromPoint(center, topRight);

PushDraw:^{
    [bluePath addClip];
    CGContextDrawRadialGradient(UIGraphicsGetCurrentContext(),
        blueGradient.gradient, center, 0, center, width, 0);
};


```

Drawing Gradients on Path Edges

I'm often asked how to work with the edge of a path. Usually this is within the context of testing touches on a Bezier path, but sometimes developers want to know how to add special effects just to a path's edge. There's an odd little Core Graphics function called `CGPathCreateCopyByStrokingPath()`. It builds a path with a width you specify, formed around the edge of a given Bezier path.

Listing 6-6 uses this function to clip a path around its normal stroke area. You supply the width; it builds and clips the edge path. Once clipped, you can draw a gradient into your context. This produces the effect you see in Figure 6-14. In the left image, the path is filled with the gradient. In the right image, a dashed pattern stroked onto the original path highlights the way the new path was built.



Figure 6-14 Stroking a path with a gradient by using Quartz. Left: the stroked image. Right: the same image showing the original path.

For touches, you use `CGPathContainsPoint()` to test whether the stroked version of the path contains the touch point.

Listing 6-6 Clipping a Path to Its Stroke

```
- (void) clipToStroke: (NSUInteger)width
{
    CGPathRef pathRef = CGPathCreateCopyByStrokingPath(
        self.CGPath, NULL, width, kCGLineCapButt,
        kCGLineJoinMiter, 4);
    UIBezierPath *clipPath =
        [UIBezierPath bezierPathWithCGPath:pathRef];
    CGPathRelease(pathRef);
    [clipPath addClip];
}
```

Drawing 3D Letters

Listing 6-7 merges the techniques you just read about to build the 3D letter effects shown in Figure 6-15. So far in this chapter, you've read about transparency layers, gradients, and clipping to a path stroke. Alone, each of these tools can be dry and unengaging. Combined, however, they can produce the eye-catching results you see here.



Figure 6-15 Using gradients, transparency layers, and path clipping to create 3D letters.

This drawing consists of letters drawn with a light-to-dark gradient. The path is clipped, and the gradient is drawn top to bottom. Next, a path clipped to the edges adds a reversed dark-to-light gradient trim around each letter. A transparency layer ensures that together, both drawing operations create a single shadow, cast to the bottom right.

Listing 6-7 also adds an inner shadow (see Chapter 5) to the drawing, giving the text a bit of extra shape definition at the bottom of each letter. This produces the extra “height” at the bottom of each letter compared to the gray gradient outline.

Listing 6-7 Drawing Text with a 3D Effect

```
#define COMPLAIN_AND_BAIL(_COMPLAINT_, _ARG_) \
    NSLog(_COMPLAINT_, _ARG_); return;
```

```

// Brightness scaling
UIColor *ScaleColorBrightness(
    UIColor *color, CGFloat amount)
{
    if (!color) return [UIColor blackColor];

    CGFloat h, s, v, a;
    [color getHue:&h saturation:&s brightness:&v alpha:&a];
    CGFloat v1 = fmaxf(fminf(v * amount, 1), 0);
    return [UIColor colorWithHue:h
        saturation:s brightness:v1 alpha:a];
}

void DrawStrokedShadowedShape(UIBezierPath *path,
    UIColor *baseColor, CGRect dest)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (!context)
        COMPLAIN_AND_BAIL(@"No context to draw to", nil);

    PushDraw:^{
        CGContextSetShadow(context, CGSizeMake(4, 4), 4);

        PushLayerDraw:^{
            // Draw letter gradient (to half brightness)
            PushDraw:^{
                Gradient *innerGradient =
                    [Gradient gradientFrom:baseColor
                        to:ScaleColorBrightness(baseColor, 0.5)];
                [path addClip];
                [innerGradient drawTopToBottom:path.bounds];
            }];

            // Add the inner shadow with darker color
            PushDraw:^{
                CGContextSetBlendMode(context, kCGBlendModeMultiply);
                DrawInnerShadow(path, ScaleColorBrightness(
                    baseColor, 0.3), CGSizeMake(0, -2), 2);
            };

            // Stroke with reversed gray gradient
            PushDraw:^{
                [path clipToStroke:6];
                [path.inverse addClip];
            };
        };
    };
}

```

```

        Gradient *grayGradient =
            [Gradient gradientFrom:WHITE_LEVEL(0.0, 1)
                to:WHITE_LEVEL(0.5, 1)];
            [grayGradient drawTopToBottom:dest];
        });
    });
}
}

```

Building Indented Graphics

Listing 6-8 applies a different twist on gradients, as shown in Figure 6-16. This function uses a dark-to-light gradient to produce an “indented” path effect. A pair of shadows—a black inner shadow at the top and a white shadow at the bottom—adds to the illusion. Combined with the gradient and a small bevel, they trick your eye into seeing a “cut away” shape.

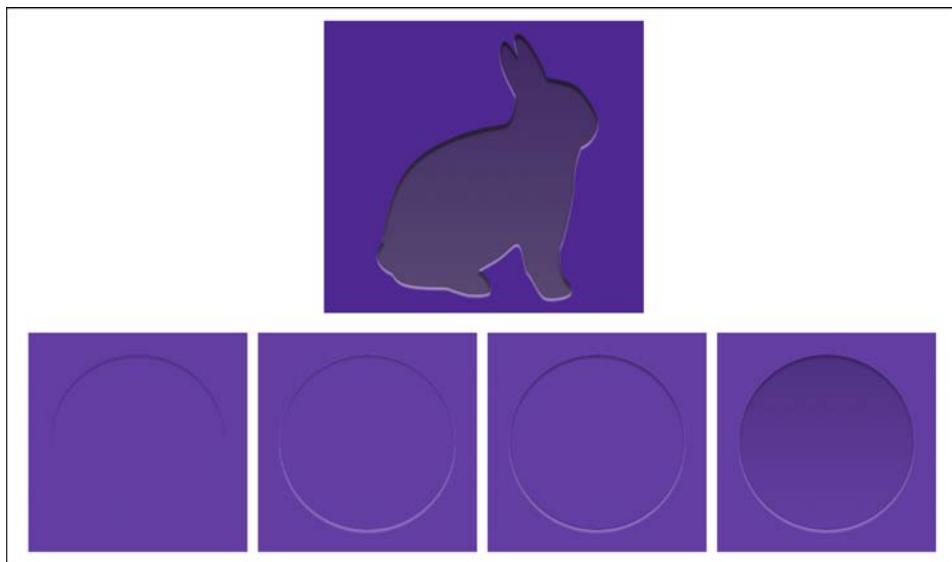


Figure 6-16 Top: Gradients reinforce the “indented” look of this image. Bottom: The progression of indentation, step by step.

Listing 6-8 Indenting Graphics

```

void DrawIndentedPath(UIBezierPath *path,
    UIColor *primary, CGRect rect)

```

```

{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (!context) COMPLAIN_AND_BAIL(@"No context to draw to", nil);

    // Draw the black inner shadow at the top
    PushDraw:^{
        CGContextSetBlendMode(UIGraphicsGetCurrentContext(),
            kCGBlendModeMultiply);
        DrawInnerShadow(path, WHITE_LEVEL(0, 0.4),
            CGSizeMake(0, 2), 1);
    });

    // Draw the white shadow at the bottom
    DrawShadow(path, WHITE_LEVEL(1, 0.5), CGSizeMake(0, 2), 1);

    // Create a bevel effect
    BevelPath(path, WHITE_LEVEL(0, 0.4), 2, 0);

    // Draw a gradient from light (bottom) to dark (top)
    PushDraw:^{
        [path addClip];
        CGContextSetAlpha(UIGraphicsGetCurrentContext(), 0.3);

        UIColor *secondary = ScaleColorBrightness(primary, 0.3);
        Gradient *gradient = [Gradient
            gradientFrom:primary to:secondary];
        [gradient drawBottomToTop:path.bounds];
    });
}

```

Combining Gradients and Texture

Textures expand the way you color objects, providing shading details for visual interest. Take Figure 6-17, for example. In these images, the `kCGBlendModeColor` Quartz blend mode enables you to draw gradients over background images. This mode picks up image texture (luminance values) from the destination context while preserving the hue and saturation of the gradient colors.



Figure 6-17 Blending modes combine the colors from a gradient with the textures of an underlying image.

The top two images display a purple gradient drawn from light (on top) to slightly darker (on the bottom, with a 25% reduction in brightness). The top image showcases the original image source (right) together with the gradient overlay (left). The bottom image applies a rainbow gradient.

Listing 6-9 shows the `DrawGradientOverTexture()` function used to create these images.

Listing 6-9 Transforming Textures with Gradient Overlays

```
void DrawGradientOverTexture(UIBezierPath *path,
    UIImage *texture, Gradient *gradient, CGFloat alpha)
{
    if (!path) COMPLAIN_AND_BAIL(
```

```
    @"Path cannot be nil", nil);
if (!texture) COMPLAIN_AND_BAIL(
    @"Texture cannot be nil", nil);
if (!gradient) COMPLAIN_AND_BAIL(
    @"Gradient cannot be nil", nil);
CGContextRef context = UIGraphicsGetCurrentContext();
if (context == NULL) COMPLAIN_AND_BAIL(
    @"No context to draw into", nil);

CGRect rect = path.bounds;
PushDraw:^{
    CGContextSetAlpha(context, alpha);
    [path addClip];
    PushLayerDraw:^{
        [texture drawInRect:rect];
        CGContextSetBlendMode(
            context, kCGBlendModeColor);
        [gradient drawTopToBottom:rect];
    }];
});
}
}
```

Adding Noise Texture

The example shown in Figure 6-17 used a `kCGBlendModeColor` blend mode to add hues onto texture. At times, you'll want to reverse the process to add texture to color. For this, you use the `kCGBlendModeScreen` blend mode. Figure 6-18 shows what this looks like. At the top, you see a normal fill, created with a solid purple color. The middle image applies and blends a noise pattern, introducing a subtle texture. The bottom image zooms into the resulting texture, to highlight the underlying variation.

This noise-based technique has become quite popular in the App Store for texturizing interface colors. Despite its reputation as a “flat” UI, iOS 7 uses subtle textures in many of its apps, such as Notes and Reminders. You may have to look very closely to see this granularity, but it is there. Noise and other textures provide more satisfying backgrounds that feel organic in nature.

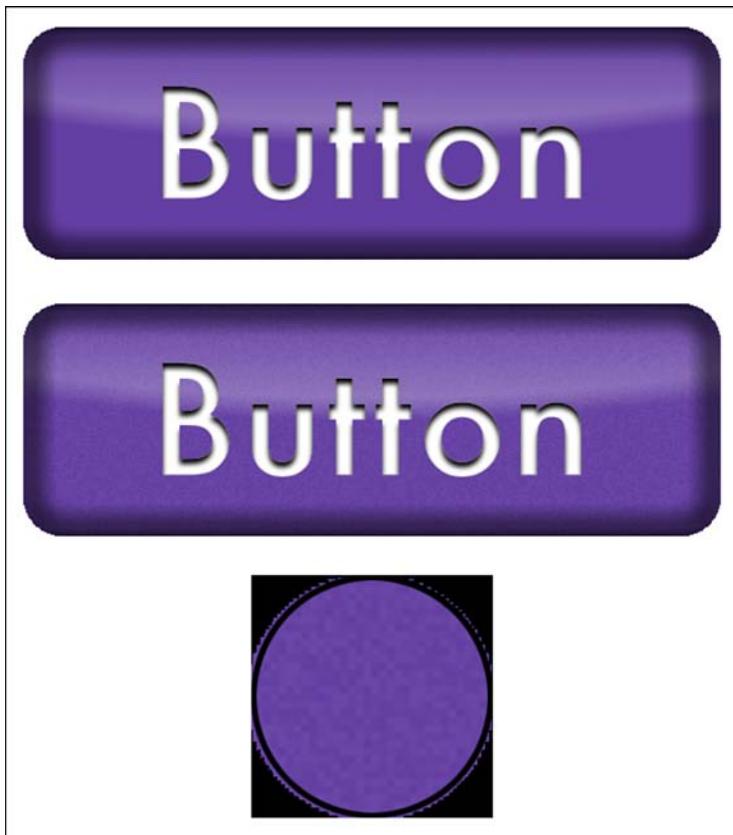


Figure 6-18 Screening enables you to overlay colors with texture.

Screen blends aren't limited to noise, of course. Figure 6-19 shows a dot pattern blended into a fill color.



Figure 6-19 A polka dot pattern adds texture to this background.

Listing 6-10 provides two Bezier path methods that enable you to fill a path. The first applies a color using a blend mode you specify, as shown in Figure 6-19. The second fills a path with a color and then screens in a noise pattern, as in Figure 6-18. A trivial noise color generator completes this listing.

Note

If you're serious about noise, consult the Perlin noise FAQ at <http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html>. Perlin noise offers a function that generates coherent (that is, smoothly changing) noise content.

Listing 6-10 Drawing Textures onto Colors

```
// Apply color using the specified blend mode
- (void) fill: (UIColor *) fillColor
    withMode: (CGBitmapInfo) blendMode
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    PushDraw(^{
        CGContextSetBlendMode(context, blendMode);
        [self fill:fillColor];
    });
}

// Screen noise into the fill
- (void) fillWithNoise: (UIColor *) fillColor
```

```

{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    [self fill:fillColor];
    [self fill:[NoiseColor()
        colorWithAlphaComponent:0.05f]
        withMode:kCGBlendModeScreen];
}

// Generate a noise pattern color
UIColor *NoiseColor()
{
    static UIImage *noise = nil;
    if (noise)
        return [UIColor colorWithPatternImage:noise];

    srand(time(0));

    CGSize size = CGSizeMake(128, 128);
    UIGraphicsBeginImageContextWithOptions(
        size, NO, 0.0);
    for (int j = 0; j < size.height; j++)
        for (int i = 0; i < size.height; i++)
    {
        UIBezierPath *path = [UIBezierPath
            bezierPathWithRect:CGRectMake(i, j, 1, 1)];
        CGFloat level = ((double) random() /
            (double) LONG_MAX);
        [path fill:[UIColor
            colorWithWhite:level alpha:1]];
    }
    noise = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return [UIColor colorWithPatternImage:noise];
}

```

Basic Button Gloss

Many iOS developers will continue to use gradients to add a pseudo-gloss to buttons, even in the brave new flat white age of iOS 7. They understand that custom button use doesn't abrogate the principles of depth and deference.

Glosses create a 3D feel and can be applied to many kinds of views, not just buttons. They play their most important role when working in interfaces created from non-system-supplied items. If your app is primarily based around Apple system controls, go ahead and drop the 3D feel—use borderless buttons and white edge-to-edge design. If not, the techniques you read about here and in the following sections help produce a rich set of alternatives.

Simple glosses consist of linear gradients drawn halfway down the button, followed by a sharp break in color sequence and continued the remaining distance. Figure 6-20 shows a common gloss approach applied to several background colors.

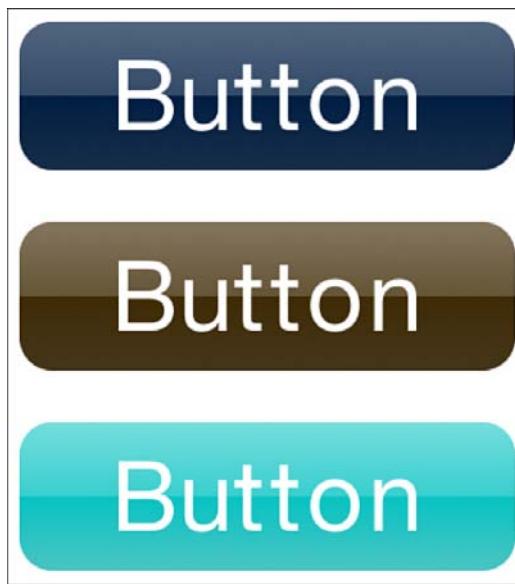


Figure 6-20 Gradient-based linear button gloss.

There are roughly a billion ways to create this kind of highlight gradient, all of which are variations on a fairly similar theme. Listing 6-11 was inspired, long ago, by a Cocoa with Love post by Matt Gallagher (<http://www.cocoawithlove.com/>). None of the mess you see in my listing is his fault, however, but everything nice about the output is reasonably due to his inspiration.

Listing 6-11 Building a Linear Gloss Gradient Overlay

```
+ (instancetype) linearGloss:(UIColor *) color
{
    CGFloat r, g, b, a;
    [color getRed:&r green:&g blue:&b alpha:&a];
```

```

// Calculate top gloss as half the core color luminosity
CGFloat l = (0.299f * r + 0.587f * g + 0.114f * b);
CGFloat gloss = pow(l, 0.2) * 0.5;

// Retrieve color values for the bottom gradient
CGFloat h, s, v;
[color getHue:&h saturation:&s brightness:&v alpha:NULL];
s = fminf(s, 0.2f);

// Rotate the color wheel by 0.6 PI. Dark colors
// move toward magenta, light ones toward yellow
CGFloat rHue = ((h < 0.95) && (h > 0.7)) ? 0.67 : 0.17;
CGFloat phi = rHue * M_PI * 2;
CGFloat theta = h * M_PI;

// Interpolate distance to the reference color
CGFloat dTheta = (theta - phi);
while (dTheta < 0) dTheta += M_PI * 2;
while (dTheta > 2 * M_PI) dTheta -= M_PI_2;
CGFloat factor = 0.7 + 0.3 * cosf(dTheta);

// Build highlight colors by interpolating between
// the source color and the reference color
UIColor *c1 = [UIColor colorWithHue:h * factor +
    (1 - factor) * rHue saturation:s
    brightness:v * factor + (1 - factor) alpha:gloss];
UIColor *c2 = [c1 colorWithAlphaComponent:0];

// Build and return the final gradient
NSArray *colors = @[WHITE_LEVEL(1, gloss),
    WHITE_LEVEL(1, 0.2), c2, c1];
NSArray *locations = @[@(0.0), @(0.5), @(0.5), @(1)];
return [Gradient gradientWithColors:colors
    locations:locations];
}

```

Clipped Gloss

Listing 6-12 offers another common take on button gloss. In this approach (see Figure 6-21), you offset the path outline and then cut away a portion of the gradient overlay. The result is a very sharp transition from the overlay to the button contents.

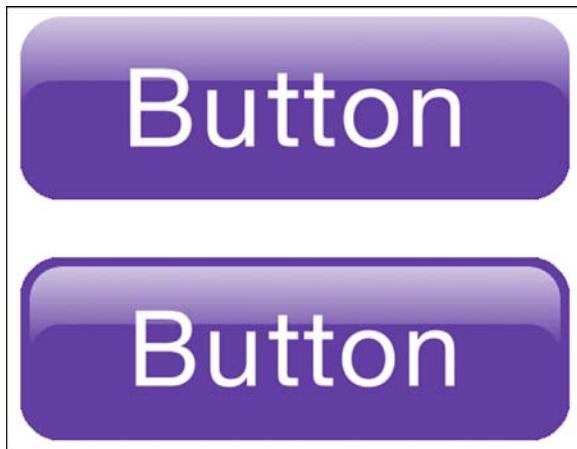


Figure 6-21 Clipped button gloss applied directly at the edges (top) and inset (bottom).

The drawing and clipping take place in a transparency layer. This approach ensures that only the intended overlay survives to deposit its shine onto the original drawing context. The function clears away the remaining material before the transparency layer completes.

Listing 6-12 Building a Clipped Button Overlay

```
void DrawButtonGloss(UIBezierPath *path)
{
    if (!path) COMPLAIN_AND_BAIL(
        @"Path cannot be nil", nil);
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    // Create a simple white to clear gradient
    Gradient *gradient =
        [Gradient gradientFrom:WHITE_LEVEL(1, 1) to:
            WHITE_LEVEL(1, 0)];

    // Copy and offset the path by 35% vertically
    UIBezierPath *offset = [UIBezierPath bezierPath];
    [offset appendPath:path];
    CGRect bounding = path.calculatedBounds;
    OffsetPath(offset, CGSizeMake(0,
        bounding.size.height * 0.35));

    // Draw from just over the path to its middle
```

```
CGPoint p1 = RectGetPointAtPercents(
    bounding, 0.5, -0.2);
CGPoint p2 = RectGetPointAtPercents(
    bounding, 0.5, 0.5);

PushLayerDraw:^{
    PushDraw:^{
        // Draw the overlay inside the path bounds
        [path addClip];
        [gradient drawFrom:p1 toPoint:p2];
    });
}

PushDraw:^{
    // And then clear away the offset area
    [offset addClip];
    CGContextClearRect(context, bounding);
});
});
}
```

Adding Bottom Glows

Bottom glows create the illusion of ambient light being reflected back to your drawing. Figure 6-22 shows the inner- and outer-glow example from Chapter 5 before and after adding this effect.



Figure 6-22 Adding a bottom glow.

Unlike the glows built in Chapter 5, which were created using Quartz shadows, this example uses an ease-in-out gradient. Listing 6-13 creates a function that clips to the path and paints the gradient from the bottom up. You specify what percentage to rise. Figure 6-22 built its bottom glow using 20% white going 40% up the outer rounded-rectangle path.

Listing 6-13 Adding a Bottom Glow

```
void DrawBottomGlow(UIBezierPath *path,
    UIColor *color, CGFloat percent)
{
    if (!path) COMPLAIN_AND_BAIL(
        @"Path cannot be nil", nil);
    if (!color) COMPLAIN_AND_BAIL(
        @"Color cannot be nil", nil);
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    CGRect rect = path.calculatedBounds;
    CGPoint h1 = RectGetPointAtPercents(rect, 0.5f, 1.0f);
    CGPoint h2 = RectGetPointAtPercents(
        rect, 0.5f, 1.0f - percent);

    Gradient *gradient = [Gradient
        easeInOutGradientBetween:color and:
        [color colorWithAlphaComponent:0.0f]];

    PushDraw:^{
        [path addClip];
        [gradient drawFrom:h1 toPoint:h2];
    });
}
```

Building an Elliptical Gradient Overlay

A gradient gloss overlay bounded to an oval path provides another way to introduce light and depth. These are similar to the button glosses you already read about in Listings 6-11 and 6-12. Figure 6-23 shows an example of adding this kind of gloss.



Figure 6-23 Painting an elliptical gloss.

The underlying algorithm is simple, as you see in Listing 6-14: You build an ellipse with the same height as the source path. Stretch its width out to either side. Then move the ellipse up, typically to 0.45 percent from the top edge.

Before drawing, clip the context to the intersection of the original path and the ellipse. Two `addClip` commands handle this request. To finish, paint a clear-to-white gradient from the top of the path to the bottom of the ellipse.

Listing 6-14 Drawing a Top Shine

```

// Build an oval path
UIBezierPath *ovalPath = [UIBezierPath
    bezierPathWithOvalInRect:offset];
Gradient *gradient = [Gradient
    gradientFrom:WHITE_LEVEL(1, 0.0)
    to:WHITE_LEVEL(1, 0.5)];;

PushDraw:^{
    [path addClip];
    [ovalPath addClip];

    // Draw gradient
    CGPoint p1 = RectGetPointAtPercents(rect, 0.5, 0.0);
    CGPoint p2 = RectGetPointAtPercents(
        ovalPath.bounds, 0.5, 1.0);
    [gradient drawFrom:p1 toPoint:p2];
});
}

```

Summary

This chapter introduces gradients, demonstrating powerful ways you can use them in your iOS drawing tasks. Here are some final thoughts about gradients:

- Gradients add a pseudo-dimension to interface objects. They enable you to mimic the effect of light and distance in your code. You'll find a wealth of Photoshop tutorials around the Web that show you how to build 3D effects into 2D drawing. I highly encourage you to seek these out and follow their steps using Quartz and UIKit implementations.
- This chapter uses linear and radial gradients, with and without Quartz shadows, to create effects. Always be careful to balance computational overhead against the visual beauty you intend to create.
- iOS 7 does not mandate “flat design” in your interfaces. Of course, if you build an application primarily around system-supplied components, try to match your app to the Apple design aesthetics. Otherwise, let your design imagination act as your compass to create apps centered on deference, clarity, and depth.

This page intentionally left blank

Masks, Blurs, and Animation

Masking, blurring, and animation represent day-to-day development challenges you experience when drawing. These techniques enable you to add soft edges to your interface, depth-of-field effects, and updates that change over time. This chapter surveys these technologies, introducing solutions for your iOS applications.

Drawing into Images with Blocks

Chapter 6 introduced custom `PushDraw()` and `PushLayerDraw()` functions that combined Quartz drawing with Objective-C blocks for graphics state management and transparency layers. Listing 7-1 riffs off that idea, introducing a new function that returns an image. It uses the same `DrawingStateBlock` type to pass a series of drawing operations within a block, inscribing them into a new image drawing context.

Although I originally built this function to create mask images (as you'll see in Listing 7-2), I have found myself using it in a wide variety of circumstances. For example, it's handy for building content for image views, creating subimages for compositing, building color swatches, and more. Listing 7-1 is used throughout this chapter in a variety of supporting roles and offers a great jumping-off point for many of the tasks you will read about.

Listing 7-1 Creating an Image from a Drawing Block

```
UIImage *DrawIntoImage(  
    CGSize size, DrawingStateBlock block)  
{  
    UIGraphicsBeginImageContextWithOptions(size, NO, 0.0);  
    if (block) block();  
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();  
    UIGraphicsEndImageContext();
```

```
    return image;
}
```

Simple Masking

As you've discovered in earlier chapters, clipping enables you to limit drawing to the area inside a path. Figure 7-1 shows an example of simple mask clipping. In this example, only the portions within the path paint into the drawing context.

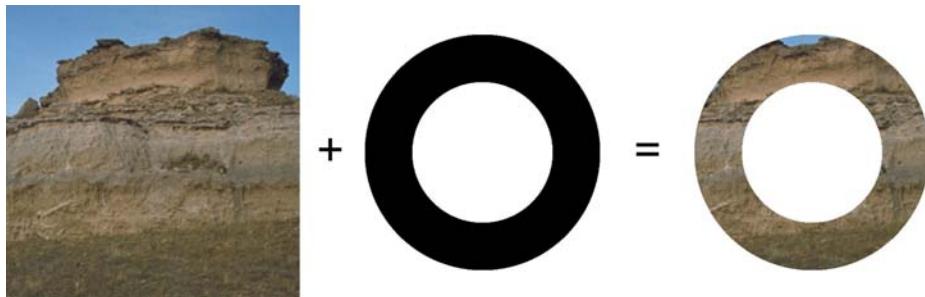


Figure 7-1 Clipping limits drawing within the bounds of a path. *Public domain images courtesy of the National Park Service.*

You achieve this result through either Quartz or UIKit calls. For example, you may call `CGContextClip()` to modify the context to clip to the current path or send the `addClip` method to a `UIBezierPath` instance. I built Figure 7-1 with the `addClip` approach.

Example 7-1 shows the code that built this figure. These commands build a path, apply a clip, and then draw an image to the context

Example 7-1 Basic Clipping

```
// Create the clipping path
UIBezierPath *path =
    [UIBezierPath bezierPathWithOvalInRect:inset];
UIBezierPath *inner = [UIBezierPath
    bezierPathWithOvalInRect:
        RectInsetByPercent(inset, 0.4)];

// The even-odd rule is essential here to establish
// the "inside" of the donut
path.usesEvenOddFillRule = YES;
[path appendPath:inner];
```

```
// Apply the clip
[path addClip];

// Draw the image
UIImage *agate = [UIImage imageNamed:@"agate.jpg"];
[agate drawInRect:targetRect];
```

Complex Masking

The masks in Figure 7-2 produce results far more complex than basic path clipping. Each grayscale mask determines not only where each pixel can or cannot be painted but also to what degree that pixel is painted. As mask elements range from white down to black, their gray levels describe the degree to which a pixel contributes to the final image.

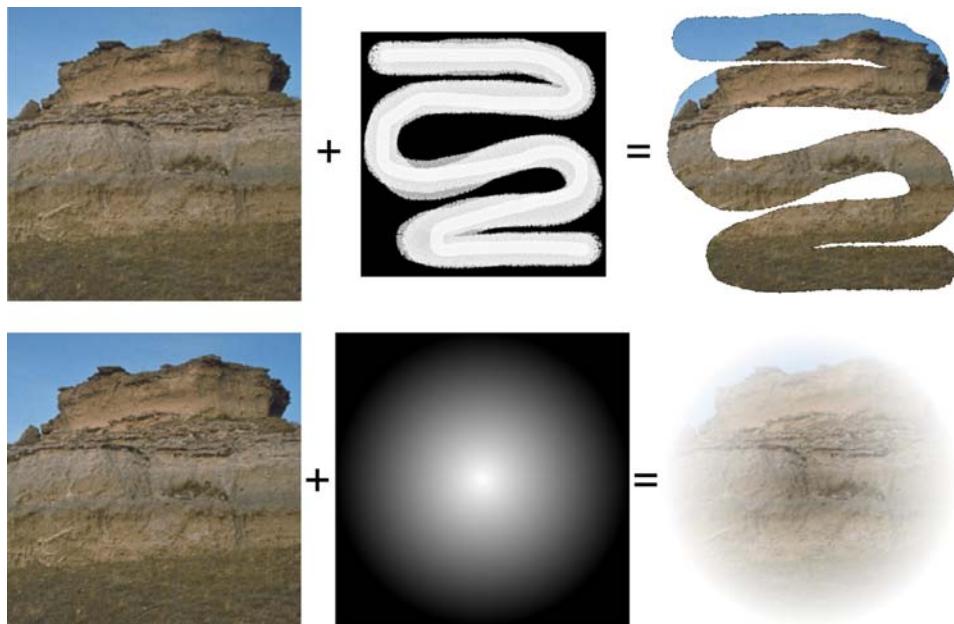


Figure 7-2 The levels in a grayscale mask establish how to paint pixels. *Public domain images courtesy of the National Park Service.*

These masks work by updating context state. Use the `CGContextClipToMask()` function to map a mask you supply into a rectangle within the current context:

```
void CGContextClipToMask (
    CGContextRef c,
    CGRect rect,
```

```
CGImageRef mask  
);
```

For complex drawing, perform your masking within a `GState` block. You can save and restore the context `GState` around masking calls to temporarily apply a mask. This enables you to restore your context back to its unmasked condition for further drawing tasks, as in this example:

```
PushDraw(^{  
    ApplyMaskToContext(mask); // See Listing 7-2  
    [image drawInRect:targetRect];  
});
```

A context mask determines what pixels are drawn and the degree to which they are painted. A black pixel in the mask is fully obscured. No data can pass through. A white pixel allows all data to pass. Gray levels between pure black and pure white apply corresponding alpha values to the painting. For example, a medium gray is painted with 50% alpha. With black-to-white masking, the mask data must use a grayscale source image. (There's also a second approach to this function, which you'll see slightly later in this chapter.)

Listing 7-2 demonstrates how you apply a grayscale mask to the current context. This function starts by converting a mask to a no-alpha device gray color space (see Chapter 3). It calculates the context size so the mask is stretched across the entire context (see Chapter 1).

To apply the mask, you must be in Quartz space, or the mask will be applied upside down. This function flips the context, adds the mask, and then flips the context back to UIKit coordinates to accommodate further drawing commands. This flip-apply-flip sequence applies the mask from top to bottom, just as you drew it.

If you're looking for inspiration, a simple "Photoshop masks" Web search will return a wealth of prebuilt black-to-white masks, ready for use in your iOS applications. Make sure to check individual licensing terms, but you'll find that many masks have been placed in the public domain or offer royalty-free terms of use.

Listing 7-2 Applying a Mask to the Context

```
void ApplyMaskToContext(UIImage *mask)  
{  
    if (!mask)  
        COMPLAIN_AND_BAIL(@"Mask image cannot be nil", nil);  
    CGContextRef context = UIGraphicsGetCurrentContext();  
    if (context == NULL) COMPLAIN_AND_BAIL(  
        @"No context to apply mask to", nil);  
  
    // Ensure that mask is grayscale  
    UIImage *gray = GrayscaleVersionOfImage(mask);  
  
    // Determine the context size
```

```
CGSize size = CGSizeMake(
    CGBitmapContextGetWidth(context),
    CGBitmapContextGetHeight(context));
CGFloat scale = [UIScreen mainScreen].scale;
CGSize contextSize = CGSizeMake(
    size.width / scale, size.height / scale);

// Update the GState for masking
FlipContextVertically(contextSize);
CGContextClipToMask(context,
    SizeMakeRect(contextSize), gray.CGImage);
FlipContextVertically(contextSize);
}
```

Blurring

Blurring is an essential, if computationally expensive, tool for drawing. It enables you to soften transitions at boundaries when masking and build eye-pleasing visuals that create a sense of pseudo-depth. You see an example of this in Figure 7-3. Called “bokeh,” this effect refers to an aesthetic of out-of-focus elements within an image. Blurring emulates the way a photographic lens captures depth of field to create a multidimensional presentation.



Figure 7-3 Blurring builds complex and interesting depth effects.

Although blurring is a part of many drawing algorithms and is strongly featured in the iOS 7 UI, its implementation lies outside the Core Graphics and UIKit APIs. At the time this book was being written, Apple had not released APIs for their custom iOS 7 blurring. Apple engineers suggest using image-processing solutions from Core Image and Accelerate for third-party development.

Listing 7-3 uses a Core Image approach. It's one that originally debuted with iOS 6. This implementation is simple, taking just a few lines of code, and acceptable in speed and overhead.

Acceptable is, of course, a relative term. I encourage you to time your device-based drawing tasks to make sure they aren't overloading your GUI. (Remember that most drawing is thread safe!) Blurring is a particularly expensive operation as these things go. I've found that Core Image and Accelerate solutions tend to run with the same overhead on-device. Core Image is slightly easier to read; that's the one I've included here.

Besides using performance-monitoring tools, you can also use simple timing checks in your code. Store the current date before drawing and examine the elapsed time interval after the drawing concludes. Here's an example:

```
NSDate *date = [NSDate date];
// Perform drawing task here
NSLog(@"%@", @"Elapsed time: %f",
        [[NSDate date] timeIntervalSinceDate:date]);
```

Remember that most drawing *is* thread safe. Whenever possible, move your blurring routines out of the main thread. Store results for re-use, whether in memory or cached locally to the sandbox.

Note

The blurred output of the Gaussian filter is larger than the input image to accommodate blurring on all sides. The Crop filter in Listing 7-3 restores the original dimensions.

Listing 7-3 Core Image Blurring

```
UIImage *GaussianBlurImage(
    UIImage *image, NSInteger radius)
{
    if (!image) COMPLAIN_AND_BAIL_NIL(
        @"Mask cannot be nil", nil);

    // Create Core Image blur filter
    CIFilter *blurFilter =
        [CIFilter filterWithName:@"CIGaussianBlur"];
    [blurFilter setValue:@(radius) forKey:@"inputRadius"];
```

```
// Pass the source image as the input
[blurFilter setValue:[CIIImage imageWithCGImage:
    image.CGImage] forKey:@"inputImage"];

CIFilter *crop =
    [CIFilter filterWithName: @"CICrop"];
[crop setDefaults];
[crop setValue:blurFilter.outputImage
    forKey:@"inputImage"];

// Apply crop
CGFloat scale = [[UIScreen mainScreen] scale];
CGFloat w = image.size.width * scale;
CGFloat h = image.size.height * scale;
CIVector *v = [CIVector vectorWithX:0 Y:0 Z:w W:h];
[crop setValue:v forKey:@"inputRectangle"];

CGImageRef cgImageRef =
    [[[CIContext contextWithOptions:nil]
        createCGImage:crop.outputImage
        fromRect:crop.outputImage.extent];

// Render the cropped, blurred results
UIGraphicsBeginImageContextWithOptions(
    image.size, NO, 0.0);

// Flip for Quartz drawing
FlipContextVertically(image.size);

// Draw the image
CGContextDrawImage(UIGraphicsGetCurrentContext(),
    SizeMakeRect(image.size), cgImageRef);

// Retrieve the final image
UIImage *blurred =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

return blurred;
}
```

Blurred Drawing Blocks

Listing 7-4 returns once again to Objective-C blocks that encapsulate a series of painting commands. In this case, the solution blurs those drawing operations and paints them into the current context.

To accomplish this, the function must emulate a transparency layer. It cannot use a transparency layer directly as there's no way to intercept that material, blur it, and then pass it on directly to the context. Instead, the function draws its block into a new image, using `DrawIntoImage()` (see Listing 7-1), blurs it (using Listing 7-3), and then draws the result to the active context.

You see the result of Listing 7-4 in Figure 7-3. This image consists of two requests to draw random circles. The first is applied through a blurred block and the second without:

```
DrawAndBlur(4, ^{[self drawRandomCircles:20
    withHue:targetColor into:targetRect];});
[self drawRandomCircles:20
    withHue:targetColor into:targetRect];
```

Listing 7-4 Applying a Blur to a Drawing Block

```
// Return the current context size
CGSize GetUIKitContextSize()
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) return CGSizeZero;

    CGSize size = CGSizeMake(
        CGBitmapContextGetWidth(context),
        CGBitmapContextGetHeight(context));
    CGFloat scale = [UIScreen mainScreen].scale;
    return CGSizeMake(size.width / scale,
        size.height / scale);
}

// Draw blurred block
void DrawAndBlur(CGFloat radius, DrawingStateBlock block)
{
    if (!block) return; // Nothing to do

    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    // Draw and blur the image
    UIImage *baseImage = DrawIntoImage(
```

```
GetUIKitContentSize(), block);  
UIImage *blurred = GaussianBlurImage(baseImage, radius);  
  
// Draw the results  
[blurred drawAtPoint:CGPointZero];  
}
```

Blurred Masks

When you blur masks, you create softer edges for drawing. Figure 7-4 shows the result of painting an image using the normal outlines of a rounded-rectangle Bezier path and one that's been blurred (see Example 7-2). In the top image, the path was filled but not blurred. In the bottom image, the `DrawAndBlur()` request softens the edges of the filled path.

Softened edges enable graphics to smoothly blend into each other onscreen. This technique is also called *feathering*. In feathering, edge masks are softened to create a smoother transition between a drawn image and its background.

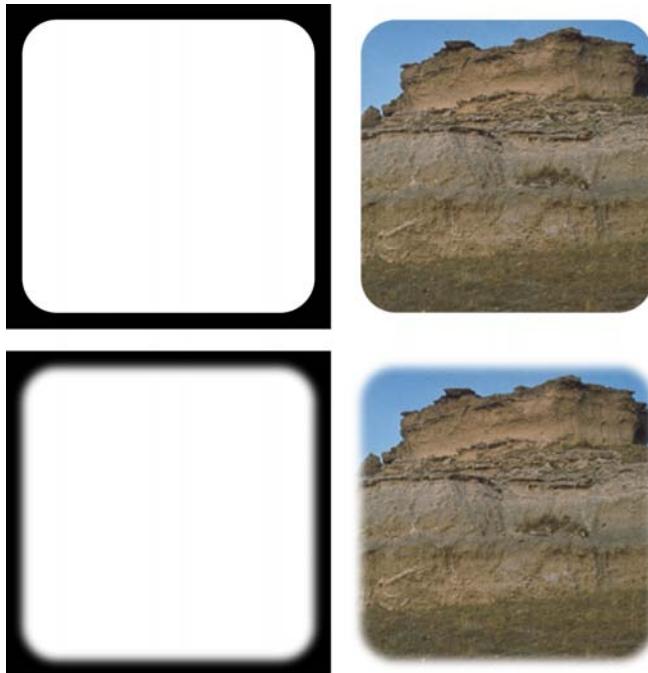


Figure 7-4 Blurring a mask creates softer edges. *Public domain images courtesy of the National Park Service.*

Example 7-2 Drawing to a Blurred Mask

```
UIBezierPath *path = [UIBezierPath  
    bezierPathWithRoundedRect:inset cornerRadius:32];  
  
UIImage *mask = DrawIntoImage(targetRect.size, ^{  
    FillRect(targetRect, [UIColor blackColor]);  
    DrawAndBlur(8, ^{[path fill:[UIColor whiteColor]}]);}); // blurred  
    // [path fill:[UIColor whiteColor]]; // non-blurred  
});  
  
ApplyMaskToContext(mask);  
[agate drawInRect:targetRect];
```

Blurred Spotlights

Drawing “light” into contexts is another common use case for blurring. Your goal in this situation is to lighten pixels, optionally add color, and blend the light without obscuring items already drawn to the context. Figure 7-5 shows several approaches. As you can see, the differences between them are quite subtle.

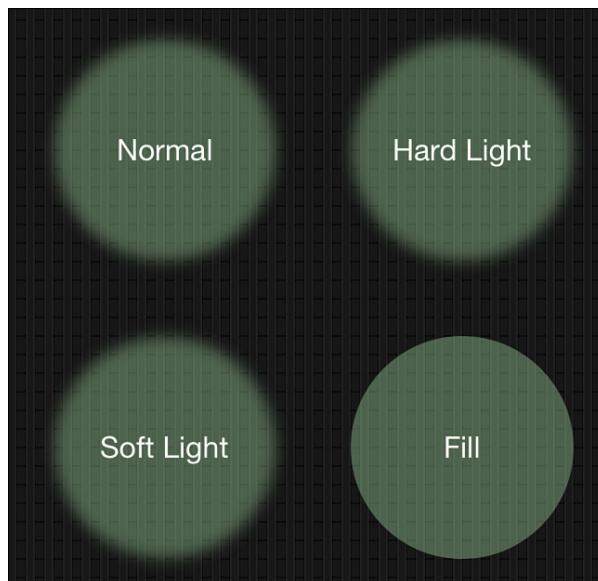


Figure 7-5 Blurring emulates light being shined on a background surface.

All four examples in Figure 7-5 consist of a circular path filled with a light, translucent green. Each sample, except the bottom-right circle, was blurred using the following code:

```
path = [UIBezierPath bezierPathWithOvalInRect:rect];
PushDraw:^{
    CGContextSetBlendMode(UIGraphicsGetCurrentContext(),
        blendMode);
    DrawAndBlur(8, ^{[path fill:spotlightColor];});
}];
```

The top-left example uses a normal blending mode, the top-right example uses a hard light mode, and the bottom-left example uses a soft light mode. Here are a few things to notice:

- The `kCGBlendModeHardLight` sample at the top right produces the subtlest lighting, adding the simplest highlights to the original background.
- The `kCGBlendModeSoftLight` sample at the bottom left is the most diffuse, with brighter highlighting.
- The `kCGBlendModeNormal` sample at the top left falls between these two. The center of the light field actually matches the sample at the bottom right—the one without blurring, which was also drawn using normal blending.

Drawing Reflections

When drawing reflections, you paint an inverted image that gently fades away. Figure 7-6 demonstrates this common technique. I've added a slight vertical gap to highlight where the original image ends and the reflected image begins. Most images that draw reflections use this gap to emulate a difference in elevation between the source image and a reflecting surface below it.

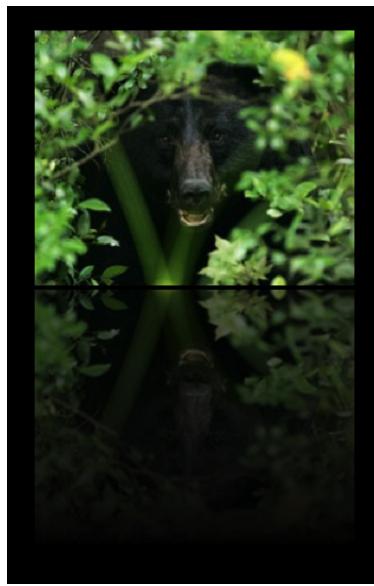


Figure 7-6 Gradients enable you to fade reflections. *Public domain images courtesy of the National Park Service.*

Listing 7-5 shows the function that built this flipped mirror. There are several things to note in this implementation:

- The context is flipped vertically at the point at which the reflection starts. This enables the reflection to draw in reverse, starting at the bottom, near the bushes, and moving up past the head of the bear.
- Unlike previous mask examples, Listing 7-5 uses a rectangle argument to limit the mask and the image drawing. This enables you to draw a reflection into a rectangle within a larger context.
- The `CGContextClipToMask()` function is applied slightly differently than in Listing 7-2. Instead of passing a grayscale image mask to the third parameter, this function passes a normal RGB image with an alpha channel. When used in this fashion, the image acts as an alpha mask. Alpha levels from the image determine what portions of the clipping area are affected by new updates. In this example, the drawn inverted image fades away from top to bottom.

Listing 7-5 Building a Reflected Image

```
// Draw an image into the target rectangle
// inverted and masked to a gradient
void DrawGradientMaskedReflection(
```

```

UIImage *image, CGRect rect)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
        COMPLAIN_AND_BAIL(
            @"No context to draw into", nil);

    // Build gradient
    UIImage *gradient = GradientImage(rect.size,
        WHITE_LEVEL(1.0, 0.5), WHITE_LEVEL(1.0, 0.0));

    PushDraw:^{
        // Flip the context vertically with respect
        // to the origin of the target rectangle
        CGContextTranslateCTM(context, 0, rect.origin.y);
        FlipContextVertically(rect.size);
        CGContextTranslateCTM(context, 0, -rect.origin.y);

        // Add clipping and draw
        CGContextClipToMask(context, rect, gradient.CGImage);
        [image drawInRect:rect];
    });
}

```

Although reflections provide an intriguing example of context clipping, they’re a feature you don’t always need to apply in applications. That’s because the `CAResuplicatorLayer` class and layer masks accomplish this too, plus they provide live updates—so if a view’s contents change, so do the reflections. Although you *can* do it all in Quartz, there are sometimes good reasons why you shouldn’t. Reflections provide one good example of that rule.

You should draw reflections in Quartz when you’re focused on images rather than views. Reflections often form part of a drawing sequence rather than the end product.

Creating Display Links for Drawing

Views may need to change their content over time. They might transition from one image to another, or provide a series of visual updates to indicate application state. Animation meets drawing through a special timing class.

The `CADisplayLink` class provides a timer object for view animation. It fires a refresh clock that’s synced to a display’s refresh rate. This enables you to redraw views on a clock. You can use this clock to produce Quartz-based animation effects such as marching ants or

to add Core Image–based transitions to your interfaces. Display links are part of the QuartzCore framework. You create these timers and associate them with run loops.

Although you can use an `NSTimer` to achieve similar results, using a display link frees you from trying to guess the ideal refresh interval. What’s more, a display link offers better guarantees about the accuracy of the timer (that it will fire on time). Apple writes in the documentation:

The actual time at which the timer fires potentially can be a significant period of time after the scheduled firing time.

Example 7-3 shows how you might create a display link. You should use common modes (`NSRunLoopCommonModes`) for the least latency. In this example, the target is a view, and the fired selector is `setNeedsDisplay`, a system-supplied `UIView` method. When triggered, this target–selector pair tells the system to mark that view’s entire bounds as dirty and request a `drawRect:` redraw on the next drawing cycle. The `drawRect:` method manually draws the content of a custom view using Quartz and iOS drawing APIs.

Example 7-3 Creating a Display Link

```
CADisplayLink *link = [CADisplayLink
    displayLinkWithTarget:view
    selector:@selector(setNeedsDisplay)];
[link addToRunLoop:[NSRunLoop mainRunLoop]
    forMode:NSRunLoopCommonModes];
```

A display link’s frame interval is the property that controls its refresh rate. This normally defaults to 1. At 1, the display link notifies the target each time the link timer fires. This results in updates that match the display’s refresh rate. To adjust this, change the display link’s integer `frameInterval` property. Higher numbers slow down the refresh rate. Setting it to 2 halves your frame rate, and so forth:

```
link.frameInterval = 2;
```

Assuming no processing bottlenecks, a well-behaved system runs at 60 frames per second (fps). You test the refresh rate using the Core Animation profiler in Instruments (see Figure 7-7) while running your app on a device with a frame interval set to 1. This way, you can get an idea of how much burden you’re placing on your app while running animated drawing tasks. If you see your refresh rate drop to, for example, 12 fps or 3 fps or worse, you need to seriously rethink how you’re performing your drawing tasks.

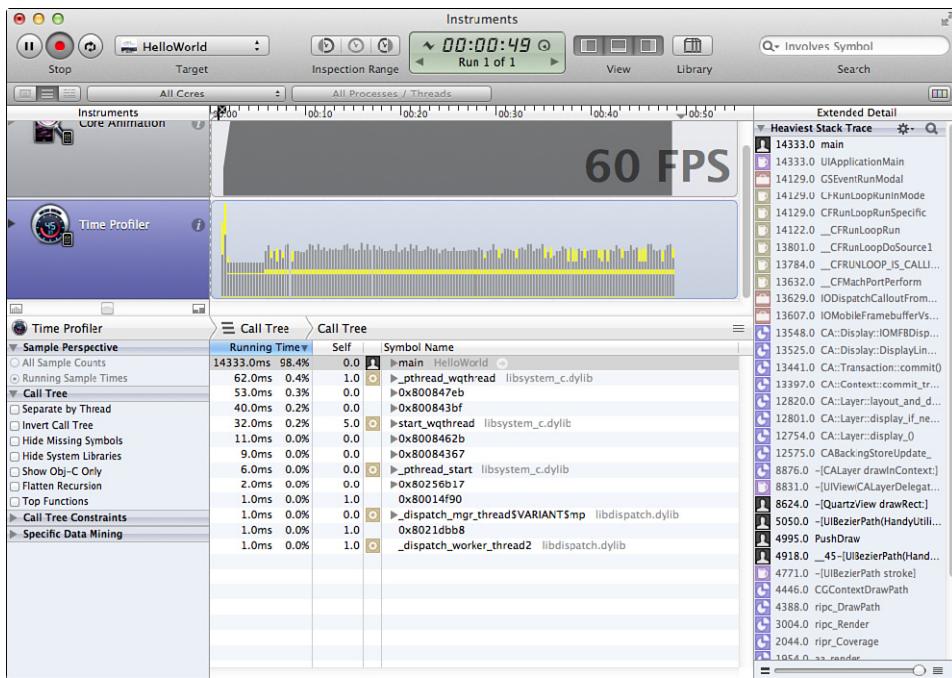


Figure 7-7 The Core Animation template, running in Instruments.

Note

Instruments plays a critical role in making sure your applications run efficiently by profiling how iOS applications work under the hood. The utility samples memory usage and monitors performance. This lets you identify and target problem areas in your applications and work on their efficiency before you ship apps.

Instruments offers graphical time-based performance plots that show where your applications are using the most resources. Instruments is built around the open-source DTrace package developed by Sun Microsystems.

In Xcode 5, the Debug Navigator enables you to track CPU and memory load as your application runs.

When you are done with your display loop, invalidate your display link (using `invalidate`). This removes it from the run loop and disassociates the target/action:

```
[link invalidate];
```

Alternatively, you can set the link’s paused property to YES and suspend the display link until it’s needed again.

Building Marching Ants

The display link technology you just read about also powers regular drawing updates. You can use this approach for many animated effects. For example, Figure 7-8 shows a common “marching ants” display. In this interface, the light gray lines animate, moving around the rectangular selection. First developed by Bill Atkinson for MacPaint on the old-style Macintosh line, it is named for the idea of ants marching in line. This presentation enables users to easily distinguish the edges of a selection.



Figure 7-8 A “marching ants” selection animates by offsetting each dash over time.

Example 7-4 presents a `drawRect:` implementation that draws a marching ants effect. It calculates a dash offset related to the current real-world time and strokes a path that you supply. Although Figure 7-8 uses a rectangle, you can use this code with any path shape.

This method is intended for use in a clear, lightweight view that’s stretched over your interface. This enables you to separate your selection presentation from the rest of your GUI, with a view you can easily hide or remove, as needed.

It uses a 12 point–3 point dash pattern for long dashes and short gaps. Importantly, it uses system time rather than any particular counter to establish its animation offsets. This ensures that any glitches in the display link (typically caused by high computing overhead) won’t affect the placement at each refresh, and the animation will proceed at the rate you specify.

There are two timing factors working here at once. The first is the refresh rate. It controls how often the `drawRect:` method fires to request a visual update. The second controls the

pattern offsets. This specifies to what degree the dash pattern has moved and is calculated, independently, from system time.

To animate, Example 7-4 calculates a phase. This is what the `UIBezierPath` class (and, more to the point, its underlying Quartz `CGPath`) uses to present dashed offsets. The phase can be positive (typically, counterclockwise movement) or negative (clockwise) and specifies how far into the pattern to start drawing. This example uses a pattern that is 15 points long. Every 15 points, the dashes return to their original position.

To calculate an offset, this method applies a factor called `secondsPerFrame`. Example 7-4 cycles every three-quarters of a second. You can adjust this time to decrease or increase the pattern's speed.

Example 7-4 Displaying Marching Ants

```
// The drawRect: method is called each time the display
// link fires. That's because calling setNeedsDisplay
// on the view notifies the system that the view contents
// need redrawing on the next drawing cycle.

- (void) drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextClearRect(context, rect);

    CGFloat dashes[] = {12, 3};
    CGFloat distance = 15;
    CGFloat secondsPerFrame = 0.75f; // Adjust as desired

    NSTimeInterval ti = [NSDate
        timeIntervalSinceReferenceDate] / secondsPerFrame;

    BOOL goesCW = YES;
    CGFloat phase = distance * (ti - floor(ti)) *
        (goesCW ? -1 : 1);
    [path setLineDash:dashes count:2 phase:phase];
    [path stroke:3 color:WHITE_LEVEL(0.75, 1)];
}
```

Drawing Sampled Data

There are many other applications for combining iOS drawing with a display link timer. One of the most practical involves data sampling from one of the onboard sensors. Figure 7-9 shows an app that monitors audio levels. At each link callback, it draws a `UIBezierPath` that shows the most recent 100 samples.

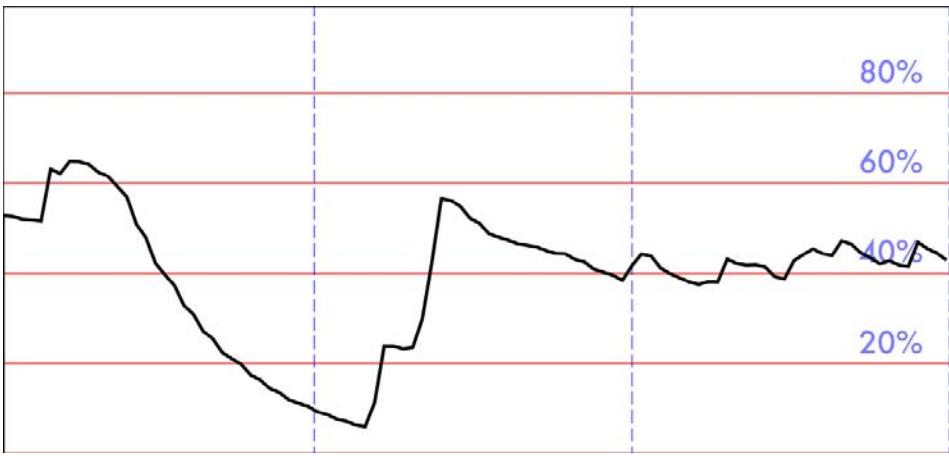


Figure 7-9 Sampling audio over time.

You can, of course, use a static background, simply drawing your data over a grid. If you use any vertical drawing elements (such as the dashed blue lines in Figure 7-9), however, you'll want those elements to move along with your data updates. The easy way to accomplish this is to build a vertical grid and draw offset copies as new data arrives. Here's one example of how to approach this problem:

```
PushDraw(^{
    UIBezierPath *vPath = [vGrid safeCopy];
    OffsetPath(vPath, CGSizeMake(-offset * deltaX, 0));
    AddDashesToPath(vPath);
    [vPath stroke:1 color:blueColor];
});
```

In this snippet, the vertical path offsets itself by some negative change in X position. Repeating this process produces drawings that appear to move to the left over time.

Applying Core Image Transitions

Core Image transitions are another valuable timer-meets-drawing solution. They enable you to create sequences between a source image and a target image in order to build lively visual effects that transition from one to the other.

You start by creating a new transition filter, such as a copy machine-style transition:

```
transition = [CIFilter filterWithName:@"CICopyMachineTransition"];
```

You provide an input image and a target image, and you specify how far along the transition has progressed, from 0.0 to 1.0. Listing 7-6 defines a method that demonstrates this, producing a `CIImage` interpolated along that timeline.

Listing 7-6 Applying the Core Image Copy Machine Transition Effect

```
- (CIImage *)imageForTransitionCopyMachine: (float) t
{
    CIFilter *crop;
    if (!transition)
    {
        transition = [CIFilter filterWithName:
                      @"CICopyMachineTransition"];
        [transition setDefaults];
    }

    [transition setValue: self.inputImage
                  forKey: @"inputImage"];
    [transition setValue: self.targetImage
                  forKey: @"inputTargetImage"];
    [transition setValue: @(fmodf(t, 1.0f))
                  forKey: @"inputTime"];

    // This next bit crops the image to the desired size
    CIFilter *crop = [CIFilter filterWithName: @"CICrop"];
    [crop setDefaults];
    [crop setValue:transition.outputImage
              forKey:@@"inputImage"];
    CIVector *v = [CIVector vectorWithX:0 Y:0
                   Z:_i1.size.width W:_i1.size.width];
    [crop setValue:v forKey:@@"inputRectangle"];
    return [crop valueForKey: @"outputImage"];
}
```

Each Core Image filter uses a custom set of parameters, which are documented in Apple’s Core Image Filter reference. The copy machine sequence is one of the simplest transition options. As Listing 7-6 reveals, it works well with nothing more than the two images and an `inputTime`. You see the transition in action in Figure 7-10.

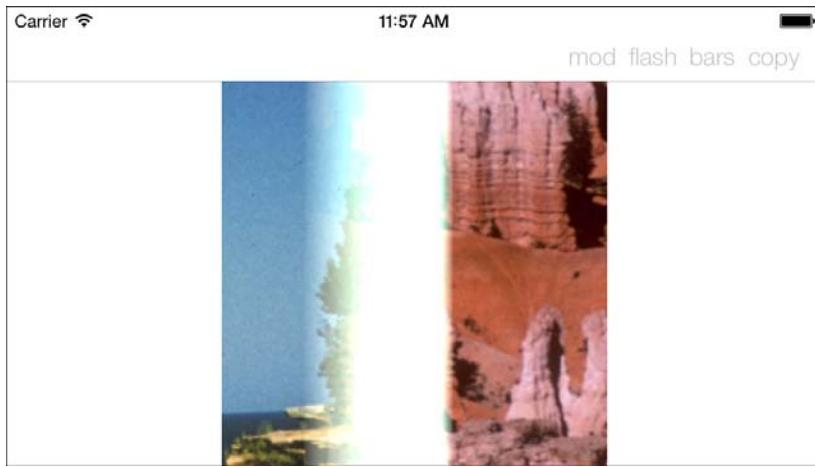


Figure 7-10 The copy machine transition moves from one image to another by mimicking a copy machine with its bright scanning bar. *Public domain images courtesy of the National Park Service.*

A display link enables you to power the transition process. Unlike Example 7-4, Listing 7-7 does not use real-world timing—although you could easily modify these methods to make them do so. Instead, it tracks a progress variable, incrementing it each time the display link fires, moving 5% of the way (`progress += 1.0f / 20.0f`) each time through.

As the link fires, the transition method updates the progress and requests a redraw. The `drawRect:` method in Listing 7-7 pulls the current “between” image from the filter and paints it to the context. When the progress reaches 100%, the display link invalidates itself.

Listing 7-7 Transitioning Using Core Image and Display Links

```
// Begin a new transition
- (void) transition: (int) theType bbis: (NSArray *) items
{
    // Disable the GUI
    for (UIBarButtonItem *item in (bbitems = items))
        item.enabled = NO;

    // Reset the current CATransition
    transition = nil;
    transitionType = theType;

    // Start the transition from zero
    progress = 0.0;
    link = [CADisplayLink displayLinkWithTarget:self
```

```
    selector:@selector(applyTransition)];
    [link addToRunLoop:[NSRunLoop mainRunLoop]
        forMode:NSRunLoopCommonModes];
}

// This method runs each time the display link fires
- (void) applyTransition
{
    progress += 1.0f / 20.0f;
    [self setNeedsDisplay];

    if (progress > 1.0f)
    {
        // Our work here is done
        [link invalidate];

        // Toggle the two images
        useSecond = ! useSecond;

        // Re-enable the GUI
        for (UIBarButtonItem *item in bbitems)
            item.enabled = YES;
    }
}

// Update the presentation
- (void) drawRect: (CGRect) rect
{
    // Fit the results
    CGRect r = SizeMakeRect(_il.size);
    CGRect fitRect = RectByFittingRect(r, self.bounds);

    // Retrieve the current progress
    CIImage *image = [self imageForTransition:progress];

    // Draw it (it's a CIImage, not a CGImage)
    if (!cicontext) cicontext =
        [CIContext contextWithOptions:nil];
    CGImageRef imageRef = [cicontext
        createCGImage:image fromRect:image.extent];

    FlipContextVertically(self.bounds.size);
    CGContextDrawImage(UIGraphicsGetCurrentContext(),
        fitRect, imageRef);
}
```

Summary

This chapter discusses techniques for masking, blurring, and animating drawn content. You read about ways to apply edge effects to selections, soften visuals, and use Core Image transitions in your `drawRect:` routines. Here are some concluding thoughts:

- No matter what kind of drawing you do, profiling your app's performance is a critical part of the development process. Always make space in your development schedule to evaluate and tune your rendering and animation tasks. If you find that in-app processing time is too expensive, consider solutions like threading your drawing (UIKit and Quartz are thread safe when drawn to a context) and predrawing effects into images (this is great for adding precalculated glosses to UI elements like buttons).
- Core Image transitions are a lot of fun, but a little pizzazz goes a very long way. Don't overload your apps with flashy effects. Your app is meant to serve your user, not to draw unnecessary attention to itself. In all things UI, less is usually more.
- When drawing animated material to external screens (whether over AirPlay or via connected cables), make sure you establish a display link for updates, just as you would for drawing to the main device screen.
- Although iOS 7 uses blurring as a key UI element, Apple has not yet made those real-time APIs public at the time this book was being written. Apple engineers recommend rolling your own blurring solutions and caching the results, especially when working with static backdrops.
- Core Image isn't just about transitions. It offers many image-processing *and* image-generation options you might find handy. More and more filters are arriving on iOS all the time. It's worth taking another look if you haven't poked around for a while.

8

Drawing Text

So far in this book, you've seen bits and pieces about strings, but strings haven't formed the focus of a discussion. Their story, as you'll discover in this chapter, extends well beyond picking a point within the drawing context to paint some text or transforming a string into a Bezier path. This chapter dives deep into text, covering techniques for drawing, measuring, and laying out strings.

Drawing Strings

In iOS, it's easy to use labels and text views to display strings. However, when your text acts as a component of a larger layout or drawing, it's time to move to direct string drawing. Doing so unlocks the door to a wide, varied, and powerful set of text layout tools that enable you to add string elements to your contexts.

At the simplest end of the spectrum, you can draw text by telling a string to paint itself. The `drawAtPoint:withAttributes:` method draws any string to the current context. Example 8-1 demonstrates this approach, specifying a font and a text color.

Example 8-1 Drawing Hello World in Gray

```
NSString *string = @"Hello World";
UIFont *font = [UIFont fontWithName:@"Futura" size:36.0f];

// Starting in iOS 7, all string drawing uses attributes
NSDictionary *attributes = @{
    NSFontAttributeName:font,
    NSForegroundColorAttributeName:[UIColor grayColor]
};

// Draw the string
[string drawAtPoint:drawingPoint withAttributes:attributes];
```

The system uses UIKit geometry (top to bottom) to draw your string. Figure 8-1 shows the output of Example 8-1's commands: text drawn in large gray letters.



Figure 8-1 Every `NSString` instance can draw itself at a point you specify.

Point Versus Rect Methods

String drawing methods offer two kinds of API calls: points and rects. This applies to both the `NSString` and `NSAttributedString` classes. The rule of thumb is this:

- A point method, like the one used in Example 8-1, draws a single line—regardless of any word wrapping attributes you've specified. The width of the rendering area is treated as unlimited.
- A rect version draws wrapped text within the boundaries you've provided. Any material that extends beyond those bounds is clipped.

iOS 7 Changes

The `drawAtPoint:withAttributes:` method used in Example 8-1 is new to iOS 7. Most older string drawing methods, such as `drawAtPoint:withFont:`, have been deprecated (see Figure 8-2). New technologies such as layout managers and dynamic text have brought about a revolution in this arena. But as you will see, not all of these methods are ready for direct context drawing yet.

```

Computing Metrics for a Single Line of Text
- sizeWithFont: Deprecated in iOS 7.0
- sizeWithFont:forWidth:lineBreakMode: Deprecated in iOS 7.0
- sizeWithFont:minFontSize:actualFontSize:forWidth:lineBreakMode: Deprecated in iOS 7.0

Computing Metrics for Multiple Lines of Text
- sizeWithFont:constrainedToSize: Deprecated in iOS 7.0
- sizeWithFont:constrainedToSize:lineBreakMode: Deprecated in iOS 7.0

Drawing Strings on a Single Line
- drawAtPoint:withAttributes:
- drawAtPoint:forWidth:withFont:fontSize:lineBreakMode:baselineAdjustment: Deprecated in iOS 7.0
- drawAtPoint:forWidth:withFont:lineBreakMode: Deprecated in iOS 7.0
- drawAtPoint:forWidth:withFont:minFontSize:actualFontSize:lineBreakMode:baselineAdjustment: Deprecated in iOS 7.0
- drawAtPoint:withFont: Deprecated in iOS 7.0

Drawing Strings in a Given Area
- boundingRectWithSize:options:attributes:context:
- drawInRect:withAttributes:
- drawWithRect:options:attributes:context:
- sizeWithAttributes:
- drawInRect:withFont: Deprecated in iOS 7.0
- drawInRect:withFont:lineBreakMode: Deprecated in iOS 7.0
- drawInRect:withFont:lineBreakMode:alignment: Deprecated in iOS 7.0

```

Figure 8-2 iOS 7 introduced many changes to string drawing.

Dynamic Text

Dynamic text is a technology that automatically adapts the members of a font family to fill user interface roles. For example, a “headline” font introduces material sections onscreen. For one user with limited vision, that font might be much larger than the same “headline” font for another user who has 20/20 eyesight. As a user adjusts his or her sizing preferences, the font roles adjust—changing both size and overall weight to preserve text readability beyond simple scaling.

These new features do not match well to UIKit drawing. Drawing creates static images. The dimensions of each context used to draw affects how a font is presented with respect to the overall size of the device. When the drawing context is not a pixel-perfect match to the screen, a font can be stretched, squashed, or otherwise distorted from its “ideal” user-driven size.

In addition, a drawn element cannot automatically update to dynamic text notifications. These notifications are generated whenever a user tweaks text settings. If elements need to adapt to dynamic text, avoid drawing them. Use labels and text views instead.

When you’re producing material for PDFs, for application art, or for other image output, avoid dynamic text. Use specific fonts, with specific traits and sizes, to do your drawing. This is why this chapter applies exact faces and sizing in its examples.

Text Kit

Text Kit is another exciting development to grow out of iOS 7. It enables you to apply Core Text-style typesetting to text views. Core Text is Apple's C-based technology that enables you to create flexible and powerful typeset solutions from code. Text Kit, which is built on top of Core Text, extends that functionality to UIKit views.

You can use Text Kit to draw to UIKit contexts as well. As you see in Figure 8-3, when you attempt to do anything too challenging, such as draw inside shapes or use column-based drawing outside the intended `UITextView` target, the results can be problematic. Text Kit is still very new, very young, and somewhat buggy.



Figure 8-3 Text Kit is experiencing some growing pains, as you see with the bug displayed in this screen shot. Text that should remain in the center circle has crept out both to the left and, if you look carefully for descenders, to the top of the surrounding view.

This chapter discusses Core Text solutions instead of Text Kit ones for drawing. Despite its C-based API, Core Text remains the more powerful and reliable solution for complex text layout.

Text Attributes

Much of the iOS typesetting story consists of understanding attributes. Attributes are a set of features, like font choice or text color, applied to text within a certain range. Attributed

strings, as the name implies, add characteristics to select substrings. Each attributed string contains both source text and range-specific attributes applied to that string. To get a sense of how attributes work and can combine, consider Figure 8-4. It shows a string with added foreground color and drop shadow attributes.



Figure 8-4 Text attributes include fonts, colors, and shadows. They can be arbitrarily combined and overlapped.

Creating Attributed Strings

When building text with typesetting traits, you work with members of the `NSAttributedString` class and, more typically, its mutable cousin `NSMutableAttributedString`. The mutable version offers more flexibility, allowing you to layer attributes individually rather than having to add all traits at once.

To create a nonmutable attributed string, you allocate it and initialize it with text and an attribute dictionary. Example 8-2 draws the same large gray “Hello World” you saw in Figure 8-1, but this time it does so using an attributed string rather than an `NSString` instance. The `drawAtPoint:` method fetches all the details, such as text color and font, from the attributes stored within the string.

Example 8-2 Creating an Attributed String with an Attributes Dictionary

```
// Create an attributes dictionary
NSMutableDictionary *attributes =
[NSMutableDictionary dictionaryWithCapacity:10];

// Set the font
attributes[NSShadowAttributeName] =
[NSFont fontWithName:@"Futura" size:36.0f];

// Set the foreground color
attributes[NSForegroundColorAttributeName] =
[UIColor grayColor];
```

```
// Build an attributed string with the dictionary
attributedString = [[NSAttributedString alloc]
    initWithString:@"Hello World" attributes: attributes];

// Draw the attributed string
[attributedString drawAtPoint: drawingPoint];
```

Mutable Attributed Strings

Mutable instances enable you to add each attribute individually, to the entire string at once (as demonstrated in Example 8-3), or to subranges (as you see in Figure 8-2). You use `addAttribute:value:range:` requests to specify the attribute, the range, and the value.

Other methods enable you to set attributes with dictionaries, as you would with nonmutable instances (`setAttributes:range:`), or remove attributes (`removeAttributes:range:`) from ranges within the string. You can also insert and append attributed strings (`insertAttributedString:atIndex:` and `appendAttributedString:`) to build up complex instances.

Ultimately, Example 8-3 draws the same large gray “Hello World” output you saw in Figure 8-1, using the same `drawAtPoint:` entry point as Example 8-2.

Example 8-3 Layering Attributes into a Mutable Attributed String

```
// Build mutable attributed string
attributedString = [[NSMutableAttributedString alloc]
    initWithString:@"Hello World"];

// Set the range for adding attributes
NSRange r = NSMakeRange(0, attributedString.length);

// Set the font
[attributedString
    addAttribute:NSFontAttributeName
    value:[UIFont fontWithName:@"Futura" size:36.0f] range:r];

// Set the color
[attributedString
    addAttribute:NSSForegroundColorAttributeName
    value:[UIColor grayColor] range:r];

// Draw the attributed string
[attributedString drawAtPoint: inset.origin];
```

Kinds of Attributes

iOS typesetting attributes define the way text is drawn and styled into the context. The following sections enumerate the attributes you work with and the values you assign to them.

Fonts

Attribute: `NSFontAttributeName`

Assign a `UIFont` object to set the text font. Examples 8-2 and 8-3 set this attribute to 36-point Futura. Figure 8-5 shows a variety of faces (Chalkboard, Helvetica, and Times New Roman) applied as font attributes.



Figure 8-5 Font attributes applied to text.

Text Colors

Attributes: `NSForegroundColorAttributeName` and
`NSBackgroundColorAttributeName`

`UIColor` objects set the color of the text and the color shown behind the text. Figure 8-6 shows text drawn with a green foreground color on a purple background.



Figure 8-6 Foreground and background colors specify the text fill color and the background drawn behind the text.

Attribute: **NSStrikeColorAttributeName**

A `UIColor` specifies a stroke color. This is, for the most part, synonymous with the foreground color in that it's only used when you specify a stroke width attribute. You see it distinguished from that foreground color only when you apply a negative stroke width, as in the next section.

Stroke Style

Attribute: **NSStrikeWidthAttributeName**

Assign an `NSNumber` object that stores a floating-point value that defines the stroke width, as a percentage of the font point size. For example, in Figure 8-7, you see several widths (1, 4, and 8) applied.



Figure 8-7 Positive stroke widths outline text. From top to bottom, the stroke widths applied are 1, 4, and 8.

Negative numbers both stroke (using the stroke color) and fill (using the foreground color) the text. Positive numbers create a “hollow” presentation, stroking the edges of each character glyph, as shown in Figure 8-8.



Figure 8-8 Top: Positive stroke values outline character glyphs using the current stroke color (or foreground color, if a stroke color is not set) but do not fill the interior. Bottom: Negative stroke values fill text with the foreground color and stroke the edges with the stroke color.

Strikethroughs

Attribute: `NSStrikethroughStyleAttributeName`

This key specifies whether an item uses strikethrough. Pass an `NSNumber` instance: either 0 (disable striking) or 1 (enable striking).

Attribute: `NSStrikethroughColorAttributeName`

Pass a color for this attribute to assign a custom tint to your strikethroughs.

Strikethroughs are a typographical convention that adds a horizontal line to text, indicating that material has been redacted away. Figure 8-9 shows strikethroughs, highlighting the new iOS 7 attribute that assigns a color to the strike.



Figure 8-9 iOS 7 strikethrough formats now enable you to add custom colors.

Underlines

Attribute: `NSUnderlineStyleAttributeName`

iOS 7 introduces a variety of new underlining styles. Styles include single, double, and thick lines, as well as dashed, dotted, and word-by-word options. You can assign an `NSNumber` you create by masking together underline options.

An attributed string's underline attribute (`NSunderlineStyleAttributeName`) provides four base styles. These values are `NSUnderlineStyleNone` (0, basically no underline), `NSUnderlineStyleSingle` (1), `NSUnderlineStyleThick` (2), and `NSUnderlineStyleDouble` (9). Figure 8-10 shows these options.

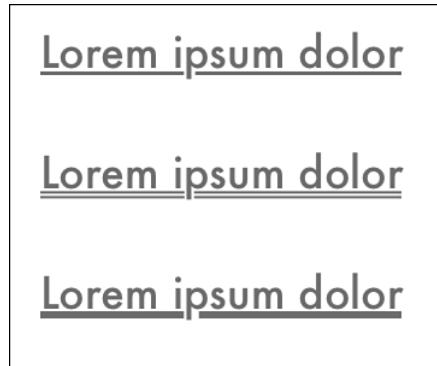


Figure 8-10 Basic underlines include single, double, and thick styles.

In addition to the base styles, you might want to add underline patterns. Choose from solid (`NSUnderlinePatternSolid`, the default), dots (`NSUnderlinePatternDot`), dashes (`NSUnderlinePatternDash`), dash-dots (`NSUnderlinePatternDashDot`), and dash-dot-dots (`NSUnderlinePatternDashDotDot`). Mask these together with whatever basic underline style you use, as in the following example:

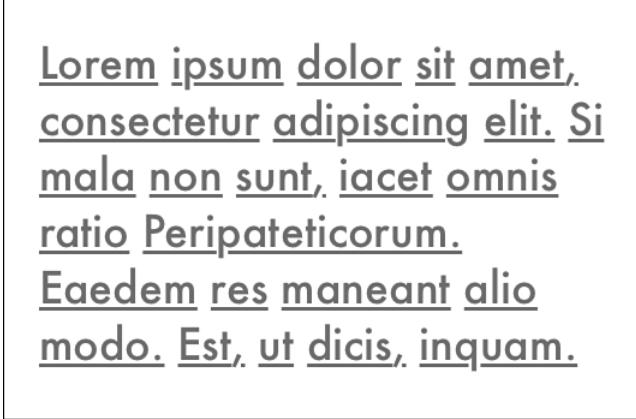
```
attributes[NSUnderlineStyleAttributeName] =  
@(NSUnderlineStyleThick | NSUnderlinePatternDash);
```

Figure 8-11 shows examples of these patterns in use, OR'ed together with the single underline style.



Figure 8-11 New iOS 7 underlining patterns include (top to bottom): solid, dotted, dashed, dash-dot, and dash-dot-dot.

A final option is `NSUnderlineByWord`. When you OR it into your attribute mask, it underlines each word individually. You see this in Figure 8-12.



Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Si
mala non sunt, iacet omnis
rato Peripateticorum.
Eaedem res maneant alo
modo. Est, ut dicis, inquam.

Figure 8-12 The underline-by-word option applies whatever underline style and pattern you've selected to each word, leaving gaps between.

Attribute: **NSUnderlineColorAttributeName**

Assign a color instance to set a distinct underline color. This is shown in Figure 8-13.



Figure 8-13 This underline has a green tint attribute applied.

Shadows

Attribute: **NSShadowAttributeName**

Assign an `NSShadow` object. This class sets a shadow's color, offset, and blur radius, as you see in Figure 8-14. Shadows work as they do in contexts. You supply a size for the shadow offset, a float for the blur radius, and a `UIColor` for the shadow color:

```
 NSShadow *shadow = [[NSShadow alloc] init];
shadow.shadowBlurRadius = 2.0f;
shadow.shadowOffset = CGSizeMake(2.0f, 2.0f);
shadow.shadowColor = [UIColor grayColor];
attributes[NSShadowAttributeName] = shadow;
```



Hello World

Figure 8-14 Use the `NSShadow` class to add shadow attributes.

Note

On OS X, you can set an `NSShadow` instance to apply its parameters to the active drawing context. This feature has not yet migrated to iOS.

Baselines

Attribute: `NSBaselineOffsetAttributeName`

This attribute, assigned as an `NSNumber`, adds an offset from the normal text drawing position, as you see in Figure 8-15. Use it for vertically positioned elements like subscripts and superscripts:

```
[string addAttribute:NSBaselineOffsetAttributeName
    value:@(20) range:NSMakeRange(6, 5)];
```



Figure 8-15 Baseline offsets move text up or down from the normal text line. I underlined this example to highlight the difference.

Text Effects

Attribute: `NSTextEffectAttributeName`

Assign an effect through a predefined `NSString` constant.

iOS 7 introduced a new kind of text attribute that applies effects to fonts. It shipped with just one option, the “letterpress” effect (`NSTextEffectLetterpressStyle`). This attribute style creates a slightly 3D text effect, shown in Figure 8-16:

```
attributes[NSTextEffectAttributeName] =  
    NSTextEffectLetterpressStyle;
```

This figure uses a dark background to showcase the difference between the two examples. On a light background, the letterpress changes can be hard to recognize. The text at the bottom of Figure 8-16 uses letterpress; the text at the top does not. Other text effects are expected in future iOS updates.



Figure 8-16 The letterpress attribute (bottom) creates a subtle 3D effect. The top text draws the text without adding letterpress.

Obliqueness and Expansion

Attribute: `NSObliquenessAttributeName`

Assign an `NSNumber` value from `@(-1.0)` to `@(1.0)`.

The `NSObliquenessAttributeName` attribute, which is new to iOS 7, adds a slant to your text. You select a skew from -1 to 1. Figure 8-17 shows a nonslanted item along with two examples of the applied attribute.



Figure 8-17 From top to bottom, these examples show obliqueness values of 0 (no slant), 0.15 (slight positive slant to the right), and -0.5 (strong negative slant to the left).

Attribute: **NSExpansionAttributeName**

You can assign an `NSNumber` value from @0.0 and up.

The `NSExpansionAttributeName` attribute, introduced in iOS 7, provides a horizontal multiplier that spaces out text. Figure 8-18 shows how expansion applies to text.



Figure 8-18 From top to bottom, these examples highlight how expansion values work. They show 0 (no expansion), 0.15 (moderate expansion), and 0.25 (strong expansion). Both text and spacing are affected by this attribute.

Ligatures and Kerning

Attribute: `NSLigatureAttributeName`

This attribute references an `NSNumber` that selects from “use no ligatures” (0) and “use the default ligature” (1).

Ligatures refer to the way that individual glyphs (character pictures) can be bound together, such as *f* with *i*, as shown in Figure 8-19. When enabled, iOS replaces separate letters with a single combined image for certain sequences. These pairs generally occur when a feature of one letter extends into the space occupied by another. Common English ligatures include *fi*, *fj*, *fl*, *ff*, *ffi*, and *ffl*. These vary by font implementation.

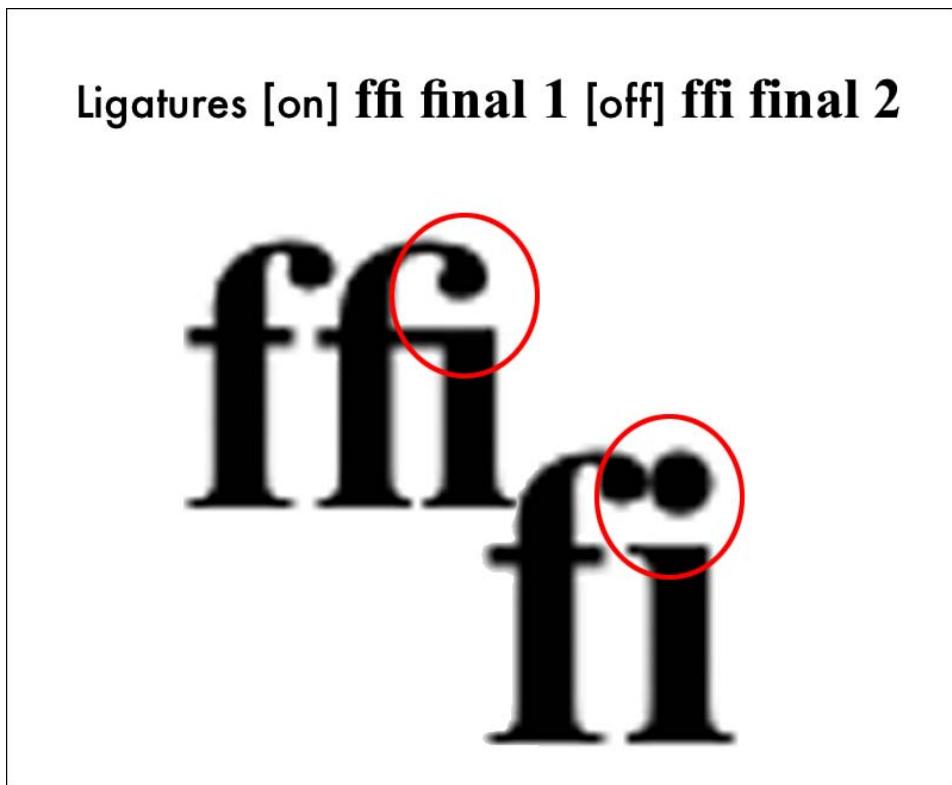


Figure 8-19 Ligatures combine certain letter combinations like *f* and *i* into a single character glyph. Notice the separate dot on the *i* when ligatures are disabled.

Attribute: `NSKernAttributeName`

This `NSNumber` indicates whether kerning is enabled (1) or disabled (0, the default).

Kerning allows typesetters to adjust the space between letters so they naturally overlap, such as when placing a capital *A* next to a capital *V*. Figure 8-20 shows kerning in action.

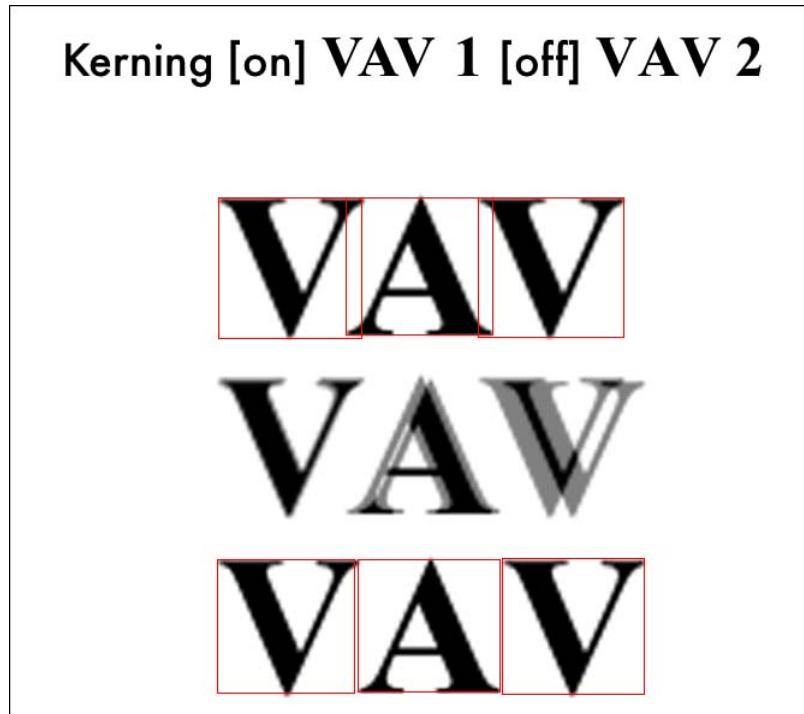


Figure 8-20 Kerning reduces the space between characters to overlap appropriate letter pairs. Top: Kerning is enabled, and character bounding boxes partially cover each other. Bottom: Kerning is disabled, so each character is freestanding.

Paragraph Styles

Attribute: `NSParagraphStyleAttributeName`

The `NSParagraphStyle` object is used to specify a number of paragraph settings, including alignment, line break mode, and indentation.

Paragraph styles are stored in their own objects, members of the `NSParagraphStyle` class. You use the mutable version of the class `NSMutableParagraphStyle` to iteratively set style specifics. Example 8-4 creates the presentation shown in Figure 8-21, using extra-large spacing between paragraphs and a generous first-line indent.

Example 8-4 Layering Attributes into a Mutable Attributed String

```
NSMutableParagraphStyle *paragraphStyle =  
    [[NSMutableParagraphStyle alloc] init];  
paragraphStyle.alignment = NSTextAlignmentLeft;  
paragraphStyle.lineBreakMode = NSLineBreakByWordWrapping;  
paragraphStyle.firstLineHeadIndent = 36.0f;  
paragraphStyle.lineSpacing = 8.0f;  
paragraphStyle.paragraphSpacing = 24.0f; // Big!  
[attributedString addAttribute:NSParagraphStyleAttributeName  
    value:paragraphStyle range:r];
```

**Lorem ipsum dolor sit amet, consectetur
 adipiscing elit. Iam enim adesse poterit.
 Scisse enim te quis coarguere possit?
 Primum divisit ineleganter; Itaque ab his
 ordiamur. Istam voluptatem, inquit, Epicurus
 ignorat? Quid nunc honeste dicit?**

**Velut ego nunc moveor. Duo Reges:
 constructio interrete. Venit ad extremum;
 Quis enim redargueret?**

Figure 8-21 Paragraph style attributes include indentation, paragraph-to-paragraph spacing, alignment, and more.

Most of these values refer to points, such as the spacing between lines or paragraphs, and indentation. If you’re careful, you can control these features on a paragraph-by-paragraph basis. Style objects are assigned, not copied. One secret lies in making sure to assign distinct paragraph style objects for each paragraph. If you do not, as I discovered the hard way long ago, you may create an attributed result whose paragraph styles all point to the same object: Update one, and you update them all.

Paragraph Style Properties

The following list enumerates paragraph style properties. Apply these to attributed strings to specify how iOS draws that text content:

- **alignment**—The paragraph alignment, as an `NSTextAlignment` value. Generally, you align paragraphs left, right, center, or justified.
- **firstLineHeadIndent**—The indentation leading into each paragraph’s first line. Supply a nonnegative value. Example 8-4 uses a 36-point first-line head indent.
- **headIndent**—The indentation on the leading edge of the paragraph—that is, the left edge in left-to-right languages like English, and the right edge in right-to-left languages like Arabic and Hebrew. Typically used when working with block quotes and other indented material, this property enables you to move in away from the text container.
- **tailIndent**—The opposite of head indentation, this indents from the trailing edge of the paragraph with respect to its text drawing container. As with other indentations, it uses a nonnegative floating-point value.
- **maximumLineHeight and minimumLineHeight**—The maximum/minimum height, in points, that a line occupies.
- **lineHeightMultiple**—According to Apple documentation, “The natural line height of the receiver is multiplied by this factor (if positive) before being constrained by minimum and maximum line height.”
- **lineSpacing**—Otherwise known as leading, refers to the space (in points) between paragraph lines.
- **paragraphSpacing**—Extra space, in points, between one paragraph and the next. In Example 8-4, this was set to 24 points.
- **paragraphSpacingBefore**—This additional space is added before the first paragraph at the start of the drawing area.
- **lineBreakMode**—An instance of `NSLineBreakMode` that specifies the kind of wrapping, truncation, or clipping to apply. Options include word wrapping (entire words wrap), character wrapping (words may wrap in the middle), clipping, head truncation (for example, “...lo World”), tail truncation (for example, “Hello Wor...”), and middle truncation (for example, “Hel...orl”).

- **hyphenationFactor**—The threshold for hyphenation. Apple writes, “Hyphenation is attempted when the ratio of the text width (as broken without hyphenation) to the width of the line fragment is less than the hyphenation factor. When the paragraph’s hyphenation factor is 0.0, the layout manager’s hyphenation factor is used instead. When both are 0.0, hyphenation is disabled.”
- **baseWritingDirection**—When set to the natural direction, the paragraph defaults to whatever value has been set in the locale associated with the current language settings—left to right or right to left.

Drawing with Core Text

iOS 7’s Text Kit made great inroads into providing high-end typesetting technologies traditionally associated with Core Text. Text Kit provides an Objective-C-based set of classes and protocols that supply typesetting features for `UITextView`. Core Text, in contrast, is a C-based API. Both typesetting features can be drawn directly to contexts. As Figure 8-3 showed, Text Kit isn’t quite fully cooked yet. For that reason, this section covers Core Text instead of Text Kit.

In Core Text, Bezier paths enable you to draw text into any shape or layout you specify. Figure 8-22 shows a simple example. This image uses Core Text layout within a star shape.



Figure 8-22 Drawing attributed text into a path. The text wraps along the path edges.

Figure 8-22 uses character wrapping, so partial words are split between lines, but no characters or words are *missing* from the layout, the way they would be if you used `addClip` and drew into the shape's bounds. All the text appears within the path but is wrapped to the surrounding shape.

Normally, of course, you don't draw text into such an odd shape. I did so here because I wanted to demonstrate the function in Listing 8-1. It accepts a path and an attributed string, and it draws that string into that path. It does this courtesy of a Core Text frame setter, an object that builds containers ("frames") for drawing text.

The Core Text framework offers a powerful text layout and font management feature set. Intended for apps that work in the text processing space, Core Text includes tools like frame setters that enable you to define geometric destinations for drawing complex string layouts. These layouts respect paragraph styles like alignment, tab stops, line spacing, and indentation that you apply through attributed strings.

Thanks to the Core Text framework, all drawing takes place in Quartz space. The `CGPath` you supply must also be defined in Quartz geometry. That's why I picked a shape for Figure 8-11 that is not vertically symmetrical. Listing 8-1 handles this drawing issue by copying whatever path you supply and mirroring that copy vertically within the context. Without this step, you end up with the results shown in Figure 8-23: The text still draws top to bottom, but the path uses the Quartz coordinate system.



Figure 8-23 Core Text expects paths to use the Quartz coordinate system.

Listing 8-1 Drawing Attributed Strings into Path Containers

```
void DrawAttributedStringInBezierPath(
    UIBezierPath *path,
    NSAttributedString *attributedString)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
        COMPLAIN_AND_BAIL(
            @"No context to draw into", nil);

    // Mirror a copy of the path
    UIBezierPath *copy = [path safeCopy];
    MirrorPathVerticallyInContext(copy);

    // Build a framesetter and extract a frame destination
    CTFramesetterRef framesetter =
        CTFramesetterCreateWithAttributedString(
            (__bridge CFAttributedStringRef) attributedString);
    CTFrameRef theFrame = CTFramesetterCreateFrame(
        framesetter, CFRRangeMake(0, attributedString.length),
        copy.CGPath, NULL);

    // Draw into the frame
    PushDraw:^{
        CGContextSetTextMatrix(context, CGAffineTransformIdentity);
        FlipContextVertically(GetUIKitContentSize());
        CTFrameDraw(theFrame, UIGraphicsGetCurrentContext());
    });

    CFRelease(theFrame);
    CFRelease(framesetter);
}

// Flip the path vertically with respect to the context
void MirrorPathVerticallyInContext(UIBezierPath *path)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    CGSize size = GetUIKitContentSize();
    CGRect contextRect = SizeMakeRect(size);
    CGPoint center = RectGetCenter(contextRect);

    // Flip path with respect to the context size
```

```

CGAffineTransform t = CGAffineTransformIdentity;
t = CGAffineTransformTranslate(t, center.x, center.y);
t = CGAffineTransformScale(t, 1, -1);
t = CGAffineTransformTranslate(t, -center.x, -center.y);
[path applyTransform:t];
}

```

Text Matrices

Text matrices define transformations that individually apply to each character rather than to the context as a whole. They are stored as part of each context's `GState`. When intermingling UIKit and Core Text drawing, make sure to reset the text matrix in your Core Text calls as Listing 8-1 does within its `PushDraw()` block.

Let's look at how you perform that update. The following call resets any text transforms used to render strings into the context:

```
CGContextSetTextMatrix(context, CGAffineTransformIdentity);
```

To understand why you must perform this step, consider Figure 8-24. The text in this figure was drawn twice—first using UIKit attributed string drawing and then using Core Text. The Core Text output appears upside down. Each letter is individually reflected vertically.



The figure shows a rectangular box containing two versions of the same text block. The top version is rendered correctly by UIKit, appearing as a single continuous line of text. The bottom version is rendered by Core Text and appears upside down, with each letter individually reflected vertically. The text is a Latin passage from Cicero's *De Finibus*.

Latin text (top):

...lorem ipsum dolor sit amet, consectetur
adipiscing elit. Maximus dolor, inquit, brevis est.
Duo Reges: constructio interrete: Bonum
Integritas corporis: misera debilitas: Quis enim
redargueret? Fallit inquit possimus: Eadem hinc
recte hinc: fallit inquit possimus: Eadem hinc
mea adversum te oratio est.
Est quod si misera debilitas: Quis enim

Latin text (bottom):

...lorem ipsum dolor sit amet, consectetur
adipiscing elit. Maximus dolor, inquit, brevis est.
Duo Reges: constructio interrete: Bonum
Integritas corporis: misera debilitas: Quis enim
redargueret? Fallit inquit possimus: Eadem hinc
recte hinc: fallit inquit possimus: Eadem hinc
mea adversum te oratio est.
Est quod si misera debilitas: Quis enim

Figure 8-24 Drawing text into a complex Bezier path may produce unexpected results.

This odd effect happens because UIKit string drawing changed the context's text matrix. You see how this happens in Example 8-5.

Example 8-5 Drawing Attributed Strings into Paths

```

UIGraphicsBeginImageContextWithOptions(size, NO, 0.0);
CGAffineTransform t;

// Retrieve the initial text matrix
t = CGContextGetTextMatrix(UIGraphicsGetCurrentContext());

```

```
NSLog(@"Before: %f", atan2f(t.c, t.d));  
  
// Draw the string  
[string drawInRect:targetRect];  
  
// Retrieve the changed text matrix  
t = CGContextGetTextMatrix(UIGraphicsGetCurrentContext());  
NSLog(@"After: %f", atan2f(t.c, t.d));  
  
UIGraphicsEndImageContext();
```

You might expect both log statements to report no rotation. Instead, here's what actually happens. The resulting 180-degree rotation used to support UIKit string drawing explains the output shown in Figure 8-24:

```
2013-05-10 09:38:06.434 HelloWorld[49888:c07] Before: 0.000000  
2013-05-10 09:38:06.438 HelloWorld[49888:c07] After: 3.141593
```

Unfortunately, you cannot work around this by saving and restoring the context's graphic state. According to Apple, "Note that the text matrix is not a part of the graphics state—saving or restoring the graphics state has no effect on the text matrix. The text matrix is an attribute of the graphics context, not of the current font."

Instead, you explicitly reset the text matrix when you switch to Core Text drawing, as in Listing 8-1:

```
CGContextSetTextMatrix(context, CGAffineTransformIdentity);
```

Drawing Columns

Figure 8-25 demonstrates a fundamental problem in Core Text. This involves drawing Core Text into columns. When you lay out columns, text should wrap at the edge of each column and not continue between them, as happens here. For example, at the top of the right column, the word *sitting* should appear on the second line of the first column, as it continues on from *tired of*. Instead, it's placed to the right on the top line of the second column of the drawing. The text flows across columns and then down rather than all the way down one column and then all the way down the next.

Alice was beginning to get very tired of having nothing to do: once or twice she had but it had no pictures or conversations in it, Alice 'without pictures or conversation?'

So she was considering in her own mind (as feel very sleepy and stupid), whether the worth the trouble of getting up and picking with pink eyes ran close by her.

There was nothing so very remarkable in the way to hear the Rabbit say to itself, 'Oh thought it over afterwards, it occurred to her at the time it all seemed quite natural); but its waistcoat-pocket, and looked at it, and it flashed across her mind that she had never

sitting by her sister on the bank, and of peeped into the book her sister was reading, 'and what is the use of a book,' thought

well as she could, for the hot day made her pleasure of making a daisy-chain would be the daisies, when suddenly a White Rabbit

that; nor did Alice think it so very much out of dear! Oh dear! I shall be late!' (when she that she ought to have wondered at this, but when the Rabbit actually took a watch out of then hurried on, Alice started to her feet, for before seen a rabbit with either a waistcoat-

Figure 8-25 Text does not naturally know where to wrap in this complex path. Instead of moving down each column, the text flows across both columns before starting the next line.

The problem is that the Core Text framesetter treats this entire Bezier path (you see the path in Figure 8-26) as a single shape. Its two vertical rectangles are meant to display independent columns of text. What's happening, however, is that the Core Text framesetter builds its frame using only one test. It determines points inside the path and points outside the path. All other considerations are omitted. Core Text and iOS in general have no concept of "column," so the default technology doesn't support my expectation of column-by-column layout.



Figure 8-26 This path contains two rectangles.

Figure 8-27 shows what I want this layout to do. Here, *sitting* wraps properly after *tired of*. The text proceeds down the first column and then continues to the second. This layout treats the two columns as a single output flow. Text moves from one column to the next.

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, 'and what is the use of a book,' thought Alice 'without pictures or conversation?'

So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

There was nothing so very remarkable in that; nor did Alice think it so very much out of the way to hear the Rabbit say to itself, 'Oh dear! Oh dear! I shall be late!' (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually took a watch out of its waistcoat-pocket, and looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it, and fortunately was just in time to see it pop down a large rabbit-hole under the

Figure 8-27 This text properly respects column layout.

Compare the visual styles in Figures 8-25 and 8-27. Figure 8-27 looks properly “text”-y, with a series of well-formed, short paragraphs. Figure 8-25 displays many more gap lines and less text in the right column. Its layout looks slightly less appealing than the layout in Figure 8-27.

I generated the “before” image of Figure 8-25 by calling `DrawAttributedStringInBezierPath()` from Listing 8-1. It actually takes very little work to adapt this function to move from the incorrect flow in Figure 8-25 to the proper layout in Figure 8-27. Listing 8-2 shows how.

This new function is called `DrawAttributedStringIntoSubpath()`. It works on a subpath-by-subpath basis and updates a string remainder parameter with any attributed string content that could not be drawn into the path. To accomplish this, it queries the Core Text frame for its visible string range. This function calculates the remainder of the attributed string—that is, the portion that was *not* visible—and assigns that to the `remainder` parameter.

The second function in Listing 8-2 is `DrawAttributedStringInBezierSubpaths()`. This entry point iterates through a path’s subpaths. At each stage, it retrieves the “remainder” string and applies it to the next stage of drawing. The function returns when it finishes drawing subpaths or when the remainder’s length drops to zero.

Listing 8-2 Drawing Attributed Strings into Independent Subpaths

```
void DrawAttributedStringIntoSubpath(
    UIBezierPath *path, NSAttributedString *attributedString,
    NSAttributedString **remainder)
{
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL)
        COMPLAIN_AND_BAIL(@"No context to draw into", nil);

    // Handle vertical mirroring
    UIBezierPath *copy = [path safeCopy];
    MirrorPathVerticallyInContext(copy);

    // Establish the framesetter and retrieve the frame
    CTFramesetterRef framesetter =
        CTFramesetterCreateWithAttributedString(
            __bridge CFAttributedStringRef) attributedString);
    CTFrameRef theFrame = CTFramesetterCreateFrame(framesetter,
        CFRRangeMake(0, attributedString.length), copy.CGPath, NULL);

    // If the remainder can be dereferenced, calculate
    // the remaining attributed string
    if (remainder)
    {
```

```

CFRange range = CTFrameGetVisibleStringRange(theFrame) ;
NSInteger startLocation = range.location + range.length;
NSInteger extent = attributedString.length - startLocation;
NSAttributedString *substring =
    [attributedString attributedSubstringFromRange:
        NSMakeRange(startLocation, extent)];
*remainder = substring;
}

// Perform the drawing in Quartz coordinates
PushDraw:^{
    FlipContextVertically(GetUIKitContentSize());
    CTFrameDraw(theFrame, UIGraphicsGetCurrentContext());
};

// Clean up the Core Text objects
CFRelease(theFrame);
CFRelease(framesetter);
}

void DrawAttributedStringInBezierSubpaths(UIBezierPath *path,
    NSAttributedString *attributedString)
{
    NSAttributedString *string;
    NSAttributedString *remainder = attributedString;

    // Iterate through subpaths, drawing the
    // attributed string into each section
    for (UIBezierPath *subpath in path.subpaths)
    {
        string = remainder;
        DrawAttributedStringIntoSubpath(
            subpath, string, &remainder);
        if (remainder.length == 0) return;
    }
}

```

Image Cutouts

Listing 8-2 works best for columns or other standalone elements in your drawing context. It does *not* work for layouts with holes you create using the even/odd fill rule. If you use the `DrawAttributedStringInBezierSubpaths()` function, you'll end up drawing the text across the entire path, and then again into the hole, as the outer path and its inner hole will separate into two distinct subpaths.

To work with complex paths that leverage the even/odd fill rule, perform your own path decomposition in a way that makes topological sense. Then, call directly into the `DrawAttributedStringIntoSubpath()` function.

That said, the even/odd fill rule *does* enable you to create simple cutouts within your paths. These can accommodate image drawing, as you see in Figure 8-28. To draw this, I created an internal rectangle and added it to my path. This established a “hole” big enough to draw my image.

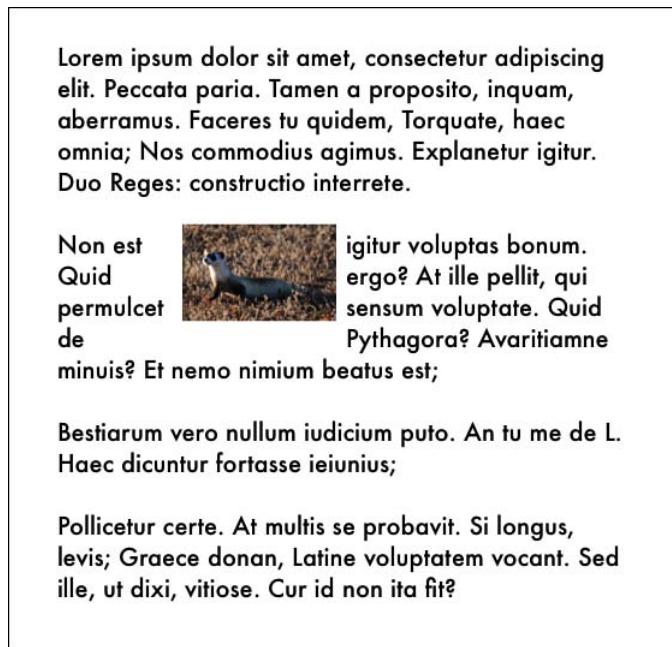


Figure 8-28 Drawing an image onto a path cutout.

Drawing Attributed Text Along a Path

Drawing text along a path presents another common typesetting challenge. Figure 8-29 shows a string drawn onto a star-shaped path. To highlight the fact that the input is an attributed string, I’ve randomly colored each letter before performing any drawing.

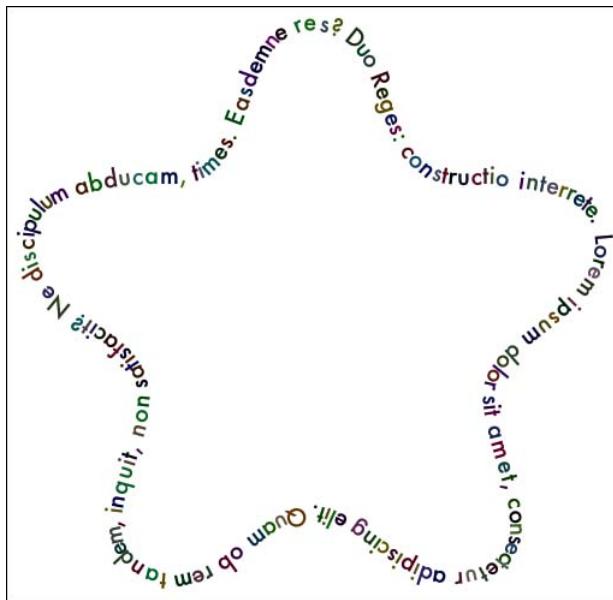


Figure 8-29 Typesetting along a path.

Listing 8-3 calculates the rendered size of each character in the attributed string, determining the bounding height and width. Knowing this size allows the drawing function to determine how much of the path each character (or “glyph,” if you want to use Core Text terms) consumes:

```
CGRect bounding = [item boundingRectWithSize:CGSizeMake(  
    CGFLOAT_MAX, CGFLOAT_MAX) options:0 context:nil];
```

This bounding rectangle establishes where the glyph center would appear if laid out along a line. Listing 8-3 uses that distance to calculate a percentage of the path’s length, and Chapter 5’s path interpolation returns a position and slope for placement. As you saw in Chapter 1, you can translate and rotate the context to position and orient your text exactly. This allows the string to render using `NSAttributedString`drawAtPoint:` method.

After consuming the entire path, the routine stops. This clips any remaining characters; they simply don’t get drawn. If you want to ensure that the entire string appears, you need to fiddle with your font choices to match the path length.

Listing 8-3 Laying Out Text Along a Bezier Path

```
@implementation UIBezierPath (TextUtilities)  
- (void) drawAttributedString:  
    (NSAttributedString *) string
```

```

{
    if (!string) return;
    CGContextRef context = UIGraphicsGetCurrentContext();
    if (context == NULL) COMPLAIN_AND_BAIL(
        @"No context to draw into", nil);

    // Check the points
    if (self.elements.count < 2) return;

    // Keep a running tab of how far the glyphs have traveled to
    // be able to calculate the percent along the point path
    float glyphDistance = 0.0f;
    float lineLength = self.pathLength;

    for (int loc = 0; loc < string.length; loc++)
    {
        // Retrieve the character
        NSRange range = NSMakeRange(loc, 1);
        NSAttributedString *item =
            [string attributedSubstringFromRange:range];

        // Start halfway through each character
        CGRect bounding = [item boundingRectWithSize:
            CGSizeMake(CGFLOAT_MAX, CGFLOAT_MAX)
            options:0 context:nil];
        glyphDistance += bounding.size.width / 2;

        // Find new point on path
        CGPoint slope;
        CGFloat percentConsumed = glyphDistance / lineLength;
        CGPoint targetPoint =
            [self pointAtPercent:percentConsumed
                withSlope:&slope];

        // Accommodate the forward progress
        glyphDistance += bounding.size.width / 2;
        if (percentConsumed >= 1.0f) break;

        // Calculate the rotation
        float angle = atan(slope.y / slope.x);
        if (slope.x < 0) angle += M_PI;

        // Draw the glyph
        PushDraw(^{
            // Translate to target on path
            CGContextTranslateCTM(context,

```

```
    targetPoint.x, targetPoint.y);

    // Rotate along the slope
    CGContextRotateCTM(context, angle);

    // Adjust for the character size
    CGContextTranslateCTM(context,
        -bounding.size.width / 2,
        -item.size.height / 2);

    // Draw the character
    [item drawAtPoint:CGPointZero];
});

}

@end
```

Fitting Text

Several UIKit classes include size-to-fit options. However, there are no equivalent functions for general string drawing tasks. It's up to you to find a font that "fits" a string to a destination. Take rectangles, for example. Nearly any text will "fit" into a rectangle if its font is small enough. Unfortunately, that results in teeny-tiny single lines of text. The challenge is to find the biggest font that wraps to match a target, so the drawn text's aspect corresponds to the aspect of that rectangle. Figure 8-30 shows text strings of different lengths. Each drawing fits its destination by adjusting the font's size while testing for aspect.

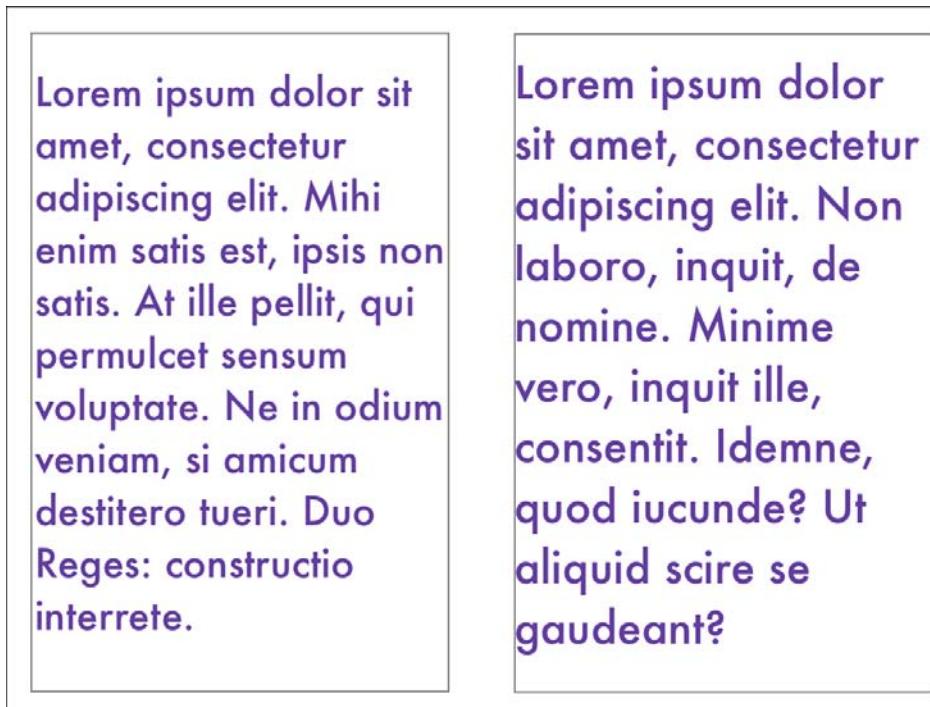


Figure 8-30 Font fitting.

Listing 8-4 details the algorithm I used to create the output in Figure 8-30. It's not a precise system because I've added a bit of flexibility through the `tolerance` parameter. Because of that, this implementation can sometimes slightly overshoot the destination. For that reason, consider calling this function with a slightly smaller destination than you actually intend to use.

This function iteratively selects a font. It tests its output size, and when the aspect of the output approaches the aspect of the destination rectangle, the algorithm stops and returns the most recently successful font.

As with all the other material in this chapter, Listing 8-4 has been updated to use iOS 7 calls. The new method that retrieves a bounding box for text cannot be deployed to pre-iOS 7 systems.

Listing 8-4 Choosing a Fitting Font

```
UIFont *FontForWrappedString(  
    NSString *string, NSString *fontFace,  
    CGRect rect, CGFloat tolerance)
```

```
{  
    if (rect.size.height < 1.0f) return nil;  
  
    CGFloat adjustedWidth = tolerance * rect.size.width;  
    CGSize measureSize =  
        CGSizeMake(adjustedWidth, CGFLOAT_MAX);  
  
    // Initialize the proposed font  
    CGFloat fontSize = 1;  
    UIFont *proposedFont =  
        [UIFont fontWithName:fontFace size:fontSize];  
  
    NSMutableParagraphStyle *paragraphStyle =  
        [[NSMutableParagraphStyle alloc] init];  
    paragraphStyle.lineBreakMode = NSLineBreakByWordWrapping;  
  
    NSMutableDictionary *attributes =  
        [NSMutableDictionary dictionary];  
    attributes[NSSParagraphStyleAttributeName] = paragraphStyle;  
    attributes[NSFontAttributeName] = proposedFont;  
  
    // Measure the target  
    CGSize targetSize =  
        [string boundingRectWithSize:measureSize  
            options:NSStringDrawingUsesLineFragmentOrigin  
            attributes:attributes context:nil].size;  
  
    // Double until the size is exceeded  
    while (targetSize.height <= rect.size.height)  
    {  
        // Establish a new proposed font  
        fontSize *= 2;  
        proposedFont =  
            [UIFont fontWithName:fontFace size:fontSize];  
  
        // Measure the target  
        attributes[NSFontAttributeName] = proposedFont;  
        targetSize =  
            [string boundingRectWithSize:measureSize  
                options:NSStringDrawingUsesLineFragmentOrigin  
                attributes:attributes context:nil].size;  
  
        // Break when the calculated height is too much  
        if (targetSize.height > rect.size.height)  
            break;  
    }  
}
```

```
// Search between the previous and current font sizes
CGFloat minFontSize = fontSize / 2;
CGFloat maxFontSize = fontSize;
while (1)
{
    // Get the midpoint between the two
    CGFloat midPoint = (minFontSize +
        (maxFontSize - minFontSize) / 2);
    proposedFont =
        [UIFont fontWithName:fontFace size:midPoint];
    attributes[NSFontAttributeName] = proposedFont;
    targetSize =
        [string boundingRectWithSize:measureSize
            options:NSStringDrawingUsesLineFragmentOrigin
            attributes:attributes context:nil].size;

    // Look up one font size
    UIFont *nextFont =
        [UIFont fontWithName:fontFace size:midPoint + 1];
    attributes[NSFontAttributeName] = nextFont;
    CGSize nextTargetSize =
        [string boundingRectWithSize:measureSize
            options:NSStringDrawingUsesLineFragmentOrigin
            attributes:attributes context:nil].size;;

    // Test both fonts
    CGFloat tooBig = targetSize.height > rect.size.height;
    CGFloat nextIsTooBig =
        nextTargetSize.height > rect.size.height;

    // If the current is sized right
    // but the next is too big, it's a win
    if (!tooBig && nextIsTooBig)
        return [UIFont fontWithName:fontFace size:midPoint];

    // Adjust the search space
    if (tooBig)
        maxFontSize = midPoint;
    else
        minFontSize = midPoint;
}

// Should never get here
return [UIFont fontWithName:fontFace size:fontSize / 2];
}
```

Summary

This chapter explores challenges you encounter when drawing text in iOS. You read about attributed strings and their features, as well as how to draw strings into paths using Core Text. Here are a few final thoughts about this topic:

- This is not the original chapter I wrote for this book. After iOS 7 debuted, my editorial team and I made the judgment call that updated material would be far more valuable to you as a reader. For that reason, I revised all the material to exclude deprecated calls such as `sizeWithFont:` and `drawInRect:withFont:`. This means that the vast majority of this chapter will not run directly on iOS 6 and earlier installs without retrofitting. The road back to iOS 6 or earlier isn't hard for many of these algorithms. Let Xcode be your guide in returning from “with attribute” calls to “with font” calls and so forth.
- I developed the ancestors of the fitting algorithm you see in Listing 8-4 long before I came up with the `string-to-UIBezierPath` solutions you saw in Chapter 4. These days, when working with short strings, I'm far more likely to convert the string to a path and fit it to a rectangle. For long strings with wrapping, however, Listing 8-4 remains my go-to.
- Don't forget the centered string drawing solution introduced in Chapter 2. It provides an easy way to paint a string into the middle of a destination rectangle or onto shapes drawn into that rectangle's bounds.

This page intentionally left blank

A

Blend Modes

Blend modes tell a context how to apply new content to itself. They determine how pixel data is digitally blended. For example, the default blend mode simply adds new material on top of whatever data is already in the context. You can do much more, however. Blend modes allow you to emulate Photoshop-style layer effects. You'll find familiar operations like multiply, color dodge, hard light, and difference among the available modes.

Listing A-1 demonstrates how you set a context's blending mode. You call `CGContextSetBlendMode()` with the current context, specifying which mode you wish to use. These modes, which are enumerated as `CGBitmapMode`, are detailed in Table A-1.

Listing A-1 generated the first of each figure pair in Table A-1. This listing draws a green circle to the context and then sets a blend mode. That blend mode applies to the purple circle, which is drawn next. The result of that operation depends on the blend mode that was specified. The second image in each pair draws a circle over a photograph. This supplies a more nuanced expression of each blend mode. It uses the same approach as Listing A-1 but replaces the green circle with an image-drawing command.

Listing A-1 Applying a Blend Mode

```
// Draw the first shape
[greenColor set];
[shape1 fill];

// Set the blending mode
CGContextSetBlendMode(
    UIGraphicsGetCurrentContext(), theMode);

// Draw the second shape
[purpleColor set];
[shape2 fill];
```

One of the reasons there are so many blend modes listed in Table A-1 is that the first half of the list is older modes. The normal (`kCGBlendModeNormal`) through luminosity (`kCGBlendModeLuminosity`) modes first appeared in OS X Tiger. The Porter-Duff blend modes that follow didn't debut until OS X Leopard. These consist of the clear (`kCGBlendModeClear`) through plus-lighter (`kCGBlendModePlusLighter`) blends. Every mode in Table A-1 is available in iOS.

The Porter-Duff collection is annotated with mathematical equations, which refer to R, S, and D. The result (R) of any blend operation represents the result of blending the source pixel values (S), the source alpha levels (Sa), the destination pixel values (D), and the destination alpha levels (Da).

Apple writes in the Quartz documentation:

Note that the Porter-Duff blend modes are not necessarily supported in every context. In particular, they are only guaranteed to work in bitmap-based contexts, such as those created by `CGBitmapContextCreate`. It is your responsibility to make sure that they do what you want when you use them in a `CGContext`.... Note that the Porter-Duff “XOR” mode is only titularly related to the classical bitmap XOR operation which is unsupported by CoreGraphics.

For example, here is the equation for the normal blend mode:

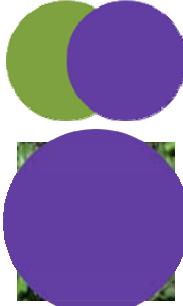
$$R = (Sa \times S) + ((1 - Sa) \times D)$$

The result consists of the source pixels multiplied by their alpha level, added to the remaining alpha capacity, multiplied by destination pixels.

If the source is entirely opaque in normal blending, the result entirely represents the source color. If the source is half transparent, the result is created half from the source and half from whatever material has already been added to the context at that point.

Each blend mode, regardless of when it was added to iOS or OS X or how it is documented, creates a rule that expresses the relationship between the source material (the pixels being applied) and the destination (the pixels already stored to the context).

Table A-1 Blend Modes

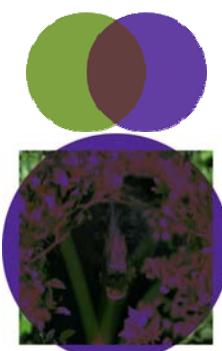
Mode	Result	Description
kCGBlendModeNormal		The new (source) drawing is painted on top of any existing (destination) material. $(R = (Sa \times S + (1 - Sa) \times D))$
kCGBlendModeMultiply		This mode multiplies source samples with the destination. Colors are at least as dark as either the source or destination pixels.
kCGBlendModeScreen		This mode multiplies the inverse of the source samples by the inverse of the destination samples. The results are at least as light as either the source or destination pixels.

`kCGBlendModeOverlay`



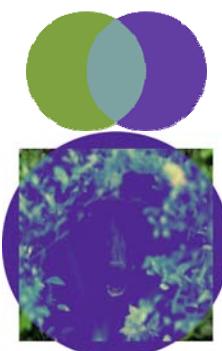
Depending on the background color, this mode multiplies or screens the source samples with the destination samples. Apple docs explain “The result is to overlay the existing image samples while preserving the highlights and shadows of the background. The background color mixes with the source image to reflect the lightness or darkness of the background.”

`kCGBlendModeDarken`



This mode chooses the darker of the source and destination samples and replaces destination samples with any source image samples that are darker.

`kCGBlendModeLighten`



This mode chooses the lighter of the source and destination samples and replaces destination samples with source image samples that are lighter.

kCGBlendModeColorDodge



This mode brightens destination samples to reflect source image samples. No change to 0% (black) source values.

In traditional photography, dodging refers to underexposing a print, producing lighter results.

kCGBlendModeColorBurn



This mode darkens destination samples to reflect source image samples. No changes to 100% (white) source values.

In traditional photography, burning would lengthen exposure to light, producing a darker print.

kCGBlendModeSoftLight



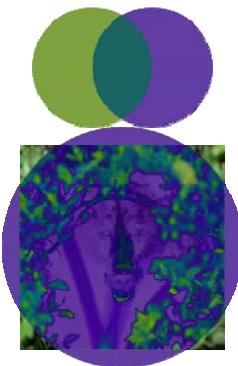
This mode darkens or lightens colors, depending on the source samples. If the source is under 50% gray, it dodges. If over 50% gray, it burns. At 50% gray exactly, the source is unchanged. Apple docs add, “Image samples that are equal to pure black or pure white produce darker or lighter areas, but do not result in pure black or white.... Use this to add highlights to a scene.” This effect is similar to shining a diffuse spotlight on the source image.

kCGBlendModeHardLight



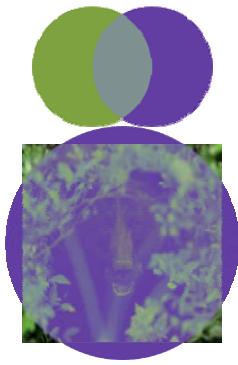
Similar to the soft-light blending mode, this effect is like shining a harsh spotlight on the source image. If the source is under 50% gray, it screens. If over 50%, it multiplies. At 50%, the source is unchanged. Pure black and white result in pure black and white.

kCGBlendModeDifference



This mode subtracts the lesser brightness (source or destination) from the greater brightness value. Black source samples produce no change, and white samples invert the background color.

kCGBlendModeExclusion



This mode is similar to kCGBlendModeDifference but with lower contrast.

kCGBlendModeHue



The source hue (here, purple) blends with the luminance and saturation of the destination.

kCGBlendModeSaturation



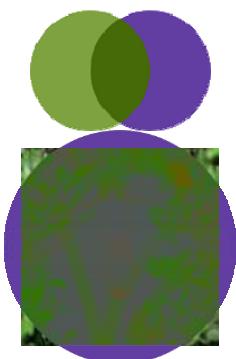
The destination hue (here, green) and luminance blend with the source saturation.

kCGBlendModeColor



The output equals the destination luminance values plus the hue and saturations of the source (here, purple). This mode is normally used to color monochrome images or add a tint to color images.

`kCGBlendModeLuminosity`



The output equals the destination hue and saturation (here, green) combined with the luminance of the source image. This produces the inverse effect of the `kCGBlendModeColor` mode.

`kCGBlendModeClear`

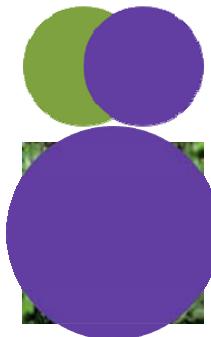


This mode causes source pixels to be cleared from the destination.

($R = 0$)



`kCGBlendModeCopy`



This mode causes source pixels to be drawn onto the destination.

($R = S$)

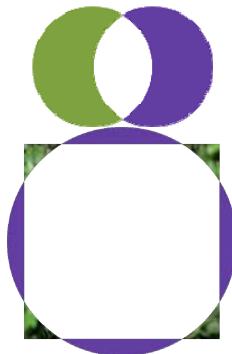
`kCGBlendModeSourceIn`



Source pixels are drawn
on destination pixels.

($R = S \times Da$, where Da
is the alpha component
of the destination)

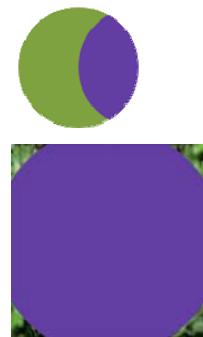
`kCGBlendModeSourceOut`



Source pixels are drawn
on pixels, not in the
destination.

($R = S \times (1 - Da)$)

`kCGBlendModeSourceAtop`



Source pixels are drawn
onto pixels in the
destination.

($R = S \times Da + D \times
(1 - Sa)$)

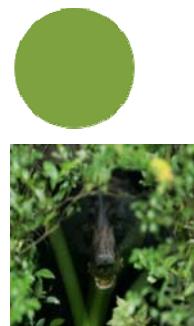
kCGBlendModeDestination
Over



Source pixels are drawn
onto pixels, not in
destination.

$$(R = S \times (1 - Da) + D)$$

kCGBlendModeDestination
In



Source pixels use the
destination value
multiplied by the source
alpha level.

$$(R = D \times Sa)$$

kCGBlendModeDestination
Out



Source pixels are
essentially removed
from the destination.

$$(R = D \times (1 - Sa))$$



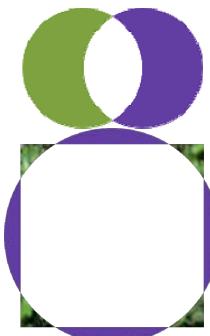
kCGBlendModeDestination
Atop



Source pixels are drawn outside the destination.

$$(R = S \times Da + D \times (1 - Sa))$$

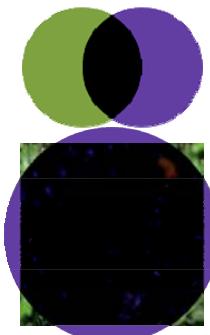
kCGBlendModeXOR



Pixels appearing in both source and destination are removed.

$$(R = S \times (1 - Da) + D \times (1 - Sa))$$

kCGBlendModePlusDarker



Overlap is black.

$$(R = \text{MAX}(0, 1 - ((1 - D) + (1 - S))))$$

kCGBlendModePlusLighter

Overlap is white.

$$(R = \min(1, S + D))$$



B

Miter Threshold Limits

Chapter 4 introduces miter limits and bevels. This appendix explores their iOS implementation with practical numbers. When the diagonal length of a miter—that is, the triangular join between two lines—exceeds a path’s limit, Quartz converts those points into bevel joins. The normal geometric distance for a miter can be calculated like this:

```
CGFloat diagonal = (lineWidth / 2.0f) / sin(theta / 2.0f);
```

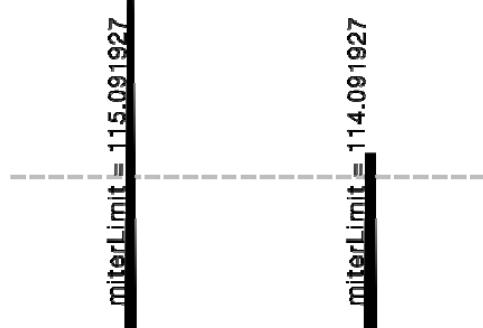
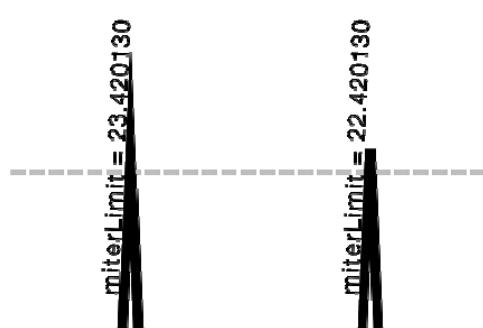
Quartz appears to use the value for a path of width 2 for all its cutoffs. Table B-1 details the resulting threshold values for angles between 1 and 45 degrees.

For example, for a Bezier path including an 8-degree angle, a miter limit of 14.3 will affect that path, but a miter limit of 14.4 will not. `UIBezierPath`’s default limit is 10, which will limit the miter for any angle under 11 degrees.

Table B-1 includes several sample images. These samples show an angle at the specified degree with the miter limit set just above (left) or just below (right) the critical level for that angle. What you see is the natural angle with and without a miter.

At 1 degree, the natural miter extends out about 114.6 points; in Table B-1, the first point is actually cropped out of the image due to this length. At 45 degrees, a miter extends just 2.6 points, barely rising above the alternative miter.

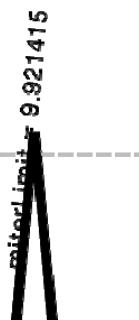
Table B-1 Miter Limits

Degrees	Radians	Threshold	Example
1°	0.017	114.593	
2°	0.035	57.2987	
3°	0.052	38.2015	
4°	0.070	28.6537	
5°	0.087	22.9256	
6°	0.105	19.1073	
7°	0.122	16.3804	
8°	0.140	14.3356	
9°	0.157	12.7455	

10°	0.175	11.4737
11°	0.192	10.4334



12°	0.209	9.5668
13°	0.227	8.8337
14°	0.244	8.2055
15°	0.262	7.6613



16°	0.279	7.1853
17°	0.297	6.7655
18°	0.314	6.3925
19°	0.332	6.0589



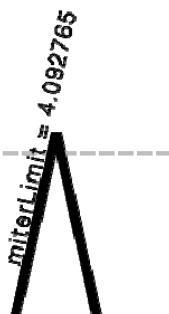
20° 0.349 5.7588



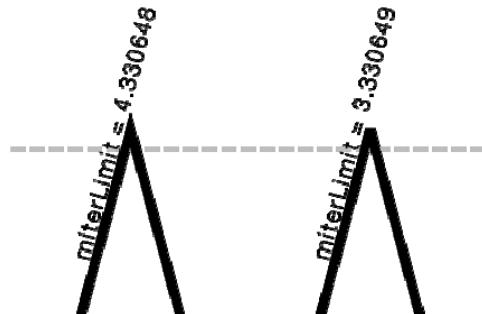
21° 0.367 5.4874
 22° 0.384 5.2408
 23° 0.401 5.0159
 24° 0.419 4.8097
 25° 0.436 4.6202



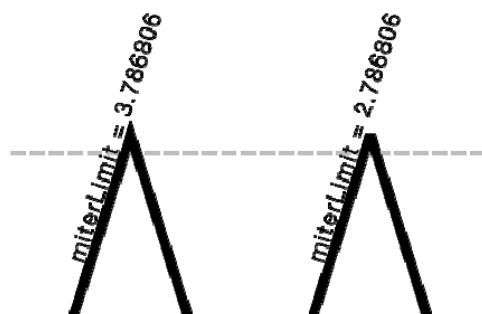
26° 0.454 4.4454
 27° 0.471 4.2837
 28° 0.489 4.1336
 29° 0.506 3.9939



30° 0.524 3.8637



31° 0.541 3.742
 32° 0.559 3.628
 33° 0.576 3.5209
 34° 0.593 3.4203
 35° 0.611 3.3255



36° 0.628 3.2361
 37° 0.646 3.1515
 38° 0.663 3.0716
 39° 0.681 2.9957

40° 0.698 2.9238

41° 0.716 2.8555
 42° 0.733 2.7904
 43° 0.750 2.7285
 44° 0.768 2.6695
 45° 0.785 2.6131

