Erica Sadun

The Gourmet iOS Developer's Cookbook

Even More Recipes for Better iOS App Development

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Gourmet iOS Developer's Cookbook

This page intentionally left blank

The Gourmet iOS Developer's Cookbook

Even More Recipes for Better iOS App Development

Erica Sadun

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
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 Gourmet iOS Developer's Cookbook: Even More Recipes for Better iOS App Development is an independent publication and has not been authorized, sponsored, or otherwise approved by Apple Inc.

Apple, the Apple logo, Apple TV, Apple Watch, Cocoa, Cocoa Touch, eMac, FaceTime, Finder, iBook, iBooks, iCal, Instruments, iPad, iPad Air, iPad mini, iPhone, iPhoto, iTunes, the iTunes logo, iWork, Keychain, Launchpad, Lightning, LocalTalk, Mac, the Mac logo, MacApp, MacBook, MacBook Air, MacBook Pro, MacDNS, Macintosh, Mac OS, Mac Pro, MacTCP, the Made for iPad logo, the Made for iPhone logo, the Made for iPod logo, Metal, the Metal logo, the Monaco computer font, Multi-Touch, the New York computer font, Objective-C, OpenCL, OS X, Passbook, Pixlet, PowerBook, Power Mac, Quartz, QuickDraw, QuickTime, the QuickTime logo, Retina, Safari, the Sand computer font, Shake, Siri, the Skia computer font, Swift, the Swift Logo, the Textile computer font, Touch ID, TrueType, WebObjects, WebScript, and Xcode are trademarks of Apple, Inc., registered in the United States and other countries. OpenGL and the logo are registered trademarks 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.

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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2015935369

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. 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, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-408622-4
ISBN-10: 0-13-408622-8

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: May 2015

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

Indexer
Tim Wright

Proofreader
Sarah Kearns

Technical Reviewers
Mark Granoff
Mike Greiner
Rich Wardwell

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

Dedicated with great affection to Chris Zahn:
editor, enabler, and wonderful person.

Contents

Preface xiii

1 Media 1

Speech 1

- Other Options 3
- Delegate Callbacks 3
- Dictation 5

Creating Barcodes 5

- Filter Parameters 5
- Building Codes 6

Reading Barcodes 8

- Listening for Metadata Objects 10
- Responding to Metadata 11
- Extracting Bounds 13
- Enhancing Recognition 14
- Detecting Faces 14

Building AVFoundation Movies 14

- Creating a Pixel Buffer 16
- Drawing into the Pixel Buffer 17
- Expressive Drawing 18
- Building Movies from Frames 19
- Adding Images to Movies 23

Wrap-up 24

2 Dynamic Typography 25

Type Size and User Needs 25

- How Dynamic Type Works 25
- Listening for Type Updates 28

Handling Dynamic Type with Attributes 31

- Scanning for Text Style Ranges 32
- Applying Text Style Ranges 34

Attribute-Ready Dynamic Elements 35

- Custom Fonts Faces 36
- Dynamic Text Views 37

Custom Sizing 38

Font Descriptors 39
Descriptor Challenges 40
Fonts with Multiple Variations 41
Using String Attributes to Modify Fonts 42
Dynamic Type Gotchas 43
Wrap-up 43

3 Text Kit 45
Creating Complex Text Layouts 45
Glyphs 46
Text Storage 55
Layout Managers 56
Text Containers 56
Adaptive Flow 58
Insets 60
Exclusion Paths 60
Bounding Rectangles 62
Using Text Kit to Add Touch to Labels 63
Establishing Text Kit 63
Synchronizing 64
Translating Coordinates 65
Glyphs and Characters 66
Checking for Links 67
Adding Visual Feedback 67
Draggable Exclusion Zones 69
Building PDFs with Text Kit 71
Printing Text View Data 73
Printing PDF Data 74
Wrap-up 74

4 Attributed Strings and Document Containers 75
Class Enhancements 75
String Attachments 77
Building Attributed Strings from HTML 78
Document Type Dictionaries 79
Converting HTML Source to Attributed Strings 80

Converting Attributed Strings to Document Representations 81
Generating HTML from Attributed Strings 82
Markup Initialization 83
RTF and RTFD 83
The RTFD Container 84
Initializing Attributed Strings from a File 84
Converting RTFD Text to Data 85
Writing RTFD Containers from Data 86
Inspecting Attributes 87
Establishing Document Attributes 89
Enhancing Attributed Strings 91
Returning Copies with New Attributes 92
Adjusting Attributes 93
Extending Mutable Attributed Strings 94
Text Ranges 95
Calculating Positions 95
Position Geometry 95
Updating Selection Points 97
Hardware Key Support 97
Wrap-up 99

5 Animation 101
Keyframe Animation 101
Building Physics with Keyframes 103
Blocking Animators 105
UIKit Spring-Based Animations 106
Practical Uses for Spring Animations 108
System Animations 109
Motion Effects 109
Building Planes 110
Shadow Effects 111
Custom Transition Animations 113
Delegation 114
Building Transitioning Objects 114

Implicit Animations 116
Building an Animation-Ready Layer 116
Building a View Around a Layer 118
Timing 118
Coordinating Animations 119
Building Implicit Completion Blocks 120
Animating Custom Properties 121
Intercepting Updates 122
Drawing Properties 123
Wrap-up 124

6 Dynamic Animators 125
Physics-Based Behaviors 125
Building Dynamics 126
Detecting Pauses 127
Creating a Frame-Watching Dynamic Behavior 131
Implementing Snap Zones 133
Leveraging Real-World Physics 135
Connecting a Gravity Behavior to Device Acceleration 137
Creating Boundaries 138
Enhancing View Dynamics 138
Custom Behaviors 139
Creating Custom Dynamic Items 139
Subverting Dynamic Behaviors 141
Better Custom Dynamic Behaviors 142
Custom Secondary Behaviors 144
Collection Views and Dynamic Animators 147
Custom Flow Layouts 147
Returning Layout Attributes 148
Updating Behaviors 149
Building a Dynamic Alert View 150
Connecting Up the Jelly 150
Drawing the View 152
Deploying Jelly 154
Wrap-up 154

7 Presentations 155
Alerts 155
Class Deprecations 155
Building Alerts 156
Enabling and Disabling Alert Buttons 161
Adding Text Fields 162
Mask Views 164
Shape Layer Masking 164
Building Mask Views 166
Building Effect Views 169
Building a Blur Effect 170
Adding Vibrancy Effects 171
Animating Effect Views 172
Building Popovers 175
Supporting Bubbles 176
Presenting Popovers 177
Wrap-up 177

8 Shape Magic 179
How to Shape a View 179
Expanding Beyond Circles 180
Resizing Bezier Paths 180
Building a Bezier-Based Shape Image View 184
Working with Unclosed Shapes 185
Adding Borders to Shaped Views 187
Building Shaped Buttons 190
Adding Attention-Grabbing Animations to Shaped Views 193
Wrap-up 199

9 Adaptive Deployment 201
Traits 201
Trait Properties 202
Defining Traits 202
Combining Trait Collections 203
Designing for Traits 204

`UIScreen` Properties 205
Coordinate Spaces 205
Application Frame 206
Screen Bounds 206
Scale 207
Rotation 207
Size Classes and Assets 208
Basic Deployment 208
UIKit and Image Views 210
The `UIImageAsset` Class 210
Building Images from PDFs 211
Overriding Trait Collections 214
Building Side-by-Side iPhone Split Views 215
A Bit More About iOS 8 Split View Controllers 218
Wrap-up 219

10 Development Helpers 221
All the Lorems 221
Placeholder Text 221
Image Ipsums 223
Generating Random User Data 225
Bulk Names 225
Generating Random Feeds 227
Random Everything 228
Directives 229
Converting Comments to Warnings 229
Warnings 231
Testing for the Simulator 232
Errors 232
Testing for Inclusion 233
Messages 234
Wrapping Pragmas 234
Overriding Diagnostics 235
Unused Variable Warnings 235
Marking Non-null and Nullable Items 236

Developer Tweaks 236
Saving Files from the Simulator 237
Tighter Logging 238
Wrap-up 238

11 A Taste of Swift 239
Swift Versus Objective-C 239
Building iOS Apps in Swift 240
Optionals 243
Inferred Types 244
The Optional Enumeration 245
Unwrapping Optionals 246
Assigning Values to Non-optionals 248
Cocoa Touch Patterns 248
Hybrid Language Development 251
Calling Objective-C from Swift 252
Accessing Classes 252
Calling Swift from Objective-C 253
Preparing Swift for Objective-C 254
Class Descent 255
Building the Basics 256
Watching Progress 257
Learning Swift 259
Wrap-up 260

Index 261

Preface

Developers can never have too many useful ideas to draw from, and this latest entry in the bestselling *Cookbook* series is filled with delicious possibilities. *The Gourmet iOS Developer's Cookbook* offers a curated selection of programming recipes to inspire your everyday iOS programming efforts. This volume serves up a new banquet of turnkey solutions for projects big and small. It offers a fresh collection of versatile solutions that promise to add spice to your code.

The goal here is simple. Each chapter should enable you to walk away with fresh ideas and master techniques off the beaten track. Whether you're reading about new takes on old technologies or completely fresh APIs, here's hoping you'll say, "Hey, I didn't know you could do that!" or "That's really cool."

The Gourmet iOS Developer's Cookbook offers a deep dive into the nonobvious. Its chapters cover techniques and technologies that skew away from the common and enable you to explore new development cuisines. It's not a book for those just learning how to cook apps. It offers tasty recipes for the iOS enthusiast who wants to builds fragrant, delicious, and exotic routines.

How This Book Is Organized

This book offers practical iOS development recipes. Here's a rundown of what you'll find in this book's chapters:

- **Chapter 1, "Media"**—This chapter explores advances that have made their way into AVFoundation over the past few years and shows how you can integrate these features into your own applications. In this chapter, you'll read about speech generation, barcode recognition (which enables you to leverage the device camera to recognize a wide range of barcode styles), and application of modern language features to AVFoundation movie creation.
- **Chapter 2, "Dynamic Typography"**—iOS's overhauled interface has shifted emphasis away from buttons and bars to a sparser and more text-centered experience, where text components have become even more critical parts of UI design. This chapter introduces ways your text can update itself automatically to match user preferences and expectations and discusses some critical lessons to be learned along the way.
- **Chapter 3, "Text Kit"**—Flexible text presentation is one of the most exciting and developing areas of iOS. With every new iOS release, these APIs have grown, matured, and expanded. Most UIKit interface classes now support rich text features. In the most modern iOS releases, that support has expanded to a suite of layout classes that continue to add mature type and frame settings to create flexible presentations.
- **Chapter 4, "Attributed Strings and Document Containers"**—Over the past few years, attributed strings have grown enormously in power and potential, and they now provide support for HTML and RTF rich text documents. Attributed strings provide seamless polymorphism between text presentation and representation. Text design now better migrates to the iOS screen and from iOS to other destinations. This chapter explores those expanded possibilities.

- **Chapter 5, "Animation"**—Of the technologies updated in the past couple years, iOS animation is one of the ones that has been most enhanced by new APIs. New dynamic styles enable your interfaces to integrate real-world physics for better and more exciting presentations and interactions. This chapter begins the discussion of animation features, introducing some of the profound updates that you'll use in your apps.
- **Chapter 6, "Dynamic Animators"**—Dynamic animators are some of the most exciting elements of iOS. Their physics-based view behaviors create lively and curious interfaces. At the same time, they can be difficult to work with. In this chapter, you'll learn how to incorporate these classes into your iOS apps for the best possible results and the fewest headaches.
- **Chapter 7, "Presentations"**—In the latest versions of iOS, user alerts are fully re-imagined and popovers are now universally available. Special effects highlight presentations to provide the greatest visual impact when you overlay content for modal interaction. This chapter gets you up to speed on these modern techniques.
- **Chapter 8, "Shape Magic"**—Non-rectangular views enable your apps to expand possibilities with fun and clever effects. For example, you might draw attention to a view by animating a halo behind it. Or you might use shapes to better stack buttons together for visual seamlessness. This chapter covers many advanced shape techniques you can use to add pizzazz to your user interfaces.
- **Chapter 9, "Adaptive Deployment"**—As the iOS family continues to grow, apps should automatically support all new displays, orientations, and screens. Although iOS targets are not nearly as splintered as Android's multitude, interfaces face numerous configurations for universal deployment. A truly adaptive app gracefully responds with a well-designed and engaging interface, ready for the user at any size. This chapter explores the basics of these new technologies and the APIs you need to learn for moving your apps forward.
- **Chapter 10, "Development Helpers"**—At times, it helps to have methods, functions, and techniques to help you through the development process. Together, the solutions in this chapter support you when building apps. They enable you to speed through your development day to better arrive at the app you're working on.
- **Chapter 11, "A Taste of Swift"**—Apple introduced the Swift programming language at the June 2014 WWDC Keynote. Swift offers a performance-tuned type-safe modern programming language. Today, many development fundamentals have coalesced, although the language and toolset have continued to evolve. This chapter surveys the base essentials of Swift development, providing a taste of this new technology. You won't learn the language in this chapter. Instead, you'll explore concepts and development issues that affect you as an iOS developer to get a sense of where this important technology is going.

About the Sample Code

This book follows the trend I started in my *iOS Developer's Cookbook* series. This book's 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—nor how they should be developing them. It's hard to tell a story when readers must search through many files and try to find out what is relevant and what is not. Offering a single launching point concentrates the story, allowing access to an idea from a coherent starting point.

Getting the Sample Code

You'll find the source code for this book at https://github.com/erica/iOS-Gourmet-Cookbook on the open-source GitHub hosting site. There, you'll find a chapter-by-chapter collection of source code that provides examples of the material covered in this book.

Retrieve sample code either by using git tools to clone the repository or by clicking GitHub's Download button, which was at the right center of the page when I wrote this book. It enables you to retrieve the entire repository as a ZIP archive or tarball.

Contribute!

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. Pitch in by suggesting bug fixes and corrections and by expanding the code that's on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features and then share them back to the main repository. If you come up with a new idea or approach, let me know.

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 among project developers, it's a great place to find new code or collaborate on existing libraries. Sign up for a free account at the GitHub website, where you can then copy and modify this 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.

Acknowledgments

My sincere thanks go out to Trina MacDonald, Chris Zahn, and Olivia Basegio, along with the entire Addison-Wesley/Pearson production team—specifically Kristy Hart, Betsy Gratner, Kitty Wilson, Nonie Ratcliff, and Chuti Prasertsith—and my technical editors Rich Wardwell, Mark Granoff, and Mike Greiner.

My gratitude extends to everyone who helped read through early drafts and provide feedback. Specific thanks go out to Oliver Drobnik, Hamish Allan, Sebastian Celis, Maurice Sharp, Wess Cope, Jeremy Tregunna, Ken Lindsay, Cameron Banga, John Grosvenor, Matthias Neeracher, Chris Woodard, David Green, Alexander Kempgen, Chris Flesner, Remy "psy" Demarest, Ken Ferry, Mike Ash, Kevin Ballard, Phil Holland, August Joki, and everyone else who contributed to this effort. If I have omitted your name here, please accept my apologies.

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 has blogged at TUAW.com, 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 domination.

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

6

Dynamic Animators

Dynamic animators are some of the most exciting elements of iOS, even if they are among the least practical. Their physics-based view behaviors create lively and curious interfaces. At the same time, they can be fussy to work with. They don't happily coexist with Auto Layout because they directly update frame values and can rotate views. That said, dynamic animators are tremendously fun. They help make your UIs pop and are well worth exploring to discover what features they can provide your users.

Physics-Based Behaviors

The `UIDynamicAnimator` class emulates interface "physics." It coalesces this functionality into distinct behaviors like snapping, pushing, attachment, and collision. Here's a quick overview of the primitive UIKit dynamic behaviors:

- **Attachments—**`UIAttachmentBehavior` instances tie a view either to a position or to another view. It's basically a virtual string with a set length, although you can make it act more like a spring by updating damping and frequency properties.
- **Collisions—**`UICollisionBehavior` instances allow views to collide with each other or with path-based boundaries. In a collision, energy can be passed from one item to another, and a view's trajectory can be changed.
- **Gravity—**`UIGravityBehavior` instances apply acceleration to views. You set where "down" is and allow the gravity vector to act on velocities over time.
- **Pushes—**`UIPushBehavior` instances add an impulse force to views, adding new energy to the system.
- **Snaps—**`UISnapBehavior` instances act as magnets, drawing views to attachment points.
- **Dynamic items—**`UIDynamicItemBehavior` is the odd man out in this list. Instead of acting as a force, dynamic items are objects affected by forces. These behaviors enable your views to participate in the other behaviors listed here. You can attach, collide, push,

snap, and weigh down views by treating them as having physical properties.
The dynamic item behavior defines density, elasticity, friction, and resistance and manages linear and angular item velocities.

You can best explore how these items work by running Apple's UIKit Dynamic Catalog sample code (https://developer.apple.com/library/ios/samplecode/DynamicsCatalog). This sample code illustrates available dynamic behaviors, presenting a wide range of effects you can create in your own apps. Most importantly, it lets you see, interact with, and explore each behavior on its own.

Building Dynamics

Once you've finished exploring Apple's dynamics catalog, start building your own examples. To begin, you need to create a dynamic animator, like this:

```
self.animator = [[UIDynamicAnimator alloc]
    initWithReferenceView:self.view];
```

This top-level class acts as an intermediary between your views and any dynamic behaviors you add to the system. The animator provides context for the animations, establishing either a reference view to create a coordinate system or a reference layout when working with collection views.

Typically, you use a view controller's primary view as a reference, although you are not limited to this. Use any view backdrop that's large enough to contain the actors in your drama. And, as you'll see, you can extend animated views beyond the parent view, if needed.

Dynamics Delegation

Delegation enables you to know when an animator pauses, an important tool for tracking the end of an animation sequence. An animator delegate declares the `UIDynamicAnimatorDelegate` protocol and conforms to that protocol by implementing the optional `dynamicAnimatorDidPause:` and `dynamicAnimatorWillResume:` methods. Assign a delegate like this:

```
self.animator.delegate = self;
```

When you implement a delegate, you know when animation sequences coalesce, which enables you to clean up your simulation after the physics have come to a static resting point. Be aware that some animations may never "stop," especially those that do not employ energy-lowering strategies like friction and resistance.

Creating and Adding Behaviors

Each dynamic animator can coordinate many behaviors at once. For example, you might want to create a dynamic system where views "fall" in the direction of gravity but bounce off each

other and remain within the boundaries of the view controller's view. Or you might create a snapping behavior that involves collision detection, bumping some views out of the way.

Add each behavior to the animator with the `addBehavior:` method. This method applies the behavior to the current state. If the animator is active, the behavior will immediately start. The following snippet creates a new snapping behavior and adds it to an animator:

```
UISnapBehavior *snapBehavior = [[UISnapBehavior alloc]
    initWithItem:testView snapToPoint:point];
[self.animator addBehavior:snapBehavior];
```

The standard behavior-creation pattern is to allocate an instance and initialize it with one or more items. This example uses a single item (`testView`) and sets a single parameter, a snap-to point. When this is added to the animator, the view moves until its center co-aligns with the snap point.

Each dynamic behavior is distinct in terms of the details associated with the class's API. Gravity behavior initializers accept an array of child items, although you can add and remove items at later times. Attachment behaviors include a suite of initializers that supply anchor points, dynamic items, and offsets away from the anchors. Each behavior class is a new adventure, and it's well worth your time to read through their APIs as they are all quite different from each other.

Detecting Pauses

Behavior lifetimes vary. After adding a behavior to an animator, you leave it in place for varying degrees of time: until some application state has changed, until the animation has come to a stopping point (or has reasonably coalesced to the point where the user perceives it as having stopped), or until the application ends. The lifetime you select depends on the kind of behavior you define. For example, a collision behavior that keeps views inside a parent view controller may persist indefinitely. You might remove a snap behavior as soon as the view has moved to the newly requested position or a push behavior as soon as the impulse has finished.

The problem is, however, that the built-in dynamic animator can take a long time to detect that the views it manages have stopped moving. Consider the following list of times and frames for a snapped view:

```
[0.03] NSRect: {{121.55639, 217.55638}, {66.88723, 66.88723}}
[0.07] NSRect: {{91.418655, 206.41866}, {81.162689, 81.162689}}
[0.10] NSRect: {{60.333874, 201.33388}, {83.332253, 83.332253}}
[0.13] NSRect: {{44.293236, 204.29323}, {79.413528, 79.413528}}
[0.17] NSRect: {{42.394054, 213.39406}, {68.211891, 68.211891}}
[0.20] NSRect: {{44.46402, 221.46402}, {60.071957, 60.071957}}
[0.23] NSRect: {{44.94722, 222.94722}, {61.105556, 61.105556}}
[0.27] NSRect: {{47.207447, 223.70744}, {60.58511, 60.58511}}
[0.30] NSRect: {{49.458027, 223.45802}, {60.083942, 60.083942}}
[0.33] NSRect: {{50.481998, 222.48199}, {60.035999, 60.035999}}
```

```
[0.37] NSRect: {{50.987999, 221.98801}, {60.023998, 60.023998}}
[0.40] NSRect: {{51, 221.5}, {60, 60}}
[0.43] NSRect: {{50.5, 221.5}, {60, 60}}
[0.47] NSRect: {{50, 221.5}, {60, 60}}
[0.50] NSRect: {{50, 222}, {60, 60}}
[0.53] NSRect: {{50, 222}, {60, 60}}
[0.57] NSRect: {{50, 222}, {60, 60}}
...[snipped 0.60 to 1.10]...
[1.13] NSRect: {{50, 222}, {60, 60}}
[1.17] NSRect: {{50, 222}, {60, 60}}
Elapsed time: 1.167326
```

This view reaches its final position after half a second has passed. The dynamic animator does not pause until 1.17 seconds—more than double the required time. In user experience terms, those extra 0.67 seconds can feel like forever.

The reason for the delay becomes clear when you sneak down into the animator and look up the view's linear and angular velocity:

```
[0.60] NSRect: {{50, 222}, {60, 60}}
    Linear Velocity: NSPoint: {1.8314272, 1.0867469}
    Angular Velocity: 0.000001
```

Those values do not drop to 0 until that extra time has passed:

```
[1.17] NSRect: {{50, 222}, {60, 60}}
    Linear Velocity: NSPoint: {0, 0}
    Angular Velocity: 0.000000
```

In a practical sense, the velocities are meaningless once the view frame stops changing. When you know in advance that no outside forces will impel a view to start moving again after it's reached a resting point, leverage this information. Trim down your waiting time by tracking a view's frame.

Listing 6-1 defines a watcher class that monitors views until they stop changing. After a view has remained fixed for a certain period of time (here for at least 0.1 seconds), this class contacts a delegate and lets it know that the view has stopped moving. That callback enables you to update your dynamic animator and remove the behavior so the animator can more quickly come to a pause.

When run with the same snap animation as the previous example, the new watcher detects the final frame at 0.50. By 0.60, the delegate knows to stop the animation, and the entire sequence stops nearly 0.55 seconds earlier:

```
[0.47] NSRect: {{50, 221.5}, {60, 60}}
[0.50] NSRect: {{50, 222}, {60, 60}}
[0.53] NSRect: {{50, 222}, {60, 60}}
[0.57] NSRect: {{50, 222}, {60, 60}}
[0.60] NSRect: {{50, 222}, {60, 60}}
Elapsed time: 0.617352
```

Use this kind of short-cutting approach to re-enable GUI items that might otherwise be inaccessible to users once you know that the animation has come to a usable end point. While this example implements a pixel-level test, you might vary this approach to detect low angular velocities and other "close enough" tests to help end the animation effects within a reasonable amount of time.

Listing 6-1 **Watching Views**

```
// Info stores the most recent frame, count, delegate
@interface WatchedViewInfo : NSObject
@property (nonatomic) CGRect frame;
@property (nonatomic) NSUInteger count;
@property (nonatomic) CGFloat pointLaxity;
@property (nonatomic) id <ViewWatcherDelegate> delegate;
@end

@implementation WatchedViewInfo
@end

// Watcher class
@implementation ViewWatcher
{
    NSMutableDictionary *dict;
}

- (instancetype) init
{
    if (!(self = [super init])) return self;
    dict = [NSMutableDictionary dictionary];
    _pointLaxity = 10;
    return self;
}

// Determine whether two frames are "close enough"
BOOL CompareFrames(CGRect frame1, CGRect frame2, CGFloat laxity)
{
    if (CGRectEqualToRect(frame1, frame2)) return YES;
    CGRect intersection = CGRectIntersection(frame1, frame2);
    CGFloat testArea =
        intersection.size.width * intersection.size.height;
    CGFloat area1 = frame1.size.width * frame1.size.height;
    CGFloat area2 = frame2.size.width * frame2.size.height;
    return ((fabs(testArea - area1) < laxity) &&
            (fabs(testArea - area2) < laxity));
}
```

```
// See whether the view has stopped moving
- (void) checkInOnView: (NSTimer *) timer
{
    int kThreshold = 3; // must remain for 0.3 secs

    // Fetch the view and the info
    UIView *view = (UIView *) timer.userInfo;
    NSNumber *key = @((int)view);
    WatchedViewInfo *watchedViewInfo = dict[key];

    // Matching frame? If so update count
    BOOL steadyFrame = CompareFrames(watchedViewInfo.frame,
        view.frame, _pointLaxity);
    if (steadyFrame) watchedViewInfo.count++;

    // Threshold met
    if (steadyFrame && (watchedViewInfo.count > kThreshold))
    {
        [timer invalidate];
        [dict removeObjectForKey:key];
        [watchedViewInfo.delegate viewDidPause:view];
        return;
    }

    if (steadyFrame) return;

    // Replace frame with new frame
    watchedViewInfo.frame = view.frame;
    watchedViewInfo.count = 0;
}

- (void) startWatchingView: (UIView *) view
    withDelegate: (id <ViewWatcherDelegate>) delegate
{
    NSNumber *key = @((int)view);
    WatchedViewInfo *watchedViewInfo = [[WatchedViewInfo alloc] init];
    watchedViewInfo.frame = view.frame;
    watchedViewInfo.count = 1;
    watchedViewInfo.delegate = delegate;
    dict[key] = watchedViewInfo;

    [NSTimer scheduledTimerWithTimeInterval:0.03 target:self
        selector:@selector(checkInOnView:) userInfo:view repeats:YES];
}
@end
```

Creating a Frame-Watching Dynamic Behavior

While the solution in Listing 6-1 provides general view oversight, you can implement the frame checker in a much more intriguing form: as the custom dynamic behavior you see in Listing 6-2. This approach that adapts Listing 6-1 to a new form requires just a couple adjustments to work as a behavior:

- The behavior from the `checkInOnView:` method is now implemented in the behavior's `action` property. This block is called directly by the animator, using its own timing system, so the threshold is slightly higher in this implementation than in Listing 6-1.
- Instead of calling back to a delegate, this approach unloads both the watcher and the client behavior directly in the `action` block. This may be problematic if the behavior controls additional items, but for snap behaviors and their single items, it is a pretty safe approach.

To enable the watcher, you must add it to the animator as a separate behavior. Here's how you allocate it and initialize it with a client view and an affected behavior:

```
UISnapBehavior *snapBehavior = [[UISnapBehavior alloc]
    initWithItem:testView snapToPoint:p];
[self.animator addBehavior:snapBehavior];
WatcherBehavior *watcher = [[WatcherBehavior alloc]
    initWithView:testView behavior:snapBehavior];
[self.animator addBehavior:watcher];
```

Once it is added, it works just like Listing 6-1, iteratively checking the view's frame to wait for a steady state.

Listing 6-2 Watching Views with a Dynamic Behavior

```
// Create custom frame watcher
@interface WatcherBehavior : UIDynamicBehavior
- (instancetype) initWithView: (UIView *) view
    behavior: (UIDynamicBehavior *) behavior;
@property (nonatomic) CGFloat pointLaxity; // defaults to 10
@end

// Store the view, its most recent frame, and a count
@interface WatcherBehavior ()
@property (nonatomic) UIView *view;
@property (nonatomic) CGRect mostRecentFrame;
@property (nonatomic) NSInteger count;
@property (nonatomic) UIDynamicBehavior *customBehavior;
@end
```

```
@implementation WatcherBehavior
- (instancetype) initWithView: (UIView *) view
    behavior: (UIDynamicBehavior *) behavior
{
    if (!(self = [super init])) return self;

    // Initialize instance
    _view = view;
    _mostRecentFrame = _view.frame;
    _count = 0;
    _pointLaxity = 10;
    _customBehavior = behavior;

    // Create custom action for the behavior
    __weak typeof(self) weakSelf = self;
    self.action = ^{
        __strong typeof(self) strongSelf = weakSelf;
        UIView *view = strongSelf.view;

        CGRect currentFrame = view.frame;
        CGRect recentFrame = strongSelf.mostRecentFrame;
        BOOL steadyFrame = CompareFrames(currentFrame,
            recentFrame, strongSelf.pointLaxity);
        if (steadyFrame) strongSelf.count++;

        NSInteger kThreshold = 5;
        if (steadyFrame && (strongSelf.count > kThreshold))
        {
            [strongSelf.dynamicAnimator
                removeBehavior:strongSelf.customBehavior];
            [strongSelf.dynamicAnimator removeBehavior:strongSelf];
            return;
        }

        if (!steadyFrame)
        {
            strongSelf.mostRecentFrame = currentFrame;
            strongSelf.count = 0;
        }
    };

    return self;
}
@end
```

Implementing Snap Zones

One of my favorite dynamic animator tricks involves creating snap zones—areas of your interface that pull in dragged items once they overlap a particular region. This approach allows you to collect items into well-managed zones and offer a pleasing "snap-into-place" animation. In the general form shown in Listing 6-3, there's no further test beyond whether a dragged view has strayed into a zone. However, you might want to expand the approach to limit blue items to blue zones or red items to red zones, and so forth.

Listing 6-3 assumes that users will have access to multiple zones and even that a view might move from one zone directly to another. It uses a tagging scheme to keep track of this potential reparenting. A free view has no current parent and can move freely about. When a free view overlaps a snap zone, however, it suspends dragging by disabling the view's gesture recognizer and adds a snap-to-parent behavior. The view slides into place into its new parent. Once it arrives, as the dynamic animator pauses, the recognizer is re-enabled.

Allowing a view to escape from its new parent's bounds is the tricky bit—and the motivating reason for the view tagging. You do not want a view to recapture its child unless the dragging gesture has ended, which is why this method keeps track of the gesture state. With new parents, however, the snap behavior is added (and the gesture is suspended) as soon as a view strays over the line. Balancing the escapes and the captures ensures that the user experience is snappy and responsive and does not thwart the user's desires to remove a view from a parent.

Listing 6-3 **Handling Multiple Snap Zones**

```
- (void) draggableViewDidMove: (NSNotification *) note
{
    // Check for view participation
    UIView *draggedView = note.object;
    UIView *nca = [draggedView nearestCommonAncestorWithView:
        _animator.referenceView];
    if (!nca) return;

    // Retrieve state
    UIGestureRecognizer *recognizer = (UIGestureRecognizer *)
        draggedView.gestureRecognizers.lastObject;
    UIGestureRecognizerState state = [recognizer state];

    // View frame and current attachment
    CGRect draggedFrame = draggedView.frame;
    BOOL free = draggedView.tag == 0;

    for (UIView *dropZone in _dropZones)
    {
        // Make sure all drop zones are views
```

```
if (![dropZone isKindOfClass:[UIView class]])
    continue;

// Overlap?
CGRect dropFrame = dropZone.frame;
BOOL overlap = CGRectIntersectsRect(draggedFrame, dropFrame);

// Free moving
if (!overlap && free)
{
    continue;
}

// Newly captured
if (overlap && free)
{
    if (suspendedRecognizer)
    {
        NSLog(@"Error: attempting to suspend second recognizer");
        break;
    }

    // New parent.
    // CAPTURED is an integer offset for tagging
    suspendedRecognizer = recognizer;
    suspendedRecognizer.enabled = NO; // stop!
    draggedView.tag = CAPTURED + dropZone.tag; // mark as captured
    UISnapBehavior *behavior = [[UISnapBehavior alloc]
        initWithItem:draggedView
        snapToPoint:RectGetCenter(dropFrame)];
    [_animator addBehavior:behavior];
    break;
}

// Is this the current parent drop zone?
BOOL isParent = (dropZone.tag + CAPTURED == draggedView.tag);

// Current parent
if (overlap && isParent)
{
    switch (state)
    {
        case UIGestureRecognizerStateEnded:
        {
            // Recapture
            UISnapBehavior *behavior = [[UISnapBehavior alloc]
                initWithItem:draggedView
```

```
                    snapToPoint:RectGetCenter(dropFrame)];
                [_animator addBehavior:behavior];
                break;
            }
            default:
            {
                // Still captured but no op
                break;
            }
        }
        break;
    }

    // New parent
    if (overlap)
    {
        suspendedRecognizer = recognizer;
        suspendedRecognizer.enabled = NO; // stop!
        draggedView.tag = CAPTURED + dropZone.tag;
        UISnapBehavior *behavior = [[UISnapBehavior alloc]
            initWithItem:draggedView
            snapToPoint:RectGetCenter(dropFrame)];
        [_animator addBehavior:behavior];
        break;
    }
  }
}
```

Leveraging Real-World Physics

The built-in gravity dynamic animator consists of a downward force. You can adjust the force's vector to point gravity in other directions, but it's a static system. You can, however, integrate the gravity behavior with Core Motion to produce a much more satisfying effect. Apple's Core Motion framework enables your apps to receive motion-based data from device hardware, including the onboard accelerometer and gyroscope. The framework converts motion data into a form of input that your device can use to coordinate application changes with the way your user's device is held and moved over time.

Listing 6-4 builds a motion manager singleton. It uses Core Motion to listen for accelerometer updates, and when it receives them, it calculates a working vector and posts notifications with that information. You may be curious about that extra 0.5 added to the y component; it produces a more natural vector for holding a device in your hand.

Listing 6-4 **Broadcasting Motion Updates**

```
#define VALUE(struct) ({ __typeof__(struct) __struct = struct; \
    [NSValue valueWithBytes:&__struct \
    objCType:@encode(__typeof__(__struct))]; })

NSString *const MotionManagerUpdate = @"MotionManagerUpdate";
NSString *const MotionVectorKey = @"MotionVectorKey";

static MotionManager *sharedInstance = nil;

@interface MotionManager ()
@property (nonatomic, strong) CMMotionManager *motionManager;
@end

@implementation MotionManager
+ (instancetype) sharedInstance
{
    if (!sharedInstance)
        sharedInstance = [[self alloc] init];

    return sharedInstance;
}

- (void) shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
    [_motionManager stopAccelerometerUpdates];
    _motionManager = nil;
}

- (void) establishMotionManager
{
    if (_motionManager)
        [self shutDownMotionManager];

    // Establish the motion manager
    NSLog(@"Establishing motion manager");
    _motionManager = [[CMMotionManager alloc] init];
}

- (void) startMotionUpdates
{
    if (!_motionManager)
        [self establishMotionManager];
```

```
    if (_motionManager.accelerometerAvailable)
        [_motionManager
         startAccelerometerUpdatesToQueue:[[NSOperationQueue alloc] init]
         withHandler:^(CMAccelerometerData *data, NSError *error)
         {
             CGVector vector = CGVectorMake(data.acceleration.x, -
                 (data.acceleration.y + 0.5));
             NSDictionary *dict = @{MotionVectorKey:VALUE(vector)};
             [[NSNotificationCenter defaultCenter]
                 postNotificationName:MotionManagerUpdate
                 object:self userInfo:dict];
         }];

}
@end
```

Connecting a Gravity Behavior to Device Acceleration

On the other end of things, create an observer for motion updates. The following snippet builds a gravity behavior and updates its `gravityDirection` property whenever the physical device moves:

```
// Build device gravity behavior
_deviceGravityBehavior = [[UIGravityBehavior alloc] initWithItems:@[]];

// Add observer
__weak typeof(self) weakSelf = self;
id observer = [[NSNotificationCenter defaultCenter]
    addObserverForName:MotionManagerUpdate object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *note) {
        __strong typeof(self) strongSelf = weakSelf;

        // Retrieve vector
        NSDictionary *dict = note.userInfo;
        NSValue *value = dict[MotionVectorKey];
        CGVector vector;
        [value getValue:&vector];

        // Set gravity direction to that vector
        strongSelf.deviceGravityBehavior.gravityDirection = vector;
}];
[_observers addObject:observer];
```

As the `gravityDirection` property updates, any child items (none are yet added in this code) respond to the new force, moving in the appropriate direction.

Creating Boundaries

One of the biggest annoyances about gravity is that it never stops. When you apply a gravity behavior to a view, it will accelerate off the screen and keep going on essentially forever. Bye-bye, view. To avoid this, add a boundary. The `UICollisionBehavior` has a built-in solution for enclosures. Enable its `translatesReferenceBoundsIntoBoundary` property, and it sets the animator's reference view as a default boundary for its items:

```
_boundaryBehavior = [[UICollisionBehavior alloc] initWithItems:@[]];
_boundaryBehavior.translatesReferenceBoundsIntoBoundary = YES;
```

When building behaviors like this, it's important to spot-check your key steps. Remember that animators own behaviors, and behaviors own items, which are typically views. Don't forget to add items to each behavior that affects them. For this example of device-based gravity, add views to both the gravity behavior *and* the boundary behavior. Also, make sure to add the behaviors to the animator. Always make sure your views fall fully within the collision boundaries *before* adding a behavior to the animator. Views that cross the boundary or lie outside the boundary will not respond properly to the "keep items within the reference bounds" rule.

Collision behaviors also enable views to bounce off each other. By default, any view added to a collision behavior will participate not only in view-to-boundary collisions but also in view-to-view collisions. If for any reason you don't want this to happen, you can update the behavior's `collisionMode` property to exclude item-to-item collisions:

```
_boundaryBehavior = [[UICollisionBehavior alloc] initWithItems:@[]];
_boundaryBehavior.translatesReferenceBoundsIntoBoundary = YES;
_boundaryBehavior.collisionMode = UICollisionBehaviorModeBoundaries;
```

Enhancing View Dynamics

Dynamic item behaviors customize view traits—making them springier or duller, heavier or lighter, smoother or stickier, and so forth. Unlike the other built-in behaviors, dynamic item behaviors focus less on external forces and more on individual view properties. For example, say you have views that you want to add bounce to. Create a dynamic item behavior and adjust its `elasticity` property:

```
_elasticityBehavior = [[UIDynamicItemBehavior alloc] initWithItems:items];
_elasticityBehavior.elasticity = 0.8; // Higher values are more elastic
[_animator addBehavior:_elasticityBehavior];
```

Dynamic item properties include the following:

- **Rotation (`allowsRotation`)**—This property allows or disallows view rotation as the view participates in the dynamic system. When it is enabled (the default), views may rotate as they collide with other items.
- **Angular resistance (`angularResistance`)**—Angular resistance creates a damping effect on rotation. As the value of this property rises from 0 to 1, views stop tumbling more quickly.

- **Resistance (`resistance`)**—Also ranging from 0 to 1, the linear resistance property is analogous to angular resistance. Instead of damping rotation, it limits linear velocity. You can think of this as a natural viscosity in the view's "atmosphere," where 0 is close to operating in a vacuum, and 1 is like moving through thick syrup.
- **Density (`density`)**—An item's `density` property controls its virtual mass. Any dynamic behavior that uses mass as a factor (such as collisions and friction) responds to the current value of this property, which defaults to 1. Because items have density, a view that's twice the size of another along each dimension will contribute four times the effective mass when set to the same density or equal mass when set to a quarter of the density.
- **Elasticity (`elasticity`)**—Ranging from 0 to 1, this property establishes how elastic a view's collisions will be. At 0, collisions are lifeless, with no bounce at all. A setting of 1 creates completely elastic collisions with wildly bouncy items.
- **Friction (`friction`)**—The `friction` property creates linear resistance, producing a kind of "stickiness" for when items slide across each other. As the `friction` setting rises from 0 (friction-free) to 1 (the strongest possible friction), views tend to disperse energy on contact and connect more strongly to each other and to boundaries.

Custom Behaviors

Apple provides a library of default behaviors that includes forces (attachments, collisions, gravity, pushes, and snaps) and "dynamic items" that describe how a physics body reacts to forces. You can also create your own behaviors that operate with dynamic animators. This section discusses how you might do this in your own projects.

You choose from two approaches when creating custom dynamic behaviors. First, you can hook your changes onto an existing behavior and transform its updates to some new style. That's the approach Apple uses in the Dynamic Catalog example that converts an attachment point animator to a boundary animation. It transforms an elastic attachment to view morphing. Second, you can create a new behavior and establish your own rules for coalescing its results over time. This approach enables you create any kind of behavior you can imagine, as long as you express it with regard to the animator's timeline. Both have advantages and drawbacks.

Creating Custom Dynamic Items

Before jumping into custom behaviors, you need to understand dynamic items more fully. Dynamic items are the focal point of the dynamic animation process. Until this point, I have used views as dynamic items—after all, they provide the `bounds`, `center`, and `transform` properties required to act in this role—but dynamic items are not necessarily views. They are merely objects that conform to the `UIDynamicItem` protocol. This protocol ensures that these properties are available from conforming objects. Because of this abstraction, you can dynamically animate custom objects as easily as you animate views.

Consider the following class. It consists of nothing more than three properties, ensuring that it conforms to the `UIDynamicItem` protocol:

```
@interface CustomDynamicItem : NSObject <UIDynamicItem>
@property (nonatomic) CGRect bounds;
@property (nonatomic) CGPoint center;
@property (nonatomic) CGAffineTransform transform;
@end
@implementation CustomDynamicItem
@end
```

After adding this class to your project, you can instantiate and set properties however you like. For example, you might use the following lines of code to create a new custom item:

```
item = [[CustomDynamicItem alloc] init];
item.bounds = CGRectMake(0, 0, 100, 100);
item.center = CGPointMake(50, 50);
item.transform = CGAffineTransformIdentity;
```

Once you have established a dynamic item, you may pass it to a behavior and add that behavior to an animator, just as you would with a view:

```
animator = [[UIDynamicAnimator alloc] init];
UIPushBehavior *push = [[UIPushBehavior alloc]
    initWithItems:@[item] mode:UIPushBehaviorModeContinuous];
push.angle = M_PI_4;
push.magnitude = 1.0;
[animator addBehavior:push];
push.active = YES;
```

What happens next, however, may surprise you. If you monitor the item, you'll find that its center property updates, but its bounds and transform remain untouched:

```
2014-12-01 13:33:08.177 Hello World[55151:60b] Bounds: [0, 0, 100, 100], Center:
(86 86), Transform: Theta: {0.000000 radians, 0.000000°} Scale: {1.000000,
1.000000} Translation: {0.000000, 0.000000}
2014-12-01 13:33:09.176 Hello World[55151:60b] Bounds: [0, 0, 100, 100], Center:
(188 188), Transform: Theta: {0.000000 radians, 0.000000°} Scale: {1.000000,
1.000000} Translation: {0.000000, 0.000000}
2014-12-01 13:33:10.175 Hello World[55151:60b] Bounds: [0, 0, 100, 100], Center:
(351 351), Transform: Theta: {0.000000 radians, 0.000000°} Scale: {1.000000,
1.000000} Translation: {0.000000, 0.000000}
2014-12-01 13:33:11.176 Hello World[55151:60b] Bounds: [0, 0, 100, 100], Center:
(568 568), Transform: Theta: {0.000000 radians, 0.000000°} Scale: {1.000000,
1.000000} Translation: {0.000000, 0.000000}
```

This curious state of affair happens because the dynamic animator remains completely agnostic as to the kind of underlying object it serves. This abstract `CustomDynamicItem` class provides no links between its `center` property and its `bounds` property the way a view would. If you

want these items to update synchronously, you must add corresponding methods. For example, you might implement a solution like this:

```
- (void) setCenter:(CGPoint)center
{
    _center = center;
    _bounds = RectAroundCenter(_center, _bounds.size);
}

- (void) setBounds:(CGRect)bounds
{
    _bounds = bounds;
    _center = RectGetCenter(bounds);
}
```

I'm not going to present a full implementation that allows the item to respond to transform changes—for two reasons. First, in real life, you almost never want to create custom items in this fashion. Second, when you actually do need this, you'll be far better off using an actual view as an underlying model. Allowing a `UIView` instance to do the math for you will save you a lot of grief, especially since you're trying to emulate a view in the first place.

> **Note**
> I am unaware of any workaround that will allow you to create non-rectangular dynamic items at this time.

Subverting Dynamic Behaviors

As mentioned earlier, Apple created a Dynamic Catalog example that redirects the results of an attachment behavior to create a bounds animation. It accomplishes this by building an abstract dynamic item class. This class redirects all changes applied to the item's center to a client view's width and height. This means that while the physics engine thinks it's bouncing around a view in space, the actual expressions of those dynamics are producing `bounds` shifts. The following code performs this mapping:

```
// Map bounds to center
- (CGPoint)center
{
    return CGPointMake(_item.bounds.size.width, _item.bounds.size.height);
}

// Map center to bounds
- (void)setCenter:(CGPoint)center
{
    _item.bounds = CGRectMake(0, 0, center.x, center.y);
}
```

I dislike this approach for the following reasons:

- The animator isn't animating the view's center at the point you think it is. You must establish an anchor point within the view's own coordinate system so the center values make any sense to use.
- All you're getting back from this exercise is a damped sinusoid, as in Listing 5-2. Just use a damped sinusoid to begin with, and you'll avoid any unintentional side effects.
- How often are you just sitting around in your development job, thinking, "Hey, I'll just take the output of a physics emulation system and map its results into another dimension so I can create an overly complex sample application that has no general reuse value?" Right, me either.

Better Custom Dynamic Behaviors

As you read this section, remember that *better* is a relative term. The biggest problem when it comes to custom dynamic behaviors is that Apple has not released a public API that keeps a completely custom item animating until it reaches a coalesced state. This means that while Listing 6-5 offers a more satisfying solution than Apple's solution, it's still a hack.

The main reason for this is that while built-in dynamic behaviors can tell the animator "Hey, I'm done now" by using private APIs that allow the animator to stop, you and I cannot tickle the animator to make sure it keeps on ticking. Enter this class's "clock mandate." It's a gravity behavior added to the `ResizableDynamicBehavior` as a child.

The gravity behavior works on an invisible view, which is itself added to the animated view so that it belongs to the right hierarchy. (This is an important step so you don't generate exceptions.) Once it is added, the gravity behavior works forever. When you're ready for the dynamic behavior to end, simply remove it from its parent. Without this extra trick, the animation ends on its own about a half second after you start it.

I developed the damped equation used in the `action` block after playing with graphing. As Figure 6-1 shows, I was looking for a curve that ended after about one and a half cycles. You cannot depend on the animator's elapsed time, which doesn't reset between behaviors. To power my curve, I made sure to create a clock for each behavior and use that in the action block.

Figure 6-1 A fast-decaying sin curve provides a nice match to the view animation.

A few final notes on this one:

- You need to attach some sort of built-in animator like gravity, or your `action` property will not be called. Gravity offers the simple advantage of never ending.
- You must establish the `bounds` as is done here, or your view immediately collapses to a 0 size.
- The `identity` transform in the last step isn't strictly necessary, but I wanted to ensure that I cleaned up after myself as carefully as possible.
- To slow down the effect, reduce the number of degrees traveled per second. In this case, it goes 2 * pi every second.
- To increase or decrease the animation magnitude, adjust the multiplier. Here it is 1 + 0.5 * *the scale*. The 1 is the identity scale, and you should keep it as is. Tweak the 0.5 value up to expand the scaling or down to diminish it.
- You can bring the animation to coalescence faster or slower by adjusting the final multiplier in the exponentiation. Here it is set to 2.0, which produces fairly rapid damping. Higher values produce stronger damping; lower values allow the animation to continue longer.

Listing 6-5 **Extending a Custom Behavior's Lifetime**

```
@interface ResizableDynamicBehavior ()
@property (nonatomic, strong) UIView *view;
@property (nonatomic) NSDate *startingTime;
@property (nonatomic) CGRect frame;
@property (nonatomic) UIGravityBehavior *clockMandate;
@property (nonatomic) UIView *fakeView;
@end

@implementation ResizableDynamicBehavior
- (instancetype) initWithView: (UIView *) view
{
    if (!view) return nil;
    if (!(self = [super init])) return  self;
    _view = view;
    _frame = view.frame;

    // Establish a falling view to keep the timer going
    _fakeView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 10, 10)];
    [view addSubview:_fakeView];
    _clockMandate = [[UIGravityBehavior alloc] initWithItems:@[_fakeView]];
    [self addChildBehavior:_clockMandate];

    // The action block is called at every animation cycle
    __weak typeof(self) weakSelf = self;
```

```
    self.action = ^{
        __strong typeof(self) strongSelf = weakSelf;

        // Start or update the clock
        if (!strongSelf.startingTime)
            strongSelf.startingTime = [NSDate date];
        CGFloat time = [[NSDate date]
            timeIntervalSinceDate:strongSelf.startingTime];

        // Calculate the current change
        CGFloat scale =  1 + 0.5 * sin(time * M_PI * 2) *
            exp(-1.0 * time * 2.0);

        // Apply the bounds and transform
        CGAffineTransform transform =
            CGAffineTransformMakeScale(scale, scale);
        strongSelf.view.bounds = (CGRect){.size = strongSelf.frame.size};
        strongSelf.view.transform = transform;
        [strongSelf.dynamicAnimator
            updateItemUsingCurrentState:strongSelf.view];

        // Stop after 3 * Pi
        if (time > 1.5)
        {
            [strongSelf removeChildBehavior:strongSelf.clockMandate];
            [strongSelf.fakeView removeFromSuperview];
            strongSelf.view.transform = CGAffineTransformIdentity;
        }
    };

    return self;
}
@end
```

Custom Secondary Behaviors

You do far less work when your custom behavior acts side-by-side with a known system-supplied one. You don't have to establish an overall animation end point, the way Listing 6-5 does. Consider Listing 6-6, which creates a behavior that modifies a view transformation over time. This class is duration agnostic. Its only customizable feature is an acceleration property, which establishes how fast the changes accelerate to an end point.

With custom behaviors, it's really important that you not tie yourself to a set timeline. While a system-supplied snap behavior might end after 80 updates or so, you should never rely on knowing that information in advance. In contrast, with keyframes, you are free to interpolate a function over time. With dynamics, you establish a system that *coalesces*, reaching a natural stopping point on its own.

For example, Listing 6-6 uses velocity and acceleration to drive its changes from 0% to 100%, applying an easing function to that transit to produce a smooth animated result. At no point does the behavior reference elapsed time. Instead, all updates are driven by the dynamic animation's heartbeat and applied whenever the `action` method is called.

Figure 6-2 shows the animation in action, with the two behaviors acting in parallel. As the views draw near to their snap points, they apply the requested transforms to finish with a coordinated pile of views.

Figure 6-2 In this animation, a snap behavior draws the views together, and a transformation behavior angles each item to form a tight nest.

Listing 6-6 **Building a Transform-Updating Behavior**

```
- (instancetype) initWithItem: (id <UIDynamicItem>) item
    transform: (CGAffineTransform) transform;
{
    if (!(self = [super init])) return self;

    // Store the passed information
    _item = item;
    _originalTransform = item.transform;
    _targetTransform = transform;

    // Initialize velocity and acceleration
    _velocity = 0;
    _acceleration = 0.0025;

    // The weak and strong workarounds used here avoid retain cycles
    // when using blocks.
    ESTABLISH_WEAK_SELF;
    self.action = ^(){
        ESTABLISH_STRONG_SELF;
```

```
        // Pull out the original and destination transforms
        CGAffineTransform t1 = strongSelf.originalTransform;
        CGAffineTransform t2 = strongSelf.targetTransform;

        // Original
        CGFloat xScale1 = sqrt(t1.a * t1.a + t1.c * t1.c);
        CGFloat yScale1 = sqrt(t1.b * t1.b + t1.d * t1.d);
        CGFloat rotation1 = atan2f(t1.b, t1.a);

        // Target
        CGFloat xScale2 = sqrt(t2.a * t2.a + t2.c * t2.c);
        CGFloat yScale2 = sqrt(t2.b * t2.b + t2.d * t2.d);
        CGFloat rotation2 = atan2f(t2.b, t2.a);

        // Calculate the animation acceleration progress
        strongSelf.velocity = velocity + strongSelf.acceleration;
        strongSelf.percent = strongSelf.percent + strongSelf.velocity;
        CGFloat percent = MIN(1.0, MAX(strongSelf.percent, 0.0));
        percent = EaseOut(percent, 3);

        // Calculated items
        CGFloat targetTx = Tween(t1.tx, t2.tx, percent);
        CGFloat targetTy = Tween(t1.ty, t2.ty, percent);
        CGFloat targetXScale = Tween(xScale1, xScale2, percent);
        CGFloat targetYScale = Tween(yScale1, yScale2, percent);
        CGFloat targetRotation = Tween(rotation1, rotation2, percent);

        // Create transforms
        CGAffineTransform scaleTransform =
            CGAffineTransformMakeScale(targetXScale, targetYScale);
        CGAffineTransform rotateTransform =
            CGAffineTransformMakeRotation(targetRotation);
        CGAffineTransform translateTransform =
            CGAffineTransformMakeTranslation(targetTx, targetTy);

        // Combine and apply transforms
        CGAffineTransform t = CGAffineTransformIdentity;
        t = CGAffineTransformConcat(t, rotateTransform);
        t = CGAffineTransformConcat(t, scaleTransform);
        t = CGAffineTransformConcat(t, translateTransform);
        strongSelf.item.transform = t;
    };

    return self;
}
```

Collection Views and Dynamic Animators

Leveraging the power of dynamic animators in collection views is possible courtesy of a few UIKit extensions. Dynamic animators add liveliness to your presentations during scrolling and when views enter and leave the system. The dynamic behavior set is identical to that used for normal view animation, but the collection view approach requires a bit more overhead and bookkeeping as views may keep appearing and disappearing during scrolls.

The core of the dynamic animator system is the `UIDynamicItem` protocol. The `UICollectionViewLayoutAttributes` class, which represents items in the collection view, conforms to this protocol. Each instance provides the required `bounds`, `center`, and `transform` properties you need to work with dynamic animators. So although you don't work directly with views, you're still well set to introduce dynamics.

Custom Flow Layouts

The key to using dynamic animation classes with collection views is to build your own custom `UICollectionViewFlowLayout` subclass. Flow layouts create organized presentations in your application. Their properties and instance methods specify how the flow sets itself up to place items onscreen. In the most basic form, the layout properties provide you with a geometric vocabulary, where you talk about row spacing, indentation, and item-to-item margins. With custom subclasses, you can extend the class to produce eye-catching and nuanced results.

To support dynamic animation, your custom class must coordinate with an animator instance. You typically set it up in your flow layout initializer by using the `UIDynamicAnimator` collection view-specific initializer. This prepares the animator for use with your collection view and enables it to take control of reporting item attributes on your behalf. As you'll see, the dynamic animator takes charge of many methods you normally would have to implement by hand.

The following init method allocates an animator and adds a custom "spinner" behavior. The `UIDynamicItemBehavior` class enables you to add angular velocity to views, creating a spinning effect, which you see in action in Figure 6-3:

```
- (instancetype) initWithItemSize: (CGSize) size
{
    if (!(self = [super init])) return self;
    _animator = [[UIDynamicAnimator alloc]
        initWithCollectionViewLayout:self];
    _spinner = [[UIDynamicItemBehavior alloc] init];
    _spinner.allowsRotation = YES;
    [_animator addBehavior:_spinner];
    self.scrollDirection = UICollectionViewScrollDirectionHorizontal;
    self.itemSize = size;
    return self;
}
```

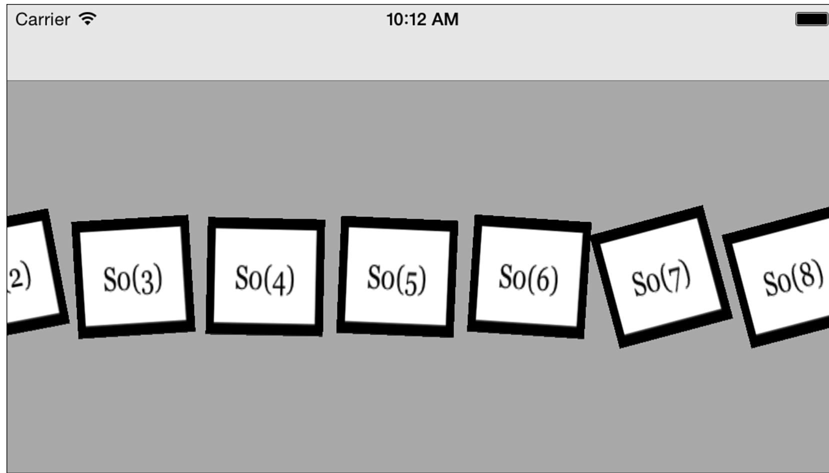

Figure 6-3 Allowing dynamic items to rotate enables you to add angular velocities, causing views to tilt and spin.

Returning Layout Attributes

As mentioned earlier, a dynamic animator can take charge of reporting layout attributes. The following methods do all the work, redirecting the normal geometry through the animator:

```
- (NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    return [_animator itemsInRect:rect];
}

- (UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath:
    (NSIndexPath *)indexPath
{
    UICollectionViewLayoutAttributes *dynamicLayoutAttributes =
        [_animator layoutAttributesForCellAtIndexPath:indexPath];

    // Check whether the attributes were properly generated
    return dynamicLayoutAttributes ?
        [_animator layoutAttributesForCellAtIndexPath:indexPath] :
        [super layoutAttributesForItemAtIndexPath:indexPath];
}

- (BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)newBounds
{
    return YES;
}
```

For safety, the second method checks that the animator properly reports attributes. If it fails, the method falls back to the default implementation.

Updating Behaviors

With collection views, the hardest work involves coordinating items with behaviors. Although you can allow behaviors to control items that are no longer onscreen, as a general rule, you probably want to weed out any items that have left the display and add any items that have moved into place. Listing 6-7 demonstrates this approach.

You start by calculating the onscreen rectangle and request the array of items that appear in that space. Use each item's index path to compare it to items owned by a behavior. If a behavior item does not appear in the onscreen list, remove it. If an onscreen item isn't yet owned by the behavior, add it.

Although you mostly just add physics behaviors and let them run, I decided to tie Listing 6-7 to user interaction. The speed and direction of the backing scroll view add "impulses" to each view, nudging their angular velocity in one direction or the other.

Listing 6-7 **Adding Physics-Based Animation to Collection Views**

```
// Scroll view delegate method establishes the current speed
- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
    scrollSpeed = scrollView.contentOffset.x - previousScrollViewXOffset;
    previousScrollViewXOffset = scrollView.contentOffset.x;
}

// Prepare the flow layout
- (void) prepareLayout
{
    [super prepareLayout];

    // The collection view isn't established in init, catch it here.
    if (!setupDelegate)
    {
        setupDelegate = YES;
        self.collectionView.delegate = self;
    }

    // Retrieve onscreen items
    CGRect currentRect = self.collectionView.bounds;
    currentRect.size = self.collectionView.frame.size;
    NSArray *items = [super layoutAttributesForElementsInRect:currentRect];

    // Clean up any item that's now offscreen
    NSArray *itemPaths = [items valueForKey:@"indexPath"];
```

```
    for (UICollectionViewLayoutAttributes *item in _spinner.items)
    {
        if (![itemPaths containsObject:item.indexPath])
            [_spinner removeItem:item];
    }

    // Add all onscreen items
    NSArray *spinnerPaths = [_spinner.items valueForKey:@"indexPath"];
    for (UICollectionViewLayoutAttributes *item in items)
    {
        if (![spinnerPaths containsObject:item.indexPath])
            [_spinner addItem:item];
    }

    // Add impulses
    CGFloat impulse = (scrollSpeed /
        self.collectionView.frame.size.width) * M_PI_4 / 4;
    for (UICollectionViewLayoutAttributes *item in _spinner.items)
    {
        CGAffineTransform t = item.transform;
        CGFloat rotation = atan2f(t.b, t.a);
        if (fabs(rotation) > M_PI / 32) impulse = - rotation * 0.01;
        [_spinner addAngularVelocity:impulse forItem:item];
    }
}
```

Building a Dynamic Alert View

I stumbled across developer Victor Baro's dynamic iOS "jelly view" (http://victorbaro.com/2014/07/vbfjellyview-tutorial/), which instantly caught my eye. This clever hack uses dynamic attachment behaviors that wiggle in harmony, enabling you to create views that emulate Jell-O. Although its utility is limited in practical deployment, it provides a superb example of how traditional iOS elements like alerts can be re-imagined using modern APIs. Figure 6-4 shows a jelly view alert in motion, squashing and stretching as it bounces off an invisible center ledge within the main UI.

Connecting Up the Jelly

The secret to the jelly effect lies in an underlying 3×3 grid of tiny views, all attached to each other and to the main view's center using `UIAttachmentBehavior` instances (see Figure 6-5). These views and their attachments create a semi-rigid backbone that provides the view physics. Listing 6-8 details how these views and attachments are made and installed. The elasticity of the connections allows the views to move toward and away from each other, creating a deformed skeleton for the view presentation.

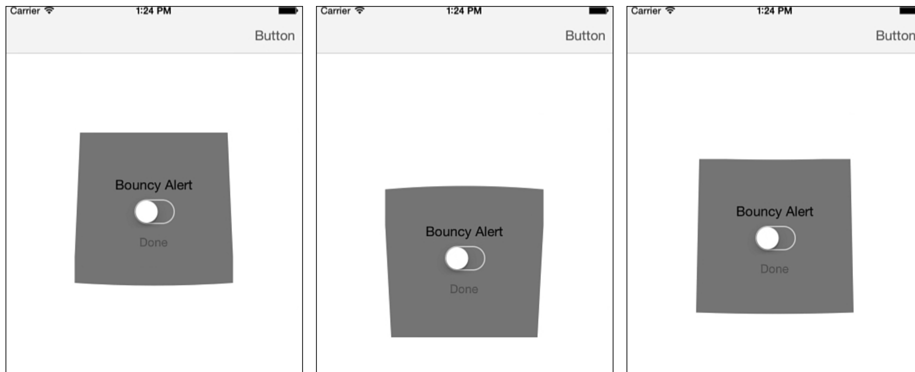

Figure 6-4 This "jelly view" distorts its shape as it uses UIKit dynamics to emulate a view built onto a blob of Jell-O.

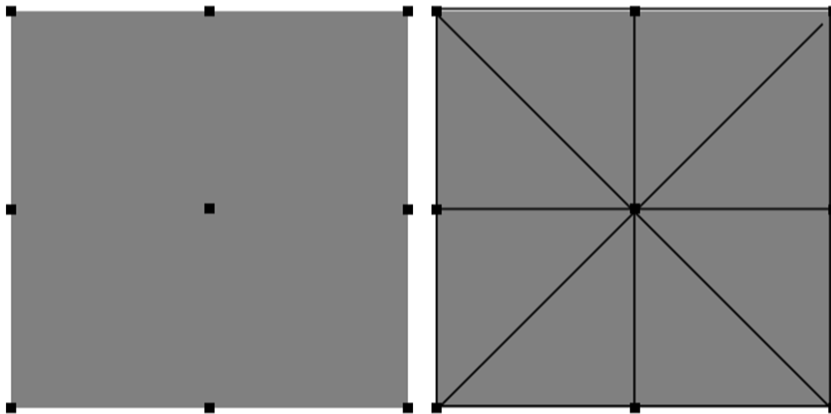

Figure 6-5 The nine connected points form a spring-based skeleton for the Jell-O animation.

Listing 6-8 **Establishing Jelly Dynamics**

```
- (void) establishDynamics : (UIDynamicAnimator *) animator
{
    if (animator) _animator = animator;

    // Create baseline dynamics for primary view
    UIDynamicItemBehavior *dynamic =
        [[UIDynamicItemBehavior alloc] initWithItems:@[self]];
    dynamic.allowsRotation = NO;
    dynamic.elasticity = _elasticity / 2;
```

```
    dynamic.density = _density;
    dynamic.resistance = 0.9;
    [_animator addBehavior:dynamic];

    // Establish jelly grid
    for (int i = 0; i < 9; i++)
    {
        // Add dynamics
        UIView *view = [self viewWithTag:(i + 1)];
        UIDynamicItemBehavior *behavior =
            [[UIDynamicItemBehavior alloc] initWithItems:@[view]];
        behavior.elasticity = _elasticity * 2;
        behavior.density = _density;
        behavior.resistance = 0.2;
        [_animator addBehavior:behavior];

        // Attach each grid view to main jelly view center
        UIAttachmentBehavior *attachment =
            [[UIAttachmentBehavior alloc] initWithItem:view attachedToItem:self];
        attachment.damping = _damping;
        attachment.frequency = _frequency;
        [_animator addBehavior:attachment];

        // Attach views to each other
        if ((i + 1) != 5) // skip center
        {
            NSInteger xTag = [@[@(1), @(2), @(5), @(0), @(4), @(8),
                @(3), @(6), @(7)][i] integerValue] + 1;
            UIView *nextView = [self viewWithTag:xTag];
            attachment = [[UIAttachmentBehavior alloc]
                initWithItem:view attachedToItem:nextView];
            attachment.damping = _damping;
            attachment.frequency = _frequency;
            [_animator addBehavior:attachment];
        }
    }
}
```

Drawing the View

`UIView` instances are rectangular, not gelatinous. To create a view that *looks* as if it deforms, even if the underlying view remains rectangular, you must hide each of the underlying views from Figure 6-5 and draw a unified shape that represents the adjusted skeleton. You do this by observing changes on each of the component views. When they move, which you detect by observing the `center` property, the jelly view needs a redraw. Listing 6-9 shows the redrawing code.

This code works by building a Bezier path from corner point to corner point to corner point. It uses the center views along each edge as control points to produce its inflected curves. Once the curved path is calculated, a standard `drawRect:` method fills in the curve to present the view.

Listing 6-9 **Drawing the Jelly View**

```
- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context
{
    // Update whenever a child view center changes
    [self setNeedsDisplay];
}

- (UIBezierPath *) cornerCurve
{
    // Build a series of quad curve elements from point to point to point
    UIBezierPath *path = [UIBezierPath bezierPath];
    UIView *v0 = [self viewWithTag:1];
    [path moveToPoint:v0.center];

    // The corner points are view destinations.
    // The centers act as control points.
    NSArray *destinations = @[@(2), @(8), @(6), @(0)];
    NSArray *controlPoints = @[@(1), @(5), @(7), @(3)];

    for (int i = 0; i < 4; i++)
    {
        NSInteger dTag = [destinations[i] integerValue] + 1;
        NSInteger cTag = [controlPoints[i] integerValue] + 1;
        UIView *vd = [self viewWithTag:dTag];
        UIView *vc = [self viewWithTag:cTag];
        [path addQuadCurveToPoint:vd.center controlPoint:vc.center];
    }
    return path;
}

- (void) drawRect:(CGRect)rect
{
    // Build the curves and draw the shape
    [_color set];
    [[self cornerCurve] fill];
}
```

Deploying Jelly

While the jelly view is fun to create, deploy with care. Most users have a fixed limit of patience. Any dynamic elements will tend to run longer in presentation and dismissal than standard system-supplied UI elements. They have more complicated visual stories to tell. Because of this, you might need to trade off the cool visual flourishes that excite a developer if you want to put the user experience first. A jelly-based alert may be exciting to develop, but an overly long alert that takes precious seconds to settle may add one-star reviews to your product.

A user will not be able to tell if your app was developed using UIKit, OpenGL, Cocos2D, or SpriteKit. Just because you can now do exciting dynamics in UIKit is not sufficient reason to include those solutions. Your apps must defer to and serve the needs of your users rather than pad your resume and augment your portfolio. Keep this in mind and use dynamic animators sparingly.

Wrap-up

Here are final points to wrap up what you've read in this chapter:

- Dynamic animators and behaviors are like a UI building toy set. They are enormously fun to work with and produce a really great range of results. I best like interactions that direct the user to natural results like the snap zones shown in Listing 6-3 and ones that provide a user-based experience like the device gravity that coordinates with a motion manager in Listing 6-4.
- Although it's easy to get super-flashy with all the built-in physics, some of the best effects are the subtlest. It's the little flourishes—such as bounces when views enter and leave a screen, or collisions when collection items interact with each other—that produce the best results.
- Layering and coordinating behaviors can stylize and customize the otherwise default animations. The scaling, stacking, and rotation I added for Figure 6-2 help send the message that these items have been "put away."
- Some things you might not initially think of as behaviors can turn out to be super-handy. You saw this with the "watcher" behavior in Listing 6-2. Although this custom behavior doesn't introduce any view changes, it helps tune the dynamic system to produce greater responsiveness.
- Always consider behavior lifetimes. You should clean up after your behaviors if they're short lived and retain them if they persist.
- Sometimes it's simpler to create basic and keyframe animations like the ones you saw in Chapter 5, "Animation," than to implement dynamic behaviors with the associated overhead.

Index

Symbols

#error directive, 232-233

#warning directive, 231-232

A

accessibility versions of font sizes, 27

accessing classes, 252-253

action controllers, building, 156-161

actionForKey method, 118

adaptive deployment

- rotation, 207-208
- trait collections, 201-204
  - combining, 203-204
  - defining, 202-203
  - designing for, 204
  - properties, 202
- UIScreen properties, 205-207
  - application frame, 206
  - coordinate spaces, 205-206
  - scale, 207
  - screen bounds, 206

adaptive flow, 58-60

adding

- animations to shaped views, 193-199
- behaviors to dynamic animators, 126-127
- borders to shaped views, 187-190

physics-based behaviors to collection views, 149-150
Quartz 2D contexts to UIKit context stack, 17-18
text fields to alerts, 162-163
touch to labels, 63-69
checking for links, 67
glyphs, 66
implementing visual feedback, 67-69
synchronizing Text Kit items with labels, 64-65
translating coordinates, 65-66

adjusting
attributes, 93-94
pitch of voice playback, 3

alerts, 155-163
building, 156-161
buttons, enabling, 161
class deprecations, 155-156
jelly view alert, building, 150-154
deploying jelly, 154
drawing the view, 152-153
text fields, adding, 162-163

angular velocity, creating the “spinner” effect, 147

angularResistance property, 138

animation, 101
adding to shaped views, 193-199
blocking animators, 105-106
custom dynamic behaviors
improving, 142-144
secondary behaviors, 144-146
custom dynamic items, 139-141
custom transition animations, 113-116
building transitioning objects, 114-116
delegation, 114
UIViewControllerAnimated-Transitioning protocol, 113
dynamic animators, 125
collection views, 147-150
creating, 126-127
detecting pauses, 127-132
snap zones, 133-135
effect views, animating, 172-174
implicit animations, 116-124
animating custom properties, 121-122
animation-ready layers, building, 117-118
completion blocks, 120-121
coordinating, 119-120
drawing properties, 123-124
intercepting updates, 122
timing, 118-119
views, 118
keyframe animation, 101-103
DampedSinusoid() function, 103
scale transformation, 103-105
shaking effect, 102-103
motion effects, 109-112
disabling, 110
shadow effects, 111-112
virtual planes, 109-111
physics-based behaviors, subverting, 141-142

spring-based animation, 106-109
damping constant, 109
practical uses for, 108-109
system animations, 109
view animation, 101

animationKey method, 194

APIs
Cocoa Touch, 248
iOS dictation APIs, 5
Swift, 249-251
UIKit, 50-51

Apple Swift Blog, 260

application frame property, 206

applying text style ranges, 34-35

apps (iOS), building in Swift, 240-243

assets, 208-214
overriding relationships with trait collections, 210-211

assigning values to non-optionals, 248

attachments, 77-78, 125

attributed strings
adjusting attributes, 93-94
attachments, 77-78
building from HTML, 78-83
document type dictionaries, 79-81
converting to document data, 89-90
converting to document representations, 81-82
enhancing, 91-94
fonts, updating, 35-38
custom font faces, 36
dynamic text views, 37-38
initializing from a file, 84-85
inspecting attributes, 87-88
integrating with Dynamic Type, 31-35
applying text style ranges, 34-35
scanning for text style ranges, 32-34
modifying fonts, 42
mutable attributed strings, extending, 94
returning copies of strings with new attributes, 92-93
RTFD integration, 76-77
tabular text, 76

attributedStringWithAttachment method, 78

Auto Layout, 201

AVAssetWriter class, 19

AVCaptureMetadataOutputObjectsDelegate protocol, 11

AVFoundation, movies
building, 14-23
pixel buffer, creating, 16-17

AVMetadataObject class, 11

AVSpeechSynthesizer class, delegate callbacks, 3-4

Aztec code, 9

B

Bacon Ipsum website, 223

barcode recognition, 1, 5-8
CIQRCodeGenerator filter, parameters, 5-6
enhancing recognition, 14
extracting bounds, 13
IOS-supported barcode formats, 8-9
metadata, responding to, 11-13
metadata objects, listening for, 10-11

QR codes
building, 6-8
disabling interpolation, 7-8
Baro, Victor, 150
beginEditing method, 56
Bezier paths
exclusion zones, 61
resizing, 181-183
Bezier-based shape image views, creating, 184-185
blocking animators, 105-106
blocks
ContextDrawingBlock, drawing into the pixel buffer, 17-18
movies, building, 15-16
blogs, Apple Swift Blog, 260
blur effect, building, 170-171
body style, 26
borders, adding to shaped views, 187-190
boundaries
creating for gravity behavior, 138
screen bounds, 206
bounding rectangles, 62
bounds, extracting, 13
bubbles, 176-177
building
action controllers, 156-161
alerts, 156-161
animation-ready layers, 117-118
attributed strings from HTML, 78-83
document type dictionaries, 79-81
AVFoundation movies, 14-23
blur effect, 170-171
fonts from text styles, 28
HTML from attributed strings, 82
images from PDFs, 211-214
iOS apps in Swift, 240-243
jelly view alert, 150-154
deploying jelly, 154
drawing the view, 152-153
mask views, 166-169
movies
expressive drawing, 18-19
from frames, 19-23
images, adding, 23
pixel buffer, creating, 16-17
QR codes, 6-8
shaped buttons, 190-193
side-by-side iPhone split views, 215-218
transitioning objects, 114-116
views around layers, 118
virtual planes, 110-111
buttons
alert buttons, enabling/disabling, 161
shaped buttons, building, 190-193

C

calculating text positions, 95
characterOffsetOfPosition:withinRange: method, 95
CIQRCodeGenerator filter, 5
circular views, creating, 180-183
Clang compiler, 229
class descent, 255-256
classes
accessing, 252-253
AVAssetWriter class, 19
AVMetadataObject class, 11
NSMutableAttributedString class, 56
size classes, 204-205
UIAlertController class, 155

UIBlurEffect class, 170
UIDictationController class, 5
UIDynamicAnimator class, 125-126
UIFont class, 27
UIFontDescriptor class, 40-41
UIImageAsset class, 210-211
UIImageView class, 210
UIInterpolatingMotionEffect class, 111
UIKit, enhancements to, 75-78
UITextView class, 59
UITraitCollection class, 201
UIVisualEffectView class, 169

closestPositionToPoint: method, 96

Cocoa Touch, APIs, 248

Code 39 barcode system, 9

Code 93 barcode system, 9

Code 128 barcode system, 9

collapsed property, 218

collection views
dynamic animators, 147-150
custom flow layouts, 147
returning layout attributes, 148-149
physics-based behaviors, adding, 149-150

collisions, 125

combining trait collections, 203-204

comments, converting to warnings, 229-231

comparing Objective-C and Swift, 239-240

completion blocks, 3-4
implicit completion blocks, building, 120-121

containers, 46, 57-62
adaptive flow, 58-60
bounding rectangles, 62
exclusion zones, 61
insets, 60-61
RTFD containers, 84

ContextDrawingBlock, 17-18

converting
attributed strings to document data, 89-90
attributed strings to document representations, 81-82
comments to warnings, 229-231
HTML to attributed strings, 78-83
document type dictionaries, 79-81
RTFD text to data, 85-86

coordinate spaces, 205-206

coordinating implicit animations, 119-120

Core Image filter, 5
CIQRCodeGenerator filter, parameters, 5-6

Core Motion, integrating with gravity behavior, 135-137

Core Text
glyphs, 47-50
Text Kit, ligatures, 46-47

creating
attributed strings from HTML, document type dictionaries, 79-81
boundaries for gravity behavior, 138
custom behaviors, 139-146
dynamic animators, 126-127
adding behaviors, 126-127
delegation, 126
frame-watching dynamic behaviors, 131-132
HTML from attributed strings, 82
mask views, 166-169
movies, 14-23
expressive drawing, 18-19
from frames, 19-23
images, adding, 23

PDFs, 71-73
QR codes, 6-8
views
Bezier-based shape image views, 184-185
round views, 180-183
virtual planes, 110-111
Cupcake Ipsum website, 223
custom behaviors, creating, 139-146
custom dynamic behaviors
improving, 142-144
secondary behaviors, 144-146
custom flow layouts, 147
custom properties, animating, 121-122
custom transition animations, 113-116
building transitioning objects, 114-116
delegation, 114
UIViewControllerAnimated-Transitioning protocol, 113
customAnimationForKey: method, 121
customizing font sizes, 38

D

damped harmonics, spring-based animation, 106-109
damping constant, 109
practical uses for, 108-109
DampedSinusoid() function, 103
declaring key support, 97-98
defining trait collections, 202-203
delegate callbacks for AVSpeechSynthesizer class, 3-4
delegation, 114
dynamics delegation, 126
density property, 139
designing for traits, 204
detecting
faces, 14
pauses, 127-132
diagnostics, overriding, 235
dictation, 5
directives
converting comments to warnings, 229-231
errors, 232-233
messages, 234
overriding diagnostics, 235
testing for the simulator, 232
unused variable warnings, 235-236
warnings, 231-232
wrapping pragmas, 234-235
disabling
alert buttons, 161
interpolation for QR codes, 7-8
motion effects, 110
displaying supported glyphs for fonts, 53-55
displayModeButtonItem property, 218-219
displayScale property, 202
document attribute dictionaries, establishing, 89-90
documents, creating representations from attributed strings, 81-82
draggable exclusion zones, 69-71
drawInContext:method, 123
drawing
into pixel buffer, 17-18
properties, 123-124
duration of implicit animations, 118-119

dynamic animators, 125
collection views, 147-150
custom flow layouts, 147
returning layout attributes, 148-149
creating, 126-127
adding behaviors, 126-127
delegation, 126
detecting pauses, 127-132
frame-watching dynamic behaviors, creating, 131-132
monitoring views, 128-130
gravity behavior
connecting to device acceleration, 137
creating boundaries, 138
integrating with Core Motion, 135-137
jelly view alert, building, 150-154
deploying jelly, 154
drawing the view, 152-153
physics-based behaviors, 125-126
snap zones, 133-135
dynamic behaviors, subverting, 141-142
Dynamic Type, 25-31
attribute-ready dynamic elements, 35-38
custom font faces, 36
dynamic text views, 37-38
font descriptors
caveats, 40-41
multiple font variations, 41
font sizes, 27
accessibility versions, 27
customizing, 38
user-controlled sizes, 43
integrating with attributed strings, 31-35
applying text style ranges, 34-35
scanning for text style ranges, 32-34
string attributes, modifying fonts with, 42
styles, 26
building fonts from, 28
type updates, listening for, 28-31

E

EAN (European Article Number) barcode, 9
effect views, 169-174
animating, 172-174
blur effect, building, 170-171
vibrancy effects, 171-172
elasticity property, 139
enabling
alert buttons, 161
metadata output, 11
endEditing method, 56
enhancing
attributed strings, 91-94
barcode recognition, 14
view dynamics, 138-139
enumerateAttributesInRange:options: usingBlock: method, 88
enumerating
attributes, 87-88
optionals, 245-246
error handling in Swift, 251
"even/odd" fill rule, 186
exclusion zones, 61
draggable exclusion zones, 69-71

expressive drawing, 18-19
extending mutable attributed strings, 94
extracting bounds, 13

F

faces, detecting, 14
fading logos, building, 122
Fake Name Generator, 225-226
files, saving from the simulator, 237
filters, Core Image filter, 5
flow layouts, 147
font descriptors, 39-42
caveats, 40-41
multiple font variations, 41
font sizes (Dynamic Type), 27
accessibility versions, 27
custom sizing, 38
user-controlled font sizes, 43
fonts
modifying with string attributes, 42
with multiple variations, 41
supported glyphs, displaying, 53-55
updating with dynamic attributes, 35-38
custom font faces, 36
dynamic text views, 37-38
footnotes (Dynamic Type), 26
frames, building movies from, 19-23
frame-watching dynamic behaviors, creating, 131-132
friction property, 139
Fuller, Landon, 238
functions
DampedSinusoid() function, 103
UIGraphicsPopContext() function, 17
UIGraphicsPushContext() function, 17

G

generating
random feeds, 227
random user data, 225-226
gestures
draggable exclusion zones, 69-71
taps, spring-based animation, 106-109
GitHub, xv
lorem ipsum projects, 222
glyphs, 46-55, 66
bounding rectangles, 62
layout managers, 56-57
ligatures, 46-47
supported glyphs for fonts, displaying, 53-55
UIKit, 51-53
gravity behavior, 125
connecting to device acceleration, 137
creating boundaries, 138
integrating with Core Motion, 135-137

H

hardware key support, 97-99
declaring, 97-98
headlines, 26
horizontalSizeClass property, 202
HTML
converting to attributed strings, 78-83
document type dictionaries, 79-81
creating from attributed strings, 82
markup initialization, 83
writing RTFD containers from data, 86-87

hybrid language development, 252-256
accessing classes, 252-253
class descent, 255-256
Objective-C, calling from Swift, 252

I

images
adding to movies, 23
building from PDFs, 211-214
placeholders, 223-225
implementing snap zones, 133-135
implicit animations, 116-124
animating custom properties, 121-122
completion blocks, 120-121
coordinating, 119-120
drawing properties, 123-124
intercepting updates, 122
layers
building, 117-118
views, building, 118
timing, 118-119
improving custom dynamic behaviors, 142-144
inferred types, 244
initializing attributed strings from a file, 84-85
inputCorrectionLevel parameter (CIQRCodeGenerator filter), 5-6
inputMessage parameter (CIQRCodeGenerator filter), 5-6
insets, 60-61
inspecting
attributes, 87-88
items with playgrounds, 258-259
integrating
Dynamic Type with attributed strings, 31-35
applying text style ranges, 34-35
scanning for text style ranges, 32-34
gravity behavior with Core Motion, 135-137
intercepting updates, 122
International Article Number barcode, 9
interpolation, disabling for QR codes, 7-8
iOS 8
attributed text updates, 36
split view controllers, 214-219
supported barcode formats, 8-9

J-K

jelly view alert, building, 150-154
deploying jelly, 154
drawing the view, 152-153
JSON feed resources, 227

key support, 97-99
declaring, 97-98
keyframe animation, 101-103
blocking animators, 105-106
DampedSinusoid() function, 103
scale transformation, 103-105
shaking effect, 102-103

L

labels, enabling touch, 63-69
adding visual feedback, 67-69
checking for links, 67
glyphs, 66

- synchronizing Text Kit items with labels, 64-65
- translating coordinates, 65-66

layers
- animation-ready layers, building, 117-118
- border layers, generating, 188-190
- views, building, 118

layout managers (Text Kit), 46, 56-57

layouts
- attributes, returning, 148-149
- Auto Layout, 201
- containers, 57-62
  - adaptive flow, 58-60
  - exclusion zones, 61
  - insets, 60-61
- custom flow layouts, 147
- document attribute dictionaries, 89-90
- draggable exclusion zones, 69-71
- side-by-side iPhone split views, building, 215-218

learning Swift, 259

ligatures, 46-47

listening
- for metadata objects, 10-11
- for type updates, 28-31

logging, 238

lorem ipsum text, 221-223
- requesting, 222-223

Lorem Pixel website, 224

M

Markdown, 83

marking non-null and nullable items, 236

mask views, 164-169
- building, 166-169
- shape layer masking, 164-166

media
- barcodes, 5-8
  - enhancing recognition, 14
  - extracting bounds, 13
  - iOS-supported barcode formats, 8-9
  - listening for metadata objects, 10-11
  - QR codes, building, 6-8
  - responding to metadata, 11-13
- dictation, 5
- movies
  - adding images, 23
  - building, 16-17
  - creating from frames, 19-23
  - expressive drawing, 18-19
- TTS, 1-4
  - completion blocks, 3-4
  - utterances, 2

messages, 234

metadata
- enabling output, 11
- objects, listening for, 10-11
- responding to, 11-13

methods
- actionForKey method, 118
- animationKey method, 194
- attributedStringWithAttachment method, 78
- characterOffsetOfPosition:withinRange: method, 95
- closestPositionToPoint: method, 96
- customAnimationForKey: method, 121
- drawInContext: method, 123
- enumerateAttributesInRange:options: usingBlock: method, 88
- needsDisplayForKey: method, 122
- setAnimation: method, 194

transformedMetadataObjectFor-MetadataObject method, 13
viewWillTransitionToSize:with-TransitionCoordinator: method, 207
modifying
attributed strings, 93-94
fonts with string attributes, 42
monitoring
items with playgrounds, 258-259
views, 128-130
motion effects, 109-112
disabling, 110
shadow effects, 111-112
virtual planes, 109-110
building, 110-111
movies
building, 14-23
expressive drawing, 18-19
pixel buffer, creating, 16-17
images, adding, 23
pixel buffer
creating, 16-17
drawing into, 17-18
multiple snap zones, handling, 133-135
mutable attributed strings, extending, 94

N

needsDisplayForKey: method, 122
NeXTSTEP, 83
non-null items, marking, 236
NSAttributedString
class convenience methods, 91-92
integrating with Dynamic Type, 31-35
applying text style ranges, 34-35
scanning for text style ranges, 32-34
NSMutableAttributedString class, 56
nullable items, marking, 236

O

Objective-C
calling from Swift, 252
comparing to Swift, 239-240
preparing Swift for, 254-255
objects
text ranges, 95-97
transitioning objects, building, 114-116
optionals, 243-248
enumeration, 245-246
inferred types, 244
unwrapping, 246-247
overriding
relationships between trait collections and assets, 210-211
trait collections, 214-219

P

parameters for CIQRCodeGenerator filter, 5-6
pauses, detecting, 127-132
PDF417 standard, 9
PDFs
building, 71-73
creating images from, 211-214
printing, 74
physics-based behaviors, 125-126
adding to collection views, 149-150
custom behaviors, creating, 139-146
frame-watching dynamic behaviors, creating, 131-132

gravity
connecting to device acceleration, 137
creating boundaries, 138
integrating with Core Motion, 135-137
improving, 142-144
pauses, detecting, 127-132
properties, 138-139
secondary behaviors, 144-146
subverting, 141-142

pitch of voice playback, adjusting, 3

pixel buffer
creating, 16-17
drawing into, 17-18

placeholders
for images, 223-225
lorem ipsum text, 221-223

playgrounds, 256-258

popovers, 175-177
supporting bubbles, 176-177

positions, text positions
calculating, 95
geometry, 95-96
updating selection points, 97

pragmas, wrapping, 234-235

presentations, 155
alerts, 155-163
building, 156-161
buttons, enabling, 161
class deprecations, 155-156
text fields, adding, 162-163
effect views, 169-174
animating, 172-174
blur effect, 170-171
vibrancy effects, 171-172
mask views, 164-169
building, 166-169
shape layer masking, 164-166
popovers, 175-177
supporting bubbles, 176-177

printing text views, 73-74

properties
of dynamic behaviors, 138-139
of trait collections, 202
UIScreen properties, 205-207
application frame, 206
coordinate spaces, 205-206
scale, 207
screen bounds, 206

pushes, 125

Q-R

QR (Quick Response) codes, 5
building, 6-8

Quartz 2D contexts, adding to UIKit context stack, 17-18

random feeds, generating, 227

random generation suite, 228-229

Random User Generator, 225

range dictionaries
applying text style ranges, 34-35
scanning for text style ranges, 32-34

reading barcodes
enhancing recognition, 14
extracting bounds, 13
iOS-supported barcode formats, 8-9
listening for metadata objects, 10-11
responding to metadata, 11-13

repairing attributes for text storage, 56

requesting lorem ipsum text, 222-223

resistance property, 139

resizing Bezier paths, 181-183

responding to metadata, 11-13

Retina display scales, 202

retrieving sample code, xv

returning copies of strings with new attributes, 92-93

rotation property, 138

  "spinner" effect, creating, 147

round views, creating, 180-183

RTF, 83

RTFD containers

  converting text to data, 85-86

  writing from data, 86-87

S

sample code, retrieving, xv

saving files from the simulator, 237

scale property, 207

scanning for text style ranges, 32-34

screen bounds, 206

setAnimation: method, 194

shadow effects, 111-112

shake keyframe animation, 102-103

shape layer masking, 164-166

shaped buttons, building, 190-193

shaped views

  animations, adding, 193-199

  borders, adding, 187-190

  creating, 179-187

    Bezier-based shape image views, 184-185

    round views, 180-183

shapes, unclosed shapes, 185-187

side-by-side iPhone split views, building, 215-218

simulator, saving files from, 237

size classes, 204-205

snap zones, 133-135

  multiple snap zones, handling, 133-135

snaps, 125

speech generation, 1

  completion blocks, 3-4

  TTS, utterances, 2

"spinner" effect, creating, 147

split view controllers, 214-219

  side-by-side iPhone split views, building, 215-218

spring-based animation, 106-109

  damping constant, 109

  practical uses for, 108-109

string attributes, modifying fonts with, 42

structs, UIEdgeInsets struct, 60

styles

  building fonts from, 28

  Dynamic Type, 26

  layout managers, 56-57

subheadlines, 26

subverting dynamic behaviors, 141-142

supported barcode formats, 8-9

Swift, 239

  APIs, 249-251

  calling from Objective-C, 253-254

  error handling, 251

  iOS apps, building, 240-243

  learning, 259-260

  non-optionals, assigning values to, 248

  versus Objective-C, 239-240

optionals, 243-248
enumeration, 245-246
inferred types, 244
unwrapping, 246-247
playgrounds, 256-258
preparing for Objective-C, 254-255
The Swift Programming Language, 259
system animations, 109

T

tabular text, 76
tap gestures, spring-based animation, 106-109
damping constant, 109
practical uses for, 108-109
text. _See also_ Dynamic Type
RTFD text, converting to data, 85-86
text fields, adding to alerts, 162-163
Text Kit, 43
containers, 46, 57-62
adaptive flow, 58-60
bounding rectangles, 62
exclusion zones, 61
insets, 60-61
exclusion zones, draggable exclusion zones, 69-71
glyphs, 46-55
ligatures, 46-47
layout managers, 46, 56-57
PDFs
building, 71-73
printing, 74
text storage, 46, 55-56
objects, 55
repairing attributes, 56
text views, printing, 73-74
touch-enabled labels, 63-69
adding visual feedback, 67-69
checking for links, 67
glyphs, 66
synchronizing Text Kit items with labels, 64-65
translating coordinates, 65-66
text ranges, 95-97
text positions
calculating, 95
geometry, 95-96
updating selection points, 97
text storage (Text Kit), 46, 55-56
objects, 55
repairing attributes, 56
text style ranges
applying, 34-35
scanning for, 32-34
text views
dynamic text views, 37-38
printing, 73-74
touch-enabled labels, 63-69
adding visual feedback, 67-69
checking for links, 67
glyphs, 66
synchronizing Text Kit items with labels, 64-65
translating coordinates, 65-66
trait collections, 201-204
combining, 203-204
defining, 202-203
designing for, 204
overriding relationships with assets, 210-211
properties, 202
split view controllers, 214-219

transformedMetadataObjectForMetadata-Object method, 13

TTS (text-to-speech), 1-4. *See also* dictation

utterances, 2

completion blocks, 3-4

pitchMultiplier, 3

type updates, listening for, 28-31

typography

Dynamic Type, 25-31

font sizes, 27

integrating with attributed strings, 31-35

styles, 26

type updates, listening for, 28-31

glyphs

ligatures, 46-47

supported glyphs for fonts, displaying, 53-55

U

UIAlertController class, 155

UIBlurEffect class, 170

UIDictationController class, 5

UIDynamicAnimator class, 125-126

UIDynamicItem protocol, 139

UIEdgeInsets struct, 60

UIFont class, 27

UIFontDescriptor class, 40-41

UIGraphicsPopContext() function, 17

UIGraphicsPushContext() function, 17

UIImageAsset class, 210-211

UIImageView class, 210

UIInterpolatingMotionEffect class, 111

UIKit

adding Quartz 2D contexts, 17-18

APIs, 50-51

classes, enhancements to, 75-78

dynamic behaviors, 125-126

font descriptors, 39-42

glyphs, 51-53

spring-based animation, 106-109

UINavigationControllerDelegate protocol, 114

UIScreen properties, 205-207

application frame, 206

coordinate spaces, 205-206

scale, 207

screen bounds, 206

UISystemAnimationDelete animation, 109

UITabBarControllerDelegate protocol, 114

UITextInput protocol, text ranges

geometry, 95-96

positions, calculating, 95

updating selection points, 97

UITextView class, 59

UITraitCollection class, 201

UIViewControllerAnimatedTransitioning protocol, 113-114

UIVisualEffectView class, 169

unclosed shapes, 185-187

unused variable warnings, 235-236

unwrapping optionals, 246-247

UPC (Universal Product Code) standard, 9

updating fonts, 35-38

custom font faces, 36

dynamic text views, 37-38

user interface idioms, 202

user-controlled font sizes, 43

***Using Swift with Cocoa and Objective-C*, 259**

utterances, 2

completion blocks, 3-4

V

verticalSizeClass property, 202

vibrancy effects, 171-172

view animation, 101

view controllers, UIViewControllerAnimatedTransitioning protocol, 113

views

Bezier-based shape image views, creating, 184-185

building around layers, 118

collection views

dynamic animators, 147-150

physics-based behaviors, adding, 149-150

dynamics, enhancing, 138-139

effect views, 169-174

animating, 172-174

vibrancy effects, 171-172

jelly view alert, building, 150-154

deploying jelly, 154

drawing the view, 152-153

mask views, 164-169

building, 166-169

shape layer masking, 164-166

monitoring, 128-130

round views, creating, 180-183

shaped views

animations, adding, 193-199

borders, adding, 187-190

text views, printing, 73-74

viewWillTransitionToSize: withTransitionCoordinator: method, 207

virtual planes, 109-110

building, 110-111

visual feedback, adding to touch-enabled labels, 67-69

voice playback, adjusting pitch, 3

W

warnings, 231-232

unused variable warnings, 235-236

websites

Bacon Ipsum, 223

Clang Language Extensions, 234

Cupcake Ipsum, 223

Lorem Pixel, 224

wrapping pragmas, 234-235

writing RTFD containers from data, 86-87

X-Y-Z

XML feed resources, 228

yaw, 14