

MOBILE
PROGRAMMING
SERIES



iOS

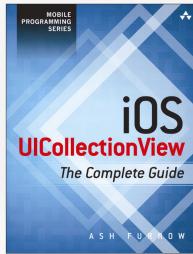
CORE ANIMATION

ADVANCED TECHNIQUES

NICK LOCKWOOD

iOS Core Animation

Addison-Wesley Mobile Programming Series



▼Addison-Wesley



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

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



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

informIT.com
the trusted technology learning source

▼Addison-Wesley

Safari
Books Online

ALWAYS LEARNING

PEARSON

iOS Core Animation

Advanced Techniques

Nick Lockwood

▼ Addison-Wesley

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

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

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

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

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

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informat.com/aw

Copyright © 2014 Pearson Education, Inc.

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

ISBN-13: 978-0-13-344075-1
ISBN-10: 0-13-344075-3

Editor-in-Chief

Mark Taub

Acquisitions Editor

Trina MacDonald

Angie Doyle

Development Editor

Michael Thurston

Managing Editor

Kristy Hart

Project Editor

Sara Schumacher

Copy Editor

Keith Cline

Proofreader

Sarah Kearns

Technical Reviewer

Mugunth Kumar

Publishing Coordinator

Olivia Basegio



For Zoe



Contents at a Glance

I: The Layer Beneath

- 1 The Layer Tree
- 2 The Backing Image
- 3 Layer Geometry
- 4 Visual Effects
- 5 Transforms
- 6 Specialized Layers

II: Setting Things in Motion

- 7 Implicit Animations
- 8 Explicit Animations
- 9 Layer Time
- 10 Easing
- 11 Timer-Based Animation

III: The Performance of a Lifetime

- 12 Tuning for Speed
- 13 Efficient Drawing
- 14 Image IO
- 15 Layer Performance

Table of Contents

Preface

- Audience and Material
- Book Structure
- Before We Begin

I: The Layer Beneath

1 The Layer Tree

- Layers and Views
 - CALayer
 - Parallel Hierarchies
- Layer Capabilities
- Working with Layers
- Summary

2 The Backing Image

- The contents Image
 - contentsGravity
 - contentsScale
 - masksToBounds
 - contentsRect
 - contentsCenter
- Custom Drawing
- Summary

3 Layer Geometry

- Layout
- anchorPoint
- Coordinate Systems
 - Flipped Geometry
 - The Z Axis
- Hit Testing
- Automatic Layout
- Summary

4 Visual Effects

- Rounded Corners
- Layer Borders

- Drop Shadows
- Shadow Clipping
- The `shadowPath` Property

- Layer Masking
- Scaling Filters
- Group Opacity
- Summary

5 **Transforms**

- Affine Transforms
 - Creating a `CGAffineTransform`
 - Combining Transforms
 - The Shear Transform
- 3D Transforms
 - Perspective Projection
 - The Vanishing Point
 - The `sublayerTransform` Property
 - Backfaces
 - Layer Flattening
- Solid Objects
 - Light and Shadow
 - Touch Events
- Summary

6 **Specialized Layers**

- `CAShapeLayer`
 - Creating a `CGPath`
 - Rounded Corners, Redux
- `CATextLayer`
 - Rich Text
 - Leading and Kerning
 - A `UILabel` Replacement
- `CATransformLayer`
- `CAGradientLayer`
 - Basic Gradients
 - Multipart Gradients
- `CAReplicatorLayer`
 - Repeating Layers

- Reflections
- CAScrollLayer
- CATiledLayer
- Tile Cutting
- Retina Tiles
- CAEmitterLayer
- CAEAGLLayer
- AVPlayerLayer
- Summary

II: Setting Things in Motion

7 Implicit Animations

- Transactions
- Completion Blocks
- Layer Actions
- Presentation Versus Model
- Summary

8 Explicit Animations

- Property Animations
 - Basic Animations
 - CAAnimationDelegate
 - Keyframe Animations
 - Virtual Properties
- Animation Groups
- Transitions
 - Implicit Transitions
 - Animating Layer Tree Changes
 - Custom Transitions
- Canceling an Animation in Progress
- Summary

9 Layer Time

- The `CAMediaTiming` Protocol
 - Duration and Repetition
 - Relative Time
 - `fillMode`
- Hierarchical Time

- Global Versus Local Time
- Pause, Rewind, and Fast-Forward
- Manual Animation
- Summary

10 Easing

- Animation Velocity
 - CAMediaTimingFunction
 - UIView Animation Easing
 - Easing and Keyframe Animations
- Custom Easing Functions
 - The Cubic Bézier Curve
 - More Complex Animation Curves
 - Keyframe-Based Easing
 - Automating the Process
- Summary

11 Timer-Based Animation

- Frame Timing
 - NSTimer
 - CADisplayLink
 - Measuring Frame Duration
 - Run Loop Modes
- Physical Simulation
 - Chipmunk
 - Adding User Interaction
 - Simulation Time and Fixed Time Steps
 - Avoiding the Spiral of Death
- Summary

III: The Performance of a Lifetime

12 Tuning for Speed

- CPU Versus GPU
 - The Stages of an Animation
 - GPU-Bound Operations
 - CPU-Bound Operations
 - IO-Bound Operations
- Measure, Don't Guess

- Test Reality, Not a Simulation
- Maintaining a Consistent Frame Rate
- Instruments
 - Time Profiler
 - Core Animation
 - OpenGL ES Driver
- A Worked Example
- Summary

13 Efficient Drawing

- Software Drawing
- Vector Graphics
- Dirty Rectangles
- Asynchronous Drawing
 - CATiledLayer
 - drawsAsynchronously
- Summary

14 Image IO

- Loading and Latency
 - Threaded Loading
 - GCD and NSOperationQueue
 - Deferred Decompression
 - CATiledLayer
 - Resolution Swapping
- Caching
 - The +imageNamed: Method
 - Custom Caching
 - NSCache
- File Format
 - Hybrid Images
 - JPEG 2000
 - PVRTC
- Summary

15 Layer Performance

- Inexplicit Drawing
 - Text
 - Rasterization

Offscreen Rendering

CAShapeLayer

Stretchable Images

shadowPath

Blending and Overdraw

Reducing Layer Count

Clipping

Object Recycling

Core Graphics Drawing

The `-renderInContext:` Method

Summary

About the Author

Nick Lockwood is head of iOS development at the digital agency AKQA in London, and a prolific developer of applications and open source libraries. He has been working with the iOS platform for the past four years, after making the switch from HTML5 web-app development. Nick first picked up a programming book in 1993 at a middle school rummage sale and hasn't looked back since. He lives in Sidcup with his wife and daughter.

Acknowledgments

I would like to thank my wife, Angela, and daughter, Zoe, for putting up with seeing even less of me than normal while I wrote this. Thanks to David Deutsch and P.J. Cook for their valuable feedback on the early manuscript. Thanks to Mugunth Kumar for generously agreeing to be my technical editor, and to all the staff at Pearson, particularly Michael Thurston, Sara Schumacher, Angie Doyle, Trina MacDonald, and Olivia Basegio, who between them somehow managed to extract a book from me in the space of four months. Thanks to my parents, for buying me my first programming book at the tender age of 12, and so starting me on the path to my eventual career. Finally, thanks to Kate for all the cups of tea!

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

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

Preface

When Apple engineers created the iPhone, they faced a challenge: they needed to create a modern, fast, and fluid user interface, the likes of which had never been seen outside of a video game, and they needed to do it on mobile hardware that was a decade behind current desktops and laptops in terms of graphics performance.

They also had an opportunity: to rebuild AppKit (the Mac OS user interface framework) from the ground up using modern graphics technology without needing to maintain support for the legacy applications inherited from nearly 25 years of Macintosh and NeXTSTEP history.

Their solution was a private framework called *Layer Kit*, developed by the iPhone software team to provide a high-performance, hardware-accelerated animation and compositing library to replace the slower, Quartz-based software drawing used by AppKit. This framework actually debuted first on Mac OS 10.5 under the name *Core Animation*, shortly before the iPhone was announced.

Core Animation is not just a set of functions for performing animations; it lies at the very heart of iOS, powering everything you see on screen. Even if you never invoke it explicitly, you are using it *implicitly* every time you display a view or transition from one screen to the next. As an iOS developer, you can build great applications without ever consciously touching Core Animation, but if you truly embrace its features, you can achieve much richer user experiences.

The purpose of this book is to demystify Core Animation, to bring it out from behind the curtains and help you to harness its full power to make spectacular applications. By the end, you will have learned exactly where and when to use Core Animation, how to work with and around its limitations, and what to do to avoid performance pitfalls so that your apps can be as responsive as Apple's own.

Audience and Material

This is neither a beginner's guide to Cocoa nor an introduction to the iOS platform for Mac developers. It is *most definitely not* an introduction to programming in general.

This book is written for an audience that is reasonably familiar with Xcode, Cocoa Touch, and UIKit but has no prior knowledge of the Core Animation framework. You need not be a very seasoned developer (perhaps you have just finished your "Hello World" project and are looking for something a bit more advanced), but you will need to be fluent in Objective-C to be able to follow the code examples.

Although no prior knowledge of Core Animation is assumed, this is not merely a cursory introduction to the framework. The aim of this book is to leave no stone unturned when it comes to the Core Animation APIs. Even if you have already used Core Animation for many years in an iOS-specific context, it would be surprising if you did not find things in here that you don't already know about or fully understand.

This book is specifically geared toward the iOS platform. Where appropriate, differences between Core Animation on iOS and Mac OS are mentioned, but Mac-specific Core Animation features such as `CALayoutManager` or Core Image integration or are not discussed in detail. If you are already well versed in Core Animation on Mac OS, much of the material will already be familiar to you, but this should serve as a useful conversion guide.

Core Animation has been a key part of iOS since the beginning, and the majority of its features are available on older iOS versions. Any methods or classes that are new to iOS 6 are highlighted as such, but the purpose of this book is to document the *current* feature set of Core Animation; so with very few exceptions, the task of providing backward compatibility for earlier iOS versions is left as an exercise for the reader.

Book Structure

This book is structured in a linear fashion, with each chapter building upon concepts introduced in the previous ones. That said, wherever we refer back to a concept covered in an earlier chapter, the chapter is referenced explicitly so that you can read the book in a nonlinear fashion if you prefer.

The subject matter is split into three parts, dealing with static content, animation, and performance optimization, respectively. These parts are self-contained, so (for example) if you already know about animation and layout, you can dive straight into the part on performance.

Each chapter contains figures and example code to illustrate the topics discussed. The sample code projects are available for download from www.informit.com/title/9780133440751 if you prefer not to retype them by hand.

Before We Begin

The examples in this book were written and tested using Xcode 4.6 on Mac OS 10.8 (*Mountain Lion*). The latest version of Xcode can be downloaded free of charge from the Mac App Store, and most of the examples can be run in the iOS simulator. In addition, you need to sign up for a free Apple developer account to access most of the tools and documentation for the classes referenced in the book.

The code examples have all been tested with iOS 6.1, but most will either run unmodified on iOS 5+, or can be trivially modified to do so by removing noncritical iOS 6 features such as Autolayout. All examples make use of modern Objective-C practices such as ARC (Automatic Reference Counting), automatic property synthesis, and object literals and so require Xcode 4.5 and iOS 4 as a minimum.

The examples in the final, performance-focused section of the book must be installed on a physical iPhone 5 running iOS 6.1 to demonstrate the exact behavior described in the text. To run the examples on an iPhone, you need to pay for an iOS developer license, which you can purchase directly from Apple. These examples will still work on the simulator, or a different device or iOS version, but will likely exhibit different performance characteristics.

This page intentionally left blank



The Layer Beneath

- 1** The Layer Tree
- 2** The Backing Image
- 3** Layer Geometry
- 4** Visual Effects
- 5** Transforms
- 6** Specialized Layers

This page intentionally left blank

The Layer Tree

Ogres have layers. Onions have layers. You get it? We both have layers.

Shrek

Core Animation is misleadingly named. You might assume that its primary purpose is *animation*, but actually animation is only one facet of the framework, which originally bore the less animation-centric name of *Layer Kit*.

Core Animation is a *compositing engine*; its job is to compose different pieces of visual content on the screen, and to do so as fast as possible. The content in question is divided into individual *layers* stored in a hierarchy known as the *layer tree*. This tree forms the underpinning for all of UIKit, and for everything that you see on the screen in an iOS application.

Before we discuss animation at all, we’re going to cover Core Animation’s *static* compositing and layout features, starting with the layer tree.

Layers and Views

If you’ve ever created an app for iOS or Mac OS, you’ll be familiar with the concept of a *view*. A view is a rectangular object that displays content (such as images, text, or video), and intercepts user input such as mouse clicks or touch gestures. Views can be nested inside one another to form a hierarchy, where each view manages the position of its children (*subviews*). Figure 1.1 shows a diagram of a typical view hierarchy.

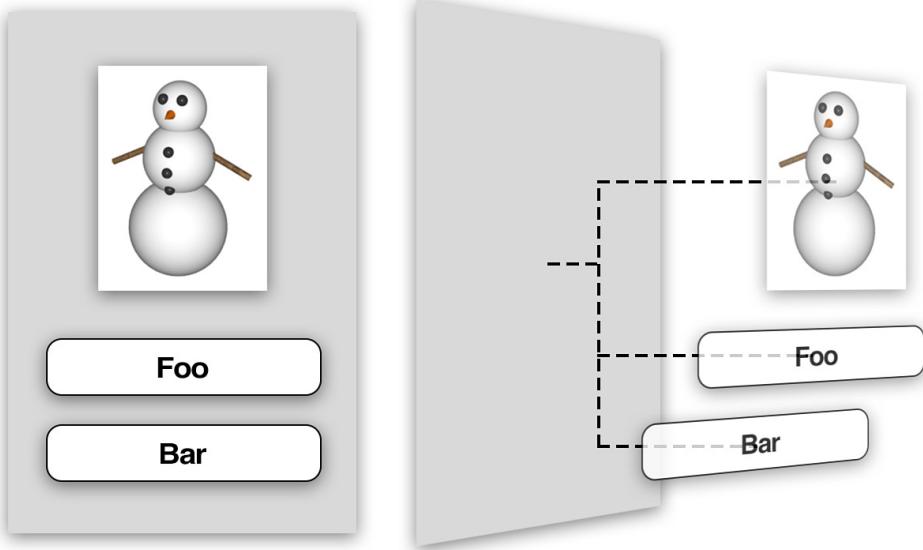


Figure 1.1 A typical iOS screen (left) and the view hierarchy that forms it (right)

In iOS, views all inherit from a common base class, `UIView`. `UIView` handles touch events and supports *Core Graphics*-based drawing, affine transforms (such as rotation or scaling), and simple animations such as sliding and fading.

What you may not realize is that `UIView` does not deal with most of these tasks itself. Rendering, layout, and animation are all managed by a Core Animation class called `CALayer`.

CALayer

The `CALayer` class is conceptually very similar to `UIView`. Layers, like views, are rectangular objects that can be arranged into a hierarchical tree. Like views, they can contain content (such as an image, text, or a background color) and manage the position of their children (*sublayers*). They have methods and properties for performing animations and transforms. The only major feature of `UIView` that isn't handled by `CALayer` is user interaction.

`CALayer` is not aware of the *responder chain* (the mechanism that iOS uses to propagate touch events through the view hierarchy) and so cannot respond to events, although it does provide methods to help determine whether a particular touch point is within the bounds of a layer (more on this in Chapter 3, “Layer Geometry”).

Parallel Hierarchies

Every `UIView` has a `layer` property that is an instance of `CALayer`. This is known as the *backing layer*. The view is responsible for creating and managing this layer and for ensuring that when subviews are added or removed from the view hierarchy that their corresponding backing layers are connected up in parallel within the layer tree (see Figure 1.2).

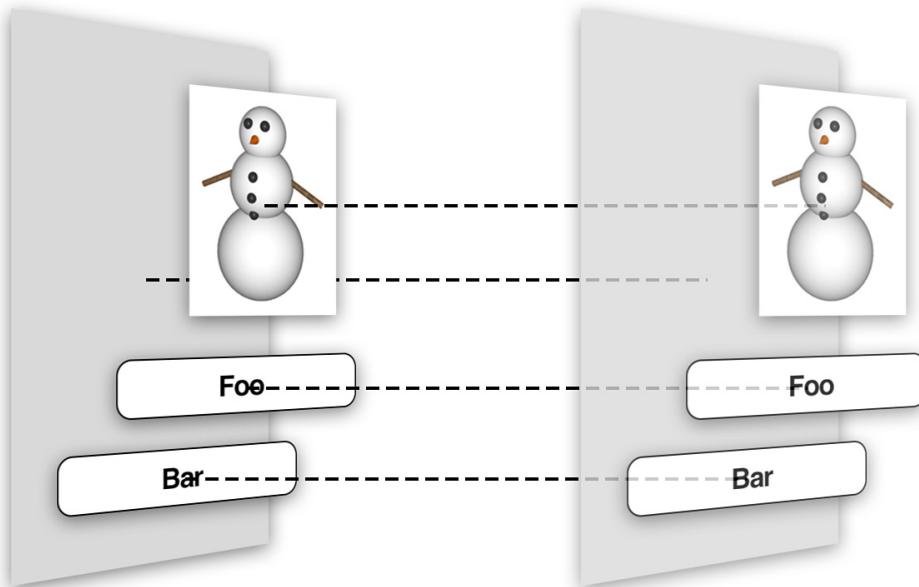


Figure 1.2 The structure of the layer tree (left) mirrors that of the view hierarchy (right)

It is actually these backing layers that are responsible for the display and animation of everything you see onscreen. `UIView` is simply a wrapper, providing iOS-specific functionality such as touch handling and high-level interfaces for some of Core Animation's low-level functionality.

Why does iOS have these two parallel hierarchies based on `UIView` and `CALayer`? Why not a single hierarchy that handles everything? The reason is to separate responsibilities, and so avoid duplicating code. Events and user interaction work quite differently on iOS than they do on Mac OS; a user interface based on multiple concurrent finger touches (*multitouch*) is a fundamentally different paradigm to a mouse and keyboard, which is why iOS has `UIKit` and `UIView` and Mac OS has `AppKit` and `NSView`. They are functionally similar, but differ significantly in the implementation.

Drawing, layout, and animation, in contrast, are concepts that apply just as much to touchscreen devices like the iPhone and iPad as they do to their laptop and desktop cousins. By separating out the logic for this functionality into the standalone Core Animation framework, Apple is able to share that code between iOS and Mac OS, making things simpler both for Apple's own OS development teams and for third-party developers who make apps that target both platforms.

In fact, there are not two, but *four* such hierarchies, each performing a different role. In addition to the view hierarchy and layer tree, there are the *presentation tree* and *render tree*, which we discuss in Chapter 7, “Implicit Animations,” and Chapter 12, “Tuning for Speed,” respectively.

Layer Capabilities

So if `CALayer` is just an implementation detail of the inner workings of `UIView`, why do we need to know about it at all? Surely Apple provides the nice, simple `UIView` interface precisely so that we don't need to deal directly with gnarly details of Core Animation itself?

This is true to some extent. For simple purposes, we don't really need to deal directly with `CALayer`, because Apple has made it easy to leverage powerful features like animation *indirectly* via the `UIView` interface using simple high-level APIs.

But with that simplicity comes a loss of *flexibility*. If you want to do something slightly out of the ordinary, or make use of a feature that Apple has not chosen to expose in the `UIView` class interface, you have no choice but to venture down into Core Animation to explore the lower-level options.

We've established that layers cannot handle touch events like `UIView` can, so what can they do that views *can't*? Here are some features of `CALayer` that are not exposed by `UIView`:

- Drop shadows, rounded corners, and colored borders
- 3D transforms and positioning
- Nonrectangular bounds
- Alpha masking of content
- Multistep, nonlinear animations

We explore these features in the following chapters, but for now let's look at how `CALayer` can be utilized within an app.

Working with Layers

Let's start by creating a simple project that will allow us to manipulate the properties of a layer. In Xcode, create a new iOS project using the *Single View Application* template.

Create a small view (around 200×200 points) in the middle of the screen. You can do this either programmatically or using Interface Builder (whichever you are more comfortable with). Just make sure that you include a property in your view controller so that you can access the small view directly. We'll call it `layerView`.

If you run the project, you should see a white square in the middle of a light gray background (see Figure 1.3). If you don't see that, you may need to tweak the background colors of the window/view.

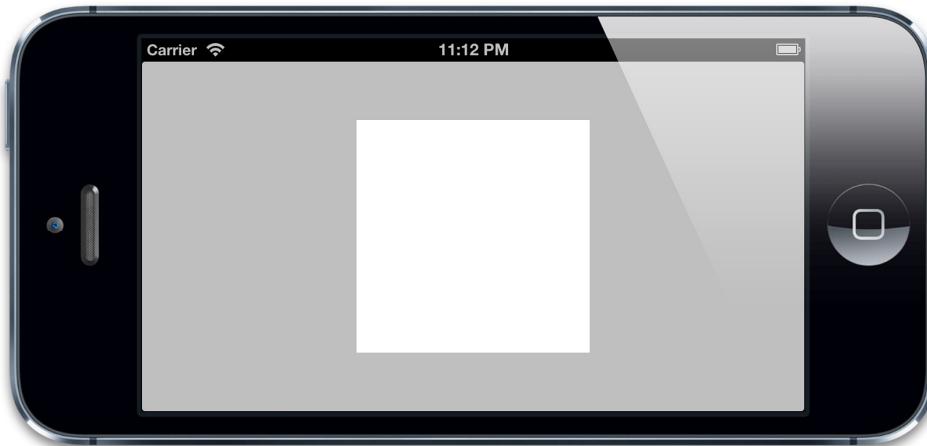


Figure 1.3 A white `UIView` on a gray background

That's not very exciting, so let's add a splash of color. We'll place a small blue square inside the white one.

We could achieve this effect by simply using another `UIView` and adding it as a subview to the one we've already created (either in code or with Interface Builder), but that wouldn't really teach us anything about layers.

Instead, let's create a `CALayer` and add it as a sublayer to the backing layer of our view. Although the `layer` property is exposed in the `UIView` class interface, the standard Xcode project templates for iOS apps do not include the Core Animation headers, so we cannot call any methods or access any properties of the layer until we add the appropriate framework to the project. To do that, first add the QuartzCore framework in the application

target's Build Phases tab (see Figure 1.4), and then import <QuartzCore/QuartzCore.h> in the view controller's .m file.

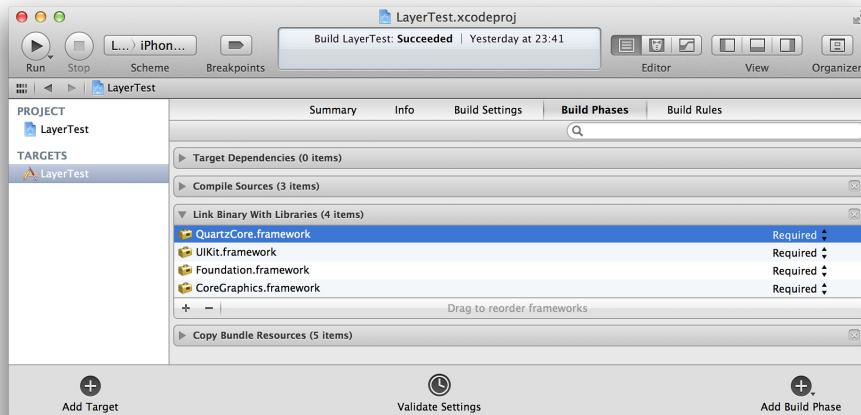


Figure 1.4 Adding the QuartzCore framework to the project

After doing that, we can directly reference `CALayer` and its properties and methods in our code. In Listing 1.1, we create a new `CALayer` programmatically, set its `backgroundColor` property, and then add it as a sublayer to the `layerView` backing layer. (The code assumes that we created the view using Interface Builder and that we have already linked up the `layerView` outlet.) Figure 1.5 shows the result.

The `CALayer` `backgroundColor` property is of type `CGColorRef`, not `UIColor` like the `UIView` class's `backgroundColor`, so we need to use the `CGColor` property of our `UIColor` object when setting the color. You can create a `CGColor` directly using Core Graphics methods if you prefer, but using `UIColor` saves you from having to manually release the color when you no longer need it.

Listing 1.1 Adding a Blue Sublayer to the View

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
```

```

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    CALayer *blueLayer = [CALayer layer];
    blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    blueLayer.backgroundColor = [UIColor blueColor].CGColor;

    //add it to our view
    [self.layerView.layer addSublayer:blueLayer];
}

@end

```

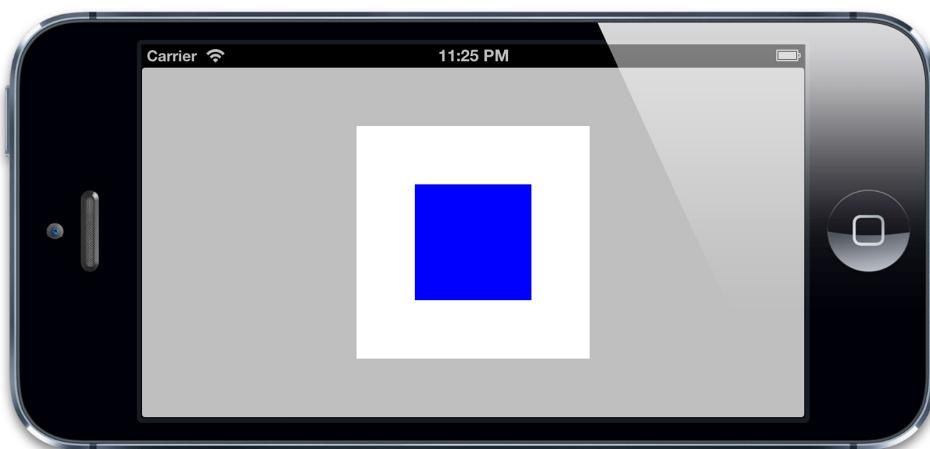


Figure 1.5 A small blue CALayer nested inside a white UIView

A view has only one *backing layer* (created automatically) but can *host* an unlimited number of additional layers. As Listing 1.1 shows, you can explicitly create standalone layers and add them directly as sublayers of the backing layer of a view. Although it is

possible to add layers in this way, more often than not you will simply work with views and their backing layers and won't need to manually create additional hosted layers.

On Mac OS, prior to version 10.8, a significant performance penalty was involved in using hierarchies of layer-backed views instead of standalone `CALayer` trees hosted inside a single view. But the lightweight `UIView` class in iOS barely has any negative impact on performance when working with layers. (In Mac OS 10.8, the performance of `NSView` is greatly improved, as well.)

The benefit of using a layer-backed view instead of a hosted `CALayer` is that while you still get access to all the low-level `CALayer` features, you don't lose out on the high-level APIs (such as autoresizing, autolayout, and event handling) provided by the `UIView` class.

You might still want to use a hosted `CALayer` instead of a layer-backed `UIView` in a real-world application for a few reasons, however:

- You might be writing cross-platform code that will also need to work on a Mac.
- You might be working with multiple `CALayer` subclasses (see Chapter 6, “Specialized Layers”) and have no desire to create new `UIView` subclasses to host them all.
- You might be doing such performance-critical work that even the negligible overhead of maintaining the extra `UIView` object makes a measurable difference (although in that case, you'll probably want to use something like OpenGL for your drawing anyway).

But these cases are rare, and in general, layer-backed views are a lot easier to work with than hosted layers.

Summary

This chapter explored the layer tree, a hierarchy of `CALayer` objects that exists in parallel beneath the `UIView` hierarchy that forms the iOS interface. We also created our own `CALayer` and added it to the layer tree as an experiment.

In Chapter 2, “The Backing Image,” we look at the `CALayer` backing image and at the properties that Core Animation provides for manipulating how it is displayed.

The Backing Image

A picture is worth a thousand words. An interface is worth a thousand pictures.

Ben Shneiderman

Chapter 1, “The Layer Tree,” introduced the `CALayer` class and created a simple layer with a blue background. Background colors are all very well, but layers would be rather boring if all they could display was a flat color. A `CALayer` can actually contain a picture of anything you like. This chapter explores the *Backing Image* of `CALayer`.

The contents Image

`CALayer` has a property called `contents`. This property’s type is defined as `id`, implying that it can be any kind of object. This is true—in the sense that you can assign any object you like to the `contents` property and your app will still compile—however, in practice, if you supply anything other than a `CGImage`, then your layer will be blank.

This quirk of the `contents` property is due to Core Animation’s Mac OS heritage. The reason that `contents` is defined as an `id` is so that on Mac OS, you can assign either a `CGImage` or an `NSImage` to the property and it will work automatically. If you try to assign a `UIImage` on iOS, however, you’ll just get a blank layer. This is a common cause of confusion for iOS developers who are new to Core Animation.

The headaches don’t stop there, though. The type you actually need to supply is a `CGImageRef`, which is a pointer to a `CGImage` struct. `UIImage` has a `CGImage` property that returns the underlying `CGImageRef`. If you try to assign that to the `CALayer` `contents` property directly, though, it won’t compile because `CGImageRef` is not really a Cocoa object; it’s a Core Foundation type.

Although Core Foundation types behave like Cocoa objects at runtime (known as *toll-free bridging*), they are not type compatible with `id` unless you use a *bridged cast*. To assign the image of a layer, you actually need to do the following:

```
layer.contents = (__bridge id)image.CGImage;
```

If you are not using ARC (Automatic Reference Counting), you do not need to include the `__bridge` part, but *why are you not using ARC?*!

Let's modify the project we created in Chapter 1 to display an image rather than a background color. We don't need the additional hosted layer any more now that we've established that it's possible to create layers programmatically, so we'll just set the image directly as the `contents` of the backing layer of our `layerView`.

Listing 2.1 shows the updated code. Figure 2.1 shows the results.

Listing 2.1 Setting a CGImage as the Layer contents

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //load an image
    UIImage *image = [UIImage imageNamed:@"Snowman.png"];

    //add it directly to our view's layer
    self.layerView.layer.contents = (__bridge id)image.CGImage;
}

@end
```



Figure 2.1 An image displayed inside the backing layer of a `UIView`

That was some very simple code, and yet we've done something quite interesting here: Using the power of `CALayer`, we've displayed an image inside an ordinary `UIView`. This isn't a `UIImageView`; it's not designed to display images normally. By manipulating the layer directly, we've exposed new functionality and made our humble `UIView` a lot more interesting.

contentsGravity

You might have noticed that our snowman looks a bit... *fat*. The image we loaded wasn't precisely square, but it's been stretched to fit the view. You've probably seen a similar situation when using `UIImageView`, and the solution there would be to set the `contentMode` property of the view to something more appropriate, like this:

```
view.contentMode = UIViewContentModeScaleAspectFit;
```

That approach works here as well (give it a try), but most visual properties of `UIView`—such as `contentMode`—are really just **manipulating equivalent properties of the underlying layer**.

The equivalent property of `CALayer` is called `contentsGravity`, and it is an `NSString` rather than an enum like its UIKit counterpart. The `contentsGravity` string should be set to one of the following constant values:

kCAGravityCenter	kCAGravityTopRight
kCAGravityTop	kCAGravityBottomLeft
kCAGravityBottom	kCAGravityBottomRight
kCAGravityLeft	kCAGravityResize
kCAGravityRight	kCAGravityResizeAspect
kCAGravityTopLeft	kCAGravityResizeAspectFill

Like `contentMode`, the purpose of `contentsGravity` is to determine how content should be aligned within the layer bounds. We will use `kCAGravityResizeAspect`, which equates to `UIViewContentModeScaleAspectFit`, and has the effect of scaling the image to fit the layer bounds without distorting its aspect ratio:

```
self.layerView.layer.contentsGravity = kCAGravityResizeAspect;
```

Figure 2.2 shows the results.

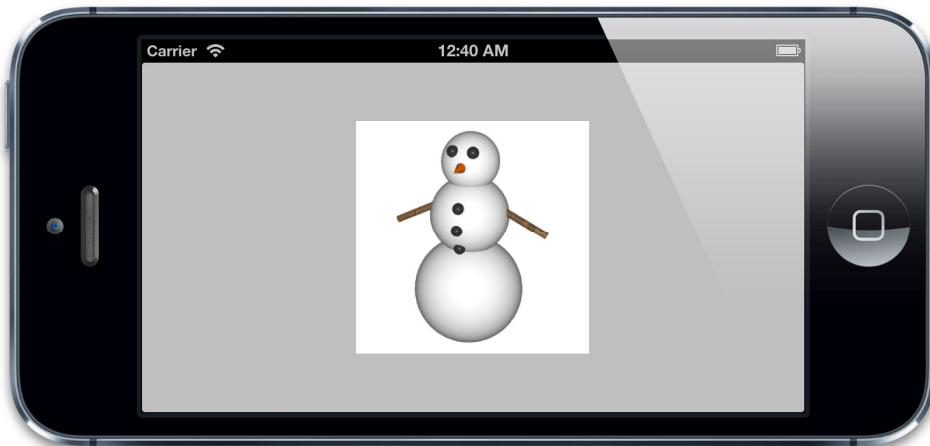


Figure 2.2 The snowman image displayed with the correct `contentsGravity`

contentsScale

The `contentsScale` property defines a ratio between the pixel dimensions of the layer's backing image and the size of the view. It's a floating-point value that defaults to 1.0.

The purpose of the `contentsScale` property is not immediately obvious. It doesn't always have the effect of scaling the backing image onscreen; if you try setting it to various values in our snowman example, you'll see it has no effect because the `contents` image is already being scaled to fit the layer bounds by the `contentsGravity` property.

If you want to simply zoom the layer contents image, you can do so using the layer's `transform` or `affineTransform` properties (see Chapter 5, "Transforms," for an explanation of transforms), but that's not the purpose of `contentsScale`.

The `contentsScale` property is actually part of the mechanism by which support for high-resolution (a.k.a. *Hi-DPI* or *Retina*) screens is implemented. It is used to determine the size of the backing image that the layer should automatically create when drawing, and the scale at which the `contents` image should be displayed (assuming that it isn't already being scaled by the `contentsGravity` setting). `UIImageView` has an equivalent-but-little-used property called `contentScaleFactor`.

If `contentsScale` is set to 1.0, drawing will be done at a resolution of 1 pixel per point. If it is set to 2.0, drawing will be done at 2 pixels per point, a.k.a. Retina resolution. (In case you are unclear on the distinction between pixels and points, this is explained later in the chapter.)

This doesn't actually make any difference when using `kCAGravityResizeAspect` because it scales the image to fit the layer, regardless of its resolution. But if we switch our `contentsGravity` to `kCAGravityCenter` instead (which doesn't scale the image), the difference will be more apparent (see Figure 2.3).



Figure 2.3 A Retina image displayed with the wrong `contentsScale` by default

As you can see, our snowman is huge and pixelated. That's because `CGImage` (unlike `UIImage`) has no internal concept of scale. When we used the `UIImage` class to load our snowman image, it correctly loaded the high-quality Retina version. But when we used the `CGImage` representation to set that image as our layer `contents`, the scale factor was lost in translation. We can fix that by manually setting the `contentsScale` to match our `UIImage` `scale` property (see Listing 2.2). Figure 2.4 shows the result.

Listing 2.2 Using contentsScale to Correct the Image Display Scale

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //load an image
    UIImage *image = [UIImage imageNamed:@"Snowman.png"];

    //add it directly to our view's layer
    self.layerView.layer.contents = (__bridge id)image.CGImage;

    //center the image
    self.layerView.layer.contentsGravity = kCAGravityCenter;

    //set the contentsScale to match image
    self.layerView.layer.contentsScale = image.scale;
}

@end
```

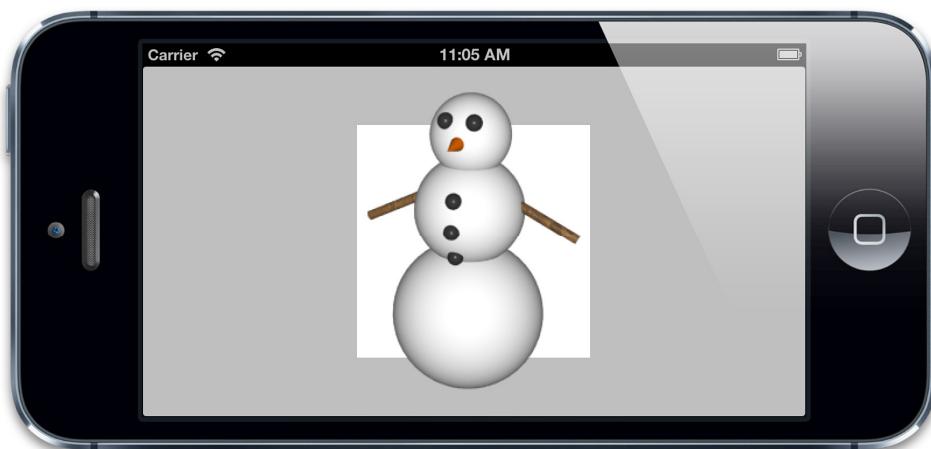


Figure 2.4 The same Retina image displayed with the correct contentsScale

When working with backing images that are generated programmatically, you'll often need to remember to manually set the layer `contentsScale` to match the screen scale; otherwise, your images will appear pixelated on Retina devices. You do so like this:

```
layer.contentsScale = [UIScreen mainScreen].scale;
```

masksToBounds

Now that our snowman is being displayed at the correct size, you might have noticed something else about him—he's poking outside of the view bounds. By default, `UIView` will happily draw content and subviews outside of its designated bounds. The same is true for `CALayer`.

There is a property on `UIView` called `clipsToBounds` that can be used to enable/disable clipping (that is, to control whether a view's contents are allowed to spill out of their frame). `CALayer` has an equivalent property called `masksToBounds`. By setting the property to YES, we can keep our snowman confined (see Figure 2.5).



Figure 2.5 Using `masksToBounds` to clip the layer contents

contentsRect

The `contentsRect` property of `CALayer` allows us to specify a subrectangle of the backing image to be displayed inside the layer frame. This allows for much greater flexibility than the `contentsGravity` property in terms of how the image is cropped and stretched.

Unlike bounds and frame, contentsRect is not measured in points; it uses *unit coordinates*. Unit coordinates are specified in the range 0 to 1, and are *relative* values (as opposed to *absolute* values like points and pixels). In this case, they are relative to the backing image's dimensions. The following coordinate types are used in iOS:

- **Points**—The most commonly used coordinate type on iOS and Mac OS. Points are *virtual* pixels, also known as *logical* pixels. On standard-definition devices, 1 point equates to 1 pixel, but on Retina devices, a point equates to 2×2 physical pixels. iOS uses points for all screen coordinate measurements so that layouts work seamlessly on both Retina and non-Retina devices.
- **Pixels**—Physical pixel coordinates are not used for screen layout, but they are often still relevant when working with images. `UIImage` is screen-resolution aware, and specifies its size in points, but some lower-level image representations such as `CGImage` use pixel dimensions, so you should keep in mind that their stated size will not match their display size on a Retina device.
- **Unit**—Unit coordinates are a convenient way to specify measurements that are relative to the size of an image or a layer's bounds, and so do not need to be adjusted if that size changes. Unit coordinates are used a lot in OpenGL for things like texture coordinates, and they are also used frequently in Core Animation.

The default contentsRect is $\{0, 0, 1, 1\}$, which means that the entire backing image is visible by default. If we specify a smaller rectangle, the image will be clipped (see Figure 2.6).

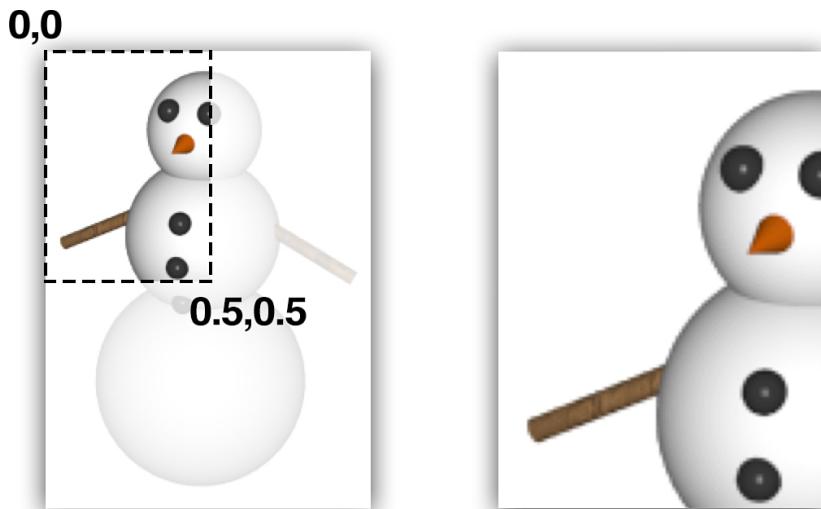


Figure 2.6 A custom contentsRect (left) and the displayed contents (right)

It is actually possible to specify a `contentsRect` with a negative origin or with dimensions larger than `{1, 1}`. In this case, the outermost pixels of the image will be stretched to fill the remaining area.

One of the most interesting applications of `contentsRect` is that it enables the use of so-called *image sprites*. If you've ever done any games programming, you'll be familiar with the concept of sprites, which are really just images that can be moved around the screen independently of one another. But outside of the gaming world, the term is usually used to refer to a common technique for *loading* sprite images, rather than having anything to do with movement.

Typically, many sprites will be packed into a single large image that can be loaded in one go. This carries various benefits over using multiple individual image files in terms of memory usage, load time, and rendering performance.

Sprites are used in 2D game engines like Cocos2D, which uses OpenGL to display the images. But we can use sprites in an ordinary UIKit application by leveraging the power of `contentsRect`.

To start off with, we need a *sprite sheet*—a large image containing our smaller sprite images. Figure 2.7 shows an example sprite sheet.

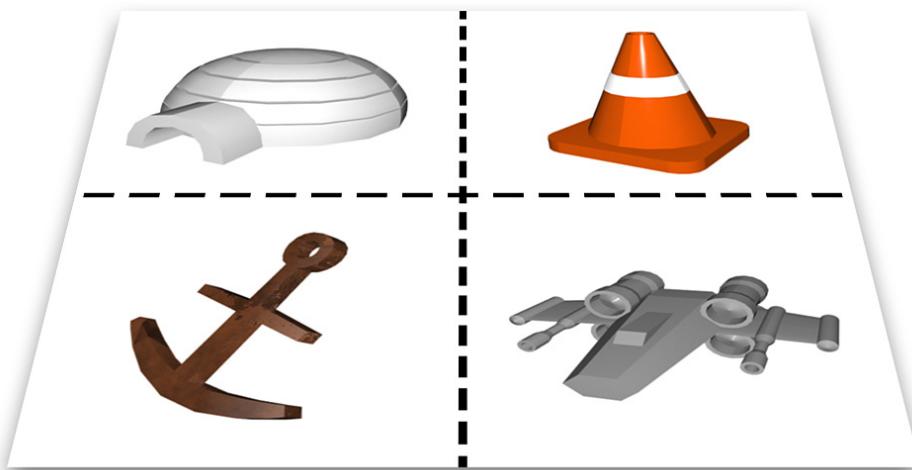


Figure 2.7 A sprite sheet

Next, we need to load and display these sprites in our app. The principle is quite simple: We load our large image as normal, assign it as the `contents` for four separate layers

(one for each sprite), and then set the `contentsRect` of each of them to mask off the parts we don't want.

We need to add some additional views to our project for the sprite layers. (These views were positioned using Interface Builder to avoid cluttering the code, but you could create them programmatically if you prefer.) Listing 2.3 shows the code, and Figure 2.8 shows the end result.

Listing 2.3 Splitting Up a Sprite Sheet Using `contentsRect`

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *coneView;
@property (nonatomic, weak) IBOutlet UIView *shipView;
@property (nonatomic, weak) IBOutlet UIView *iglooView;
@property (nonatomic, weak) IBOutlet UIView *anchorView;

@end

@implementation ViewController

- (void)addSpriteImage:(UIImage *)image
    withContentRect:(CGRect)rect
    toLayer:(CALayer *)layer
{
    //set image
    layer.contents = (__bridge id)image.CGImage;

    //scale contents to fit
    layer.contentsGravity = kCAGravityResizeAspect;

    //set contentsRect
    layer.contentsRect = rect;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //load sprite sheet
    UIImage *image = [UIImage imageNamed:@"Sprites.png"];

    //set igloo sprite
    [self addSpriteImage:image
        withContentRect:CGRectMake(0, 0, 0.5, 0.5)
        toLayer:self.iglooView.layer];
}
```

```

//set cone sprite
[self addSpriteImage:image
    withContentRect:CGRectMakeMake(0.5, 0, 0.5, 0.5)
    toLayer:self.coneView.layer];

//set anchor sprite
[self addSpriteImage:image
    withContentRect:CGRectMakeMake(0, 0.5, 0.5, 0.5)
    toLayer:self.anchorView.layer];

//set spaceship sprite
[self addSpriteImage:image
    withContentRect:CGRectMakeMake(0.5, 0.5, 0.5, 0.5)
    toLayer:self.shipView.layer];
}

@end

```

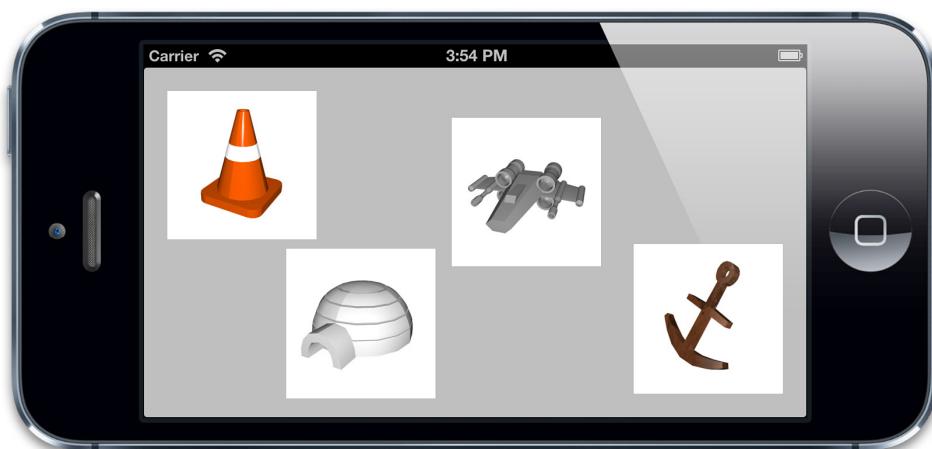


Figure 2.8 Four sprites, arranged randomly onscreen

Sprite sheets are a neat way of reducing app size and loading performance (a single large image compresses better and loads quicker than multiple small ones), but they can be cumbersome to arrange manually, and they create a maintenance burden if you need to add

new sprites or need to change the dimensions of any of the existing ones after the sheet has been created.

Several commercial applications are available for creating sprite sheets automatically on your Mac. These tools simplify the use of sprites by automatically generating an XML or Property List (Plist) file containing the sprite coordinates. This file can then be loaded along with the image and used to set the `contentsRect` for each sprite, instead of the developer having to manually code the positions into the application.

These files are usually designed to be used in OpenGL games, but if you are interested in using sprite sheets in a regular app, the open source LayerSprites library (<https://github.com/nicklockwood/LayerSprites>) can read sprite sheets in the popular Cocos2D format and display them using ordinary Core Animation layers.

contentsCenter

The last contents-related property we look at in this chapter is `contentsCenter`. You might expect from the name that `contentsCenter` would have something to do with the position of the `contents` image, but the name is misleading. The `contentsCenter` is actually a `CGRect` that defines a stretchable region inside the layer and a fixed border around the edge. Changing the `contentsCenter` makes no difference to how the backing image is displayed, *until the layer is resized*, and then its purpose becomes clear.

By default, the `contentsCenter` is set to `{0, 0, 1, 1}`, which means that the backing image will stretch uniformly when the layer is resized (depending on the `contentsGravity`). But if we increase the origin values and reduce the size, we can create a border around the image. Figure 2.9 shows how the scaling works for a `contentsCenter` value of `{0.25, 0.25, 0.5, 0.5}`.

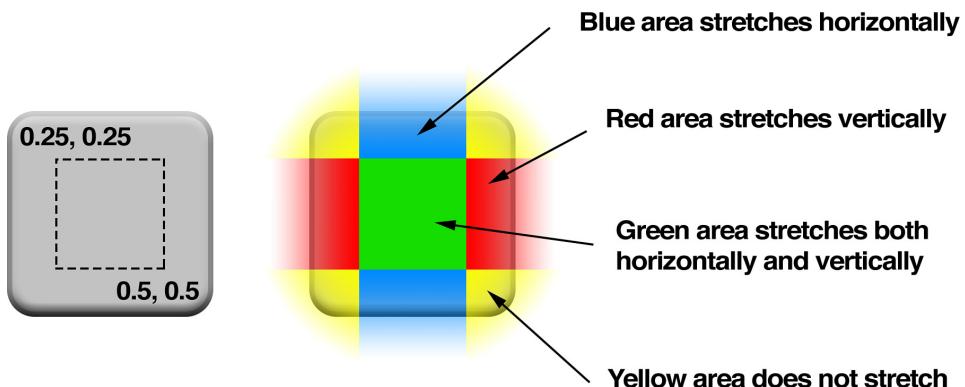


Figure 2.9 An example of a `contentsCenter` rectangle, and the effect it has on the image

This means that we can resize our views arbitrarily and the border will remain consistent (see Figure 2.10). This works in a similar way to the `-resizableImageWithCapInsets:` method of `UIImage`, but can be applied to any layer backing image, including one that is drawn at runtime using Core Graphics (as covered later in this chapter).

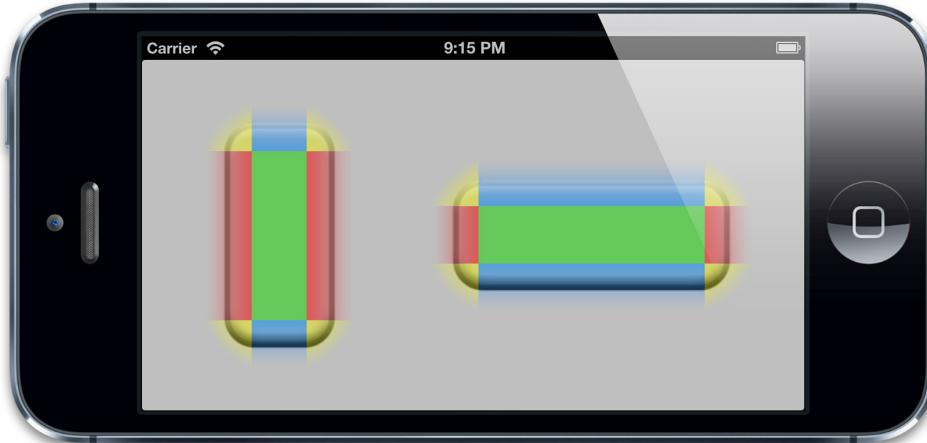


Figure 2.10 A couple of views using the same stretchable backing image

Listing 2.4 shows the code for setting up these stretchable views programmatically. However, an additional cool feature of `contentsCenter` is that it can be configured in Interface Builder without writing any code at all by using the Stretching controls in the Inspector window, as shown in Figure 2.11.

Listing 2.4 Setting Up Stretchable Views Using `contentsCenter`

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *button1;  
@property (nonatomic, weak) IBOutlet UIView *button2;  
  
@end  
  
@implementation ViewController  
  
- (void)addStretchableImage:(UIImage *)image  
withContentCenter:(CGRect)rect
```

```
        toLayer:(CALayer *)layer
{
    //set image
    layer.contents = (__bridge id)image.CGImage;

    //set contentsCenter
    layer.contentsCenter = rect;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //load button image
    UIImage *image = [UIImage imageNamed:@"Button.png"];

    //set button 1
    [self addStretchableImage:image
        withContentCenter:CGRectMake(0.25, 0.25, 0.5, 0.5)
        toLayer:self.button1.layer];

    //set button 2
    [self addStretchableImage:image
        withContentCenter:CGRectMake(0.25, 0.25, 0.5, 0.5)
        toLayer:self.button2.layer];
}

@end
```

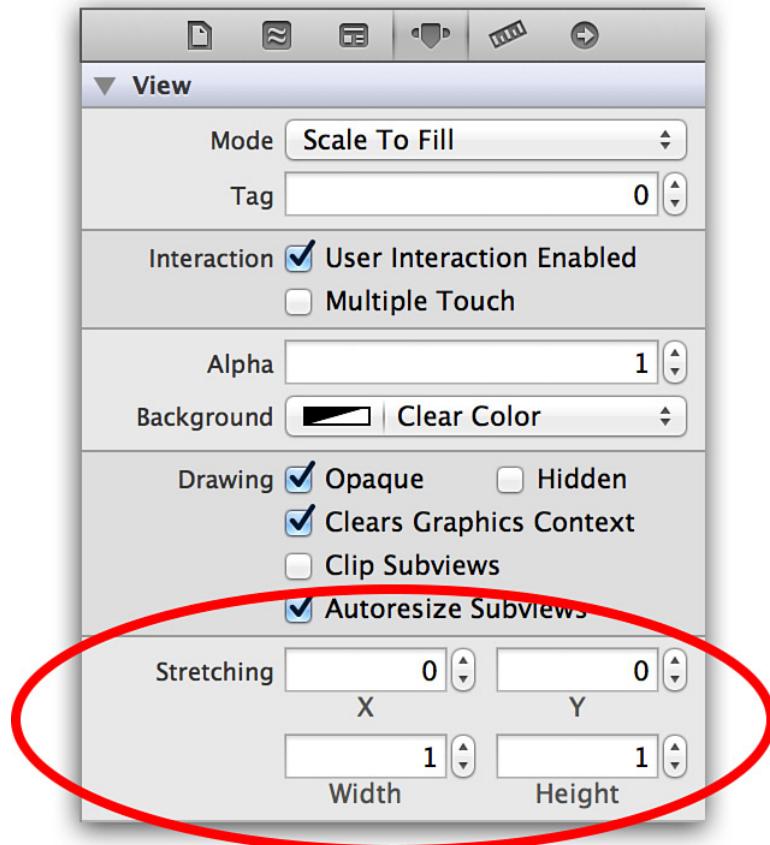


Figure 2.11 The Interface Builder Inspector controls for `contentsCenter`

Custom Drawing

Setting the layer contents with a `CGImage` is not the only way to populate the backing image. It is also possible to draw directly into the backing image using Core Graphics. The `-drawRect:` method can be implemented in a `UIView` subclass to implement custom drawing.

The `-drawRect:` method has no default implementation because a `UIView` does not require a custom backing image if it is just filled with a solid color or if the underlying layer's contents property contains an existing image instance. If `UIView` detects that

the `-drawRect:` method is present, it allocates a new backing image for the view, with pixel dimensions equal to the view size multiplied by the `contentsScale`.

If you don't need this backing image, it's a waste of memory and CPU time to create it, which is why Apple recommends that you don't leave an empty `-drawRect:` method in your layer subclasses if you don't intend to do any custom drawing.

The `-drawRect:` method is executed automatically when the view first appears onscreen. The code inside `-drawRect:` method uses Core Graphics to draw into the backing image, and the result will then be cached until the view needs to update it (usually because the developer has called the `-setNeedsDisplay` method, although some view types will be redrawn automatically whenever a property that affects their appearance is changed [such as `bounds`]). Although `-drawRect:` is a `UIView` method, it's actually the underlying `CALayer` that schedules the drawing and stores the resultant image.

`CALayer` has an optional delegate property that conforms to the `CALayerDelegate` protocol. When `CALayer` requires content-specific information, it requests it from the delegate. `CALayerDelegate` is an *informal* protocol, which is a fancy way of saying that there is no actual `CALayerDelegate` @protocol that you can reference in your class interface. You just add the methods you need and `CALayer` will call them if present. (The `delegate` property is just declared as an `id`, and all the delegate methods are treated as optional.)

When it needs to be redrawn, `CALayer` asks its delegate to supply a backing image for it to display. It does this by attempting to call the following method:

```
- (void)displayLayer:(CALayer *)layer;
```

This is an opportunity for the delegate to set the layer `contents` property directly if it wants to, in which case no further methods will be called. If the delegate does not implement the `-displayLayer:` method, `CALayer` attempts to call the following method instead:

```
- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)ctx;
```

Before calling this method, `CALayer` creates an empty backing image of a suitable size (based on the layer `bounds` and `contentsScale`) and a Core Graphics drawing context suitable for drawing into that image, which it passes as the `ctx` parameter.

Let's modify the test project from Chapter 1 so that it implements the `CALayerDelegate` protocol and does some drawing (see Listing 2.5). Figure 2.12 shows the result.

Listing 2.5 Implementing the `CALayerDelegate`

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
//create sublayer
CALayer *blueLayer = [CALayer layer];
blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
blueLayer.backgroundColor = [UIColor blueColor].CGColor;

//set controller as layer delegate
blueLayer.delegate = self;

//ensure that layer backing image uses correct scale
blueLayer.contentsScale = [UIScreen mainScreen].scale;

//add layer to our view
[self.layerView.layer addSublayer:blueLayer];

//force layer to redraw
[blueLayer display];
}

- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)ctx
{
    //draw a thick red circle
    CGContextSetLineWidth(ctx, 10.0f);
    CGContextSetStrokeColorWithColor(ctx, [UIColor redColor].CGColor);
    CGContextStrokeEllipseInRect(ctx, layer.bounds);
}

@end
```

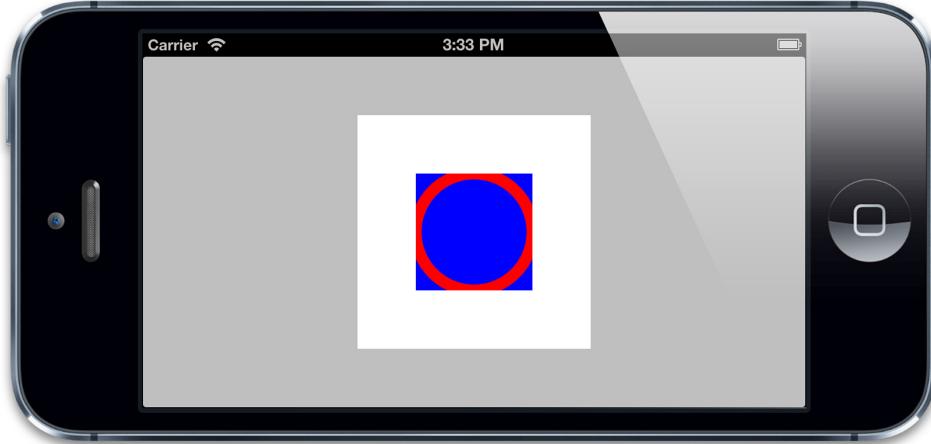


Figure 2.12 A layer with backing image drawn using `CALayerDelegate`

Note a couple of interesting things here:

- We have to manually call `-display` on `blueLayer` to force it to be updated. Unlike `UIView`, `CALayer` does not redraw its contents automatically when it appears onscreen; it is left to the discretion of the developer to decide when the layer needs redrawing.
- The circle that we have drawn is clipped to the layer bounds even though we have not enabled the `masksToBounds` property. That's because when you draw the backing image using the `CALayerDelegate`, the `CALayer` creates a drawing context with the exact dimensions of the layer. There is no provision made for drawing that spills outside of those bounds.

So now you understand the `CALayerDelegate` and how to use it. But unless you are creating standalone layers, you will almost never need to implement the `CALayerDelegate` protocol. The reason for this is that when `UIView` creates its backing layer, it automatically sets itself as the layer's `delegate` and provides an implementation for `-displayLayer`: that abstracts these issues away.

When using view-backing layers, you do not need to implement `-displayLayer`: or `-drawLayer:inContext:` to draw into your layer's backing image; you can just implement the `-drawRect:` method of `UIView` in the normal fashion, and `UIView` takes care of everything, including automatically calling `-display` on the layer when it needs to be redrawn.

Summary

This chapter explored the layer backing image and its associated properties. You learned to position and crop the image, cut individual images out of a sprite sheet, and to draw layer contents on-the-fly using the `CALayerDelegate` and Core Graphics.

In Chapter 3, “Layer Geometry,” we look at the geometry of a layer and examine how layers are positioned and resized relative to one another.

This page intentionally left blank

3

Layer Geometry

Let no one unversed in geometry enter here.

Sign above the entrance to Plato's Academy

Chapter 2, “The Backing Image,” introduced the layer-backing image and the properties used to control its position and scaling within the layer bounds. In this chapter, we look at how the layer itself is positioned and sized with respect to its superlayer and siblings. We also explore how to manage your layer’s geometry and how it is affected by autoresizing and autolayout.

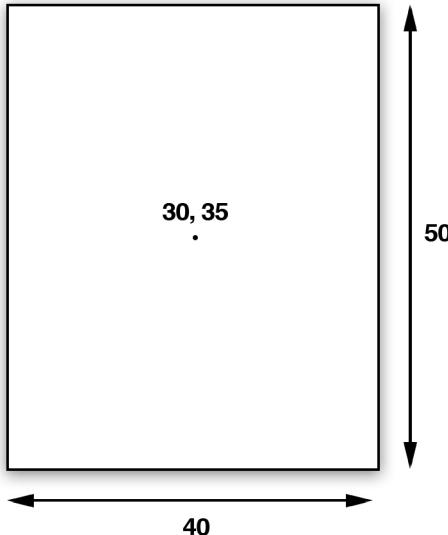
Layout

`UIView` has three primary layout properties: `frame`, `bounds`, and `center`. `CALayer` has equivalents called `frame`, `bounds`, and `position`. Why they used “position” for layers and “center” for views will become clear, but they both represent the same value.

The `frame` represents the *outer* coordinates of the layer (that is, the space it occupies within its superlayer), the `bounds` property represents the *inner* coordinates (with `{0, 0}` typically equating to the top-left corner of the layer, although this is not always the case), and the `center` and `position` both represent the location of the `anchorPoint` relative to the superlayer. The `anchorPoint` property is explained later, but for now just think of it as the center of the layer. Figure 3.1 shows how these properties relate to one another.

The view’s `frame`, `bounds`, and `center` properties are actually just *accessors* (setter and getter methods) for the underlying layer equivalents. When you manipulate the view `frame`, you are really changing the `frame` of the underlying `CALayer`. You cannot change the view’s `frame` independently of its layer.

10, 10



View:

frame = {10, 10, 40, 50}

bounds = {0, 0, 40, 50}

center = {30, 35}

Layer:

frame = {10, 10, 40, 50}

bounds = {0, 0, 40, 50}

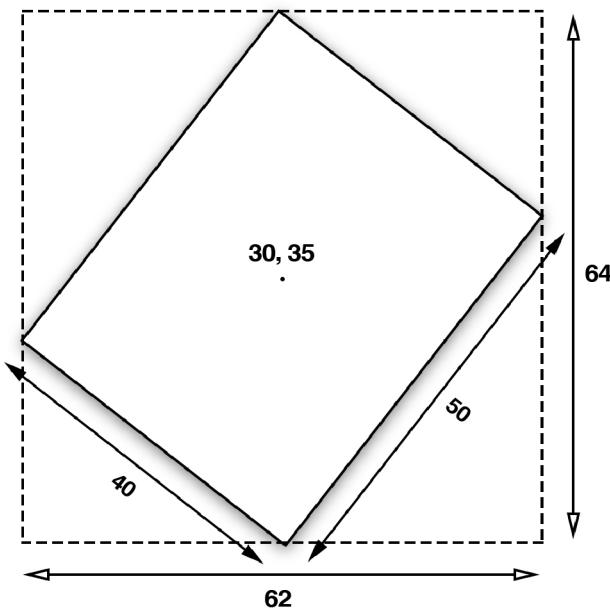
position = {30, 35}

Figure 3.1 `UIView` and `CALayer` coordinate systems (with example values)

The `frame` is not really a distinct property of the view or layer at all; it is a *virtual* property, computed from the `bounds`, `position`, and `transform`, and therefore changes when any of those properties are modified. Conversely, changing the `frame` may affect any or all of those values, as well.

You need to be mindful of this when you start working with transforms, because when a layer is rotated or scaled, its `frame` reflects the total axially aligned rectangular area occupied by the transformed layer within its parent, which means that the `frame` width and height may no longer match the `bounds` (see Figure 3.2).

-1, 3



View:

`frame = {-1, 3, 62, 64}`

`bounds = {0, 0, 40, 50}`

`center = {30, 35}`

Layer:

`frame = {-1, 3, 62, 64}`

`bounds = {0, 0, 40, 50}`

`position = {30, 35}`

Figure 3.2 The effect that rotating a view or layer has on its `frame` property

anchorPoint

As mentioned earlier, both the view's `center` property and the layer's `position` property specify the location of the `anchorPoint` of the layer relative to its superlayer. The `anchorPoint` property of a layer controls how the layer's frame is positioned relative to its `position` property. You can think of the `anchorPoint` as being the *handle* used to move the layer around.

By default, the `anchorPoint` is located in the center of the layer, so that the layer will be centered around that position, wherever it might be. The `anchorPoint` is not exposed in the `UIView` interface, which is why the view's `position` property is just called "center." But the `anchorPoint` of a layer can be moved. You could place it at the top left of the layer frame, for example, so that the layer's contents would extend down and to the right of its `position` (see Figure 3.3) instead of being centered on it.

Like the `contentsRect` and `contentsCenter` properties covered in Chapter 2, the `anchorPoint` is specified in *unit coordinates*, meaning that its coordinates are relative to the dimensions of the layer. The top-left corner of the layer is `{0, 0}`, and the bottom-right

corner is $\{1, 1\}$. The default (center) position is therefore $\{0.5, 0.5\}$. The `anchorPoint` can be placed *outside* of the layer bounds by specifying x or y values that are less than zero or greater than one.

Note in Figure 3.3 that when we change the `anchorPoint`, the `position` property does not change. Instead, the `position` remains fixed and the `frame` of the layer moves.

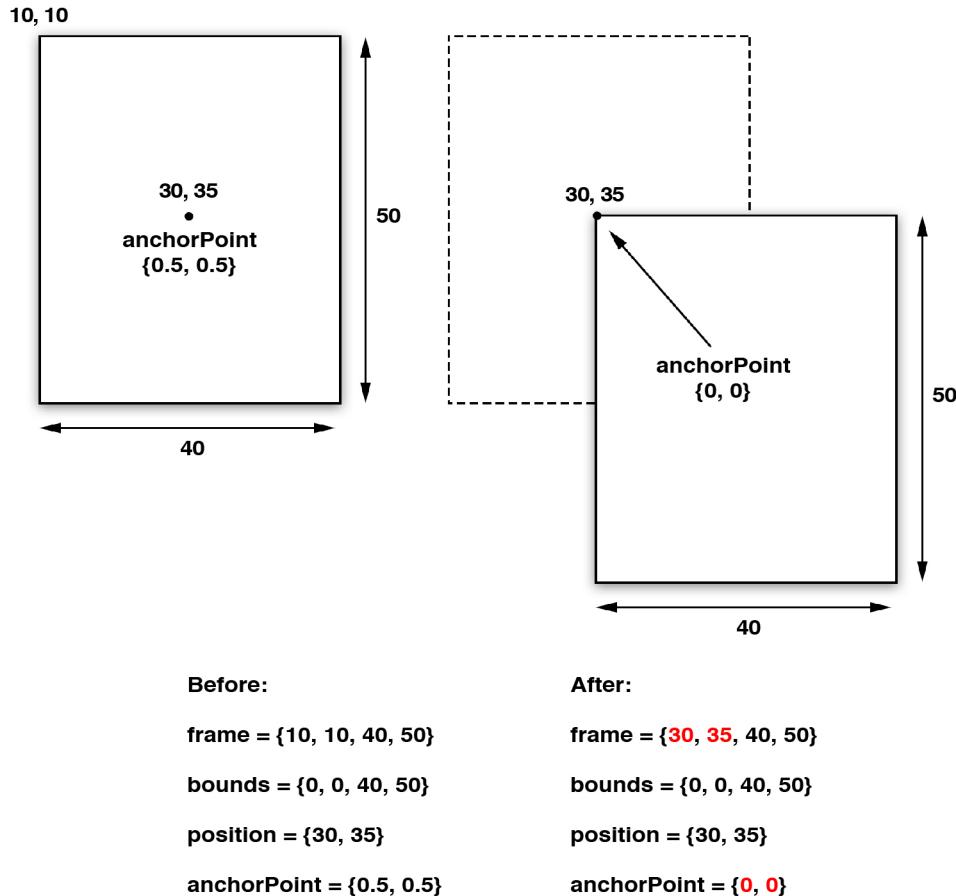


Figure 3.3 The effect that changing the `anchorPoint` has on the frame

So, why would we want to change the `anchorPoint`? We can already place the frame wherever we want, so doesn't changing the `anchorPoint` just cause confusion? To illustrate why this might be useful, let's try a practical example. Let's build an analog clock face with moving hour, minute, and second hands.

The face and hands are constructed using four images (see Figure 3.4). To keep things simple, we'll load and display these images in the traditional way, using four separate `UIImageView` instances (although we could do this using regular views by setting their backing layer `contents images`).

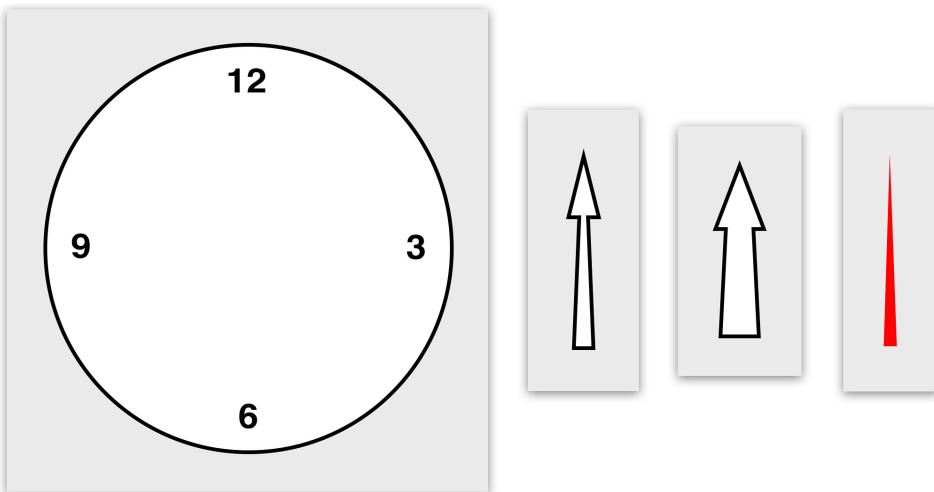


Figure 3.4 The four images that make up the clock face and hands

The clock components are arranged in Interface Builder (see Figure 3.5). The image views are nested inside another container view and have both autoresizing and autolayout disabled. This is because autoresizing acts on the view `frame`, and as demonstrated in Figure 3.2, the `frame` changes when the view is rotated, which will lead to layout glitches if a rotated view's `frame` is resized.

We'll use an `NSTimer` to update our clock, and make use of the view's `transform` property to rotate the hands. (If you are not familiar with that property, don't worry; we cover it in Chapter 5, "Transforms.") Listing 3.1 shows the code for our clock.

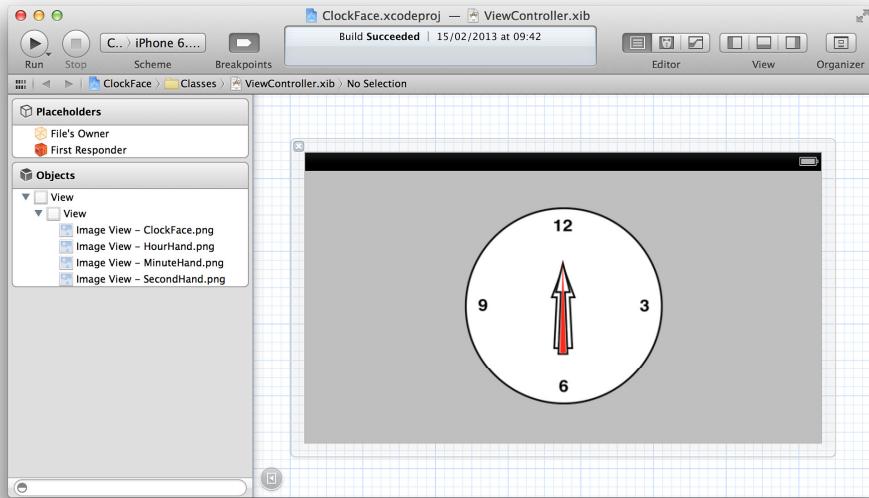


Figure 3.5 Laying out the clock views in Interface Builder

Listing 3.1 Clock

```

        repeats:YES];

//set initial hand positions
[self tick];
}

- (void)tick
{
    //convert time to hours, minutes and seconds
    NSCalendar *calendar =
        [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];

   NSUInteger units = NSHourCalendarUnit |
                       NSMinuteCalendarUnit |
                       NSSecondCalendarUnit;

    NSDateComponents *components = [calendar components:units
                                                fromDate:[NSDate date]];

    //calculate hour hand angle
    CGFloat hoursAngle = (components.hour / 12.0) * M_PI * 2.0;

    //calculate minute hand angle
    CGFloat minsAngle = (components.minute / 60.0) * M_PI * 2.0;

    //calculate second hand angle
    CGFloat secsAngle = (components.second / 60.0) * M_PI * 2.0;

    //rotate hands
    self.hourHand.transform = CGAffineTransformMakeRotation(hoursAngle);
    self.minuteHand.transform = CGAffineTransformMakeRotation(minsAngle);
    self.secondHand.transform = CGAffineTransformMakeRotation(secsAngle);
}

@end

```

When we run the clock app, it looks a bit strange (see Figure 3.6). The reason for this is that the hand images are rotating around the center of the image, which is not where we would expect a clock's hand to pivot.

You might think that this could be fixed by adjusting the position of the hand images in Interface Builder, but that won't work because the images will not rotate correctly if they are not centered on the clock face.

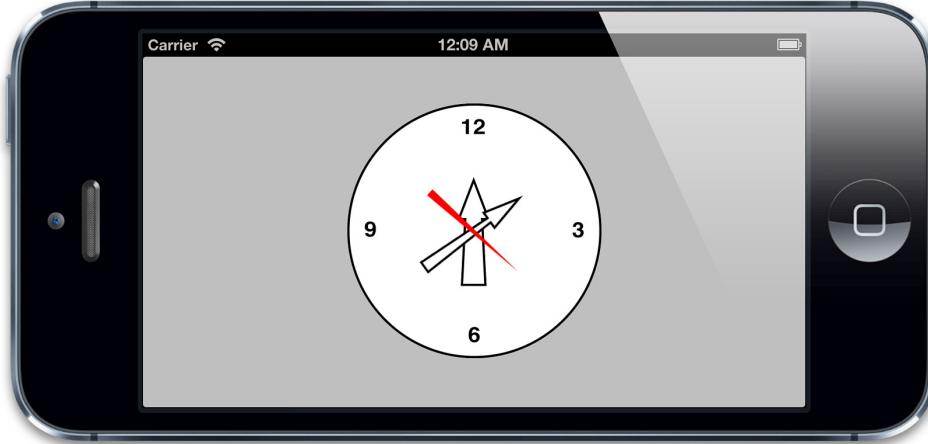


Figure 3.6 The clock face, with misaligned hands

A solution that *would* work is to add additional transparent space at the bottom of all the images, but that makes the images larger than they need to be, and they will consume more memory as a result. That's not very elegant.

A better solution is to make use of the `anchorPoint` property. Let's add some additional lines of code to our `-viewDidLoad` method to offset the `anchorPoint` for each of our clock hands (see Listing 3.2). Figure 3.7 shows the correctly aligned hands.

Listing 3.2 Clock with Adjusted `anchorPoint` Values

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //adjust anchor points
    self.secondHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.minuteHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.hourHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);

    //start timer
    ...
}
```

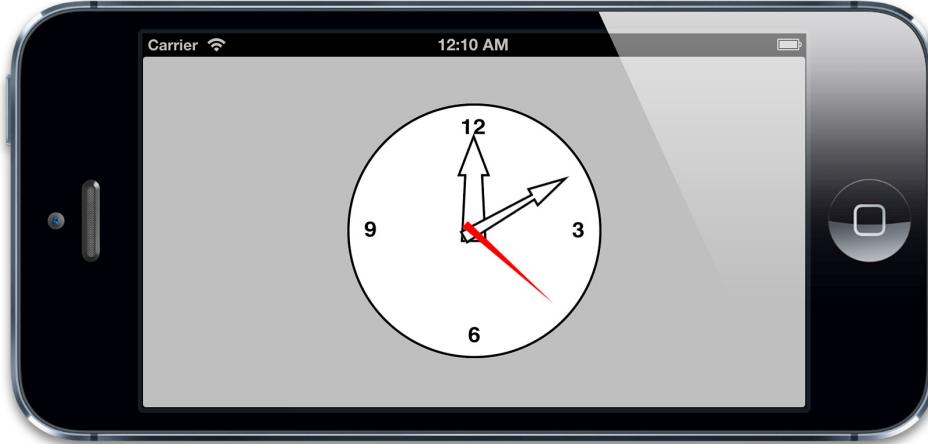


Figure 3.7 The clock face, with correctly aligned hands

Coordinate Systems

Layers, like views, are positioned hierarchically, with each placed relative to its parent in the layer tree. The position of a layer is relative to the bounds of its superlayer. If the superlayer moves, so do all of its sublayers.

This is convenient when positioning layers because it allows you to move a subtree of several layers as a single unit just by moving the root layer. But sometimes you need to know the *absolute* position of a layer or (more commonly) its position relative to a layer other than its immediate parent.

`CALayer` provides some utility methods for converting between different layers' coordinate systems:

- `(CGPoint)convertPoint:(CGPoint)point fromLayer:(CALayer *)layer;`
- `(CGPoint)convertPoint:(CGPoint)point toLayer:(CALayer *)layer;`
- `(CGRect)convertRect:(CGRect)rect fromLayer:(CALayer *)layer;`
- `(CGRect)convertRect:(CGRect)rect toLayer:(CALayer *)layer;`

These methods enable you to take either a point or rectangle defined in the coordinate system of one layer and convert it to the coordinate system of another.

Flipped Geometry

Conventionally, on iOS the position of a layer is specified relative to the top-left corner of its superlayer's bounds. On Mac OS, the convention is that the position is relative to

the *bottom*-left corner. Core Animation can support either of these conventions by virtue of the `geometryFlipped` property. This is a `BOOL` value that determines whether the geometry of a layer is vertically flipped with respect to its superlayer. Setting this property to `YES` for a layer on iOS means that its sublayers will be flipped vertically and will be positioned relative to the *bottom* of its bounds rather than the top as normal (as will all of their sublayers, and so on, unless they also have `YES` for their `geometryFlipped` property).

The Z Axis

Unlike `UIView`, which is strictly two-dimensional, `CALayer` exists in three-dimensional space. In addition to the `position` and `anchorPoint` properties that we have already discussed, `CALayer` has two additional properties, `zPosition` and `anchorPointZ`, both of which are floating-point values describing the layer's position on the Z axis.

Note that there is no `depth` property to complement the `bounds` width and height. Layers are fundamentally flat objects. You can think of them as being a bit like stiff sheets of paper that are individually two-dimensional but that can be glued together to form hollow, origami-like 3D structures.

The `zPosition` property is not particularly useful in most cases. In Chapter 5, we explore `CATransform3D`, and you learn how to move and rotate layers in three dimensions. But without using transforms, the only practical use you are likely to find for the `zPosition` property is to change the *display order* of your layers.

Normally, layers are drawn according to the order in which they appear in the `sublayers` array of their superlayer. This is known as the *painter's algorithm* because—like a painter painting a wall—layers that are painted later will obscure the layers that were painted earlier. But by increasing the `zPosition` of a layer, you can *move it forward* toward the camera so that it is physically *in front* of all other layers (or at least, in front of any layer with a lower `zPosition` value).

The “camera” in this case is just how we refer to the user’s viewpoint. It has nothing to do with the camera built in to the back of the iPhone (although it does point in the same direction, coincidentally).

Figure 3.8 shows a pair of views arranged in Interface Builder. As you can see, the green view—which appears first in the view hierarchy—is drawn underneath the red view, which appears later in the list.

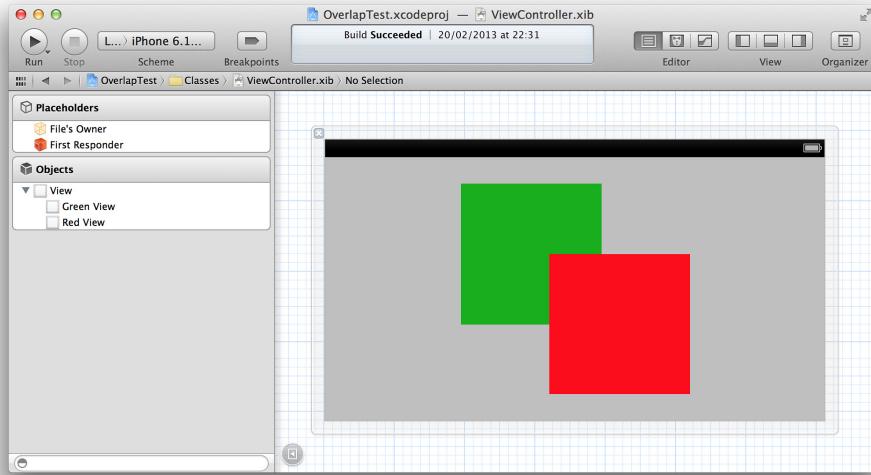


Figure 3.8 The green view is positioned underneath the red one in the view hierarchy.

We would expect this drawing order to be reflected in the actual app, as well, but if we increase the `zPosition` of the green view (see Listing 3.3), we find that the order is reversed (see Figure 3.9). Note that we don't need to increase it by much; views are infinitely thin, so even a 1-point increase in the `zPosition` brings the green view in front of the red one. A smaller value such as 0.1 or 0.0001 would also work, but be wary of using very tiny values because this can lead to visual glitches as a result of rounding errors in the floating-point calculations.

Listing 3.3 Adjusting `zPosition` to Change the Display Order

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *greenView;  
@property (nonatomic, weak) IBOutlet UIView *redView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];
```

```
//move the green view zPosition nearer to the camera  
self.greenView.layer.zPosition = 1.0f;  
}  
  
@end
```

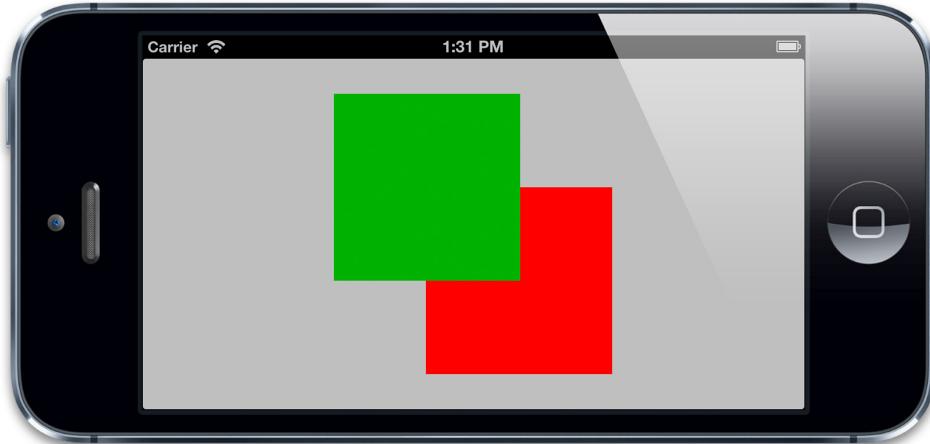


Figure 3.9 The green view is drawn in front of the red view.

Hit Testing

Chapter 1, “The Layer Tree,” stated that it’s usually preferable to use views with backing layers rather than constructing standalone layer hierarchies. One of the reasons for this is because of the extra complexity of handling touch events when using layers.

CALayer does not have any knowledge of the responder chain, so it cannot deal with touch events or gestures directly. It does have a couple of methods to help you to implement touch handling yourself, however: `-containsPoint:` and `-hitTest:`.

The `-containsPoint:` method accepts a `CGPoint` in the layer’s own coordinate system, and returns YES if the point lies inside the layer’s frame. Listing 3.4 shows the code for an adapted version of the project from Chapter 1 that uses the `-containsPoint:` method to determine whether either the white or blue layers are being touched (see Figure 3.10). This is made somewhat awkward by the need to convert the touch position to each layer’s coordinate system in turn.

Listing 3.4 Determining the Touched Layer Using containsPoint:

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) CALayer *blueLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    self.blueLayer = [CALayer layer];
    self.blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    self.blueLayer.backgroundColor = [UIColor blueColor].CGColor;

    //add it to our view
    [self.layerView.layer addSublayer:self.blueLayer];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get touch position relative to main view
    CGPoint point = [[touches anyObject] locationInView:self.view];

    //convert point to the white layer's coordinates
    point = [self.layerView.layer convertPoint:point
                                         fromLayer:self.view.layer];

    //get layer using containsPoint:
    if ([self.layerView.layer containsPoint:point])
    {
        //convert point to blueLayer's coordinates
        point = [self.blueLayer convertPoint:point
                                         fromLayer:self.layerView.layer];

        if ([self.blueLayer containsPoint:point])
        {
            [[[UIAlertView alloc] initWithTitle:@"Inside Blue Layer"
                                         message:nil
                                         delegate:nil
                                         cancelButtonTitle:@"OK"
                                         otherButtonTitles:nil] show];
        }
    }
}
```

```

        otherButtonTitles:nil] show];
    }
    else
    {
        [[[UIAlertView alloc] initWithTitle:@"Inside White Layer"
                                      message:nil
                                      delegate:nil
                                      cancelButtonTitle:@"OK"
                                      otherButtonTitles:nil] show];
    }
}
}

@end

```



Figure 3.10 The touched layer is correctly identified.

The `-hitTest:` method also accepts a `CGPoint`; but instead of a `BOOL`, it returns either the layer itself or the deepest sublayer containing the point. This means that you do not need to transform and test the point against each sublayer in turn manually, as you do when using the `-containsPoint:` method. If the point lies outside of the outermost layer's bounds, it returns `nil`. Listing 3.5 shows the code for determining the touched layer by using the `-hitTest:` method.

Listing 3.5 Determining the Touched Layer Using `hitTest`:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get touch position
    CGPoint point = [[touches anyObject] locationInView:self.view];

    //get touched layer
    CALayer *layer = [self.layerView.layer hitTest:point];

    //get layer using hitTest
    if (layer == self.blueLayer)
    {
        [[[UIAlertView alloc] initWithTitle:@"Inside Blue Layer"
                                     message:nil
                                     delegate:nil
                                     cancelButtonTitle:@"OK"
                                     otherButtonTitles:nil] show];
    }
    else if (layer == self.layerView.layer)
    {
        [[[UIAlertView alloc] initWithTitle:@"Inside White Layer"
                                     message:nil
                                     delegate:nil
                                     cancelButtonTitle:@"OK"
                                     otherButtonTitles:nil] show];
    }
}
```

You should note that when calling a layer's `-hitTest:` method (and this applies to `UIView` touch handling as well, incidentally), the order of testing is based strictly on the order of layers within the layer tree. The `zPosition` property that we mentioned earlier can affect the *apparent* order of layers onscreen, but not the order in which touches will be processed.

That means that if you change the z-order of your layers, you may find you cannot detect touches on the frontmost layer because it is blocked by another layer that has a lower `zPosition` but is situated earlier in the tree. We explore this problem in more detail in Chapter 5.

Automatic Layout

You might have come across the `UIViewAutoresizingMask` constants, used to control how a `UIView` frame is updated when its superview changes size (usually in response to the screen rotating from landscape to portrait or vice versa).

In iOS 6, Apple introduced the *autolayout* mechanism. This works in a different and more sophisticated way than the autoresizing mask, by specifying *constraints* that combine to form a system of linear equations and inequalities that define the position and size of your views.

On Mac OS, `CALayer` has a property called `layoutManager` that enables you to utilize these automatic layout mechanisms using the `CALayoutManager` informal protocol and the `CAConstraintLayoutManager` class. For some reason, however, these are not available on iOS.

When using layer-backed views, you can make use of the `UIViewAutoresizingMask` and `NSLayoutConstraint` APIs exposed by the `UIView` class interface. But if you want to control the layout of an arbitrary `CALayer`, you need to do it manually. The simplest way to do that is using the following method of the `CALayerDelegate`:

```
- (void)layoutSublayersOfLayer:(CALayer *)layer;
```

This method is called automatically whenever the layer bounds changes or the `-setNeedsLayout` method is called on the layer. It gives you the opportunity to reposition and resize your sublayers programmatically, but offers no automatic default behaviors for keeping layers aligned after a screen rotation like the `UIView` `autoresizingMask` and `constraints` properties.

This is another good reason to try to construct your interfaces using views whenever possible rather than using hosted layers.

Summary

This chapter covered the geometry of `CALayer`, including its `frame`, `position`, and `bounds`, and we touched on the concept of layers existing in three-dimensional space instead of a flat plane. We also discussed how touch handing can be implemented when working with hosted layers, and the lack of support for autoresizing and autolayout in Core Animation on iOS.

In Chapter 4, “Visual Effects,” we explore some of Core Animation’s layer appearance features.

Visual Effects

*Well, circles and ovals are good, but how about drawing rectangles with rounded corners?
Can we do that now, too?*

Steve Jobs

We looked at the layer frame in Chapter 3, “Layer Geometry,” and the layer backing image in Chapter 2, “The Backing Image.” But layers are more than mere rectangular containers for colors or images; they also have a number of built-in features that make it possible to create impressive and elegant interface elements programmatically. In this chapter, we explore the various visual effects that can be achieved using `CALayer` properties.

Rounded Corners

One of the signature features of the iOS aesthetic is the use of *rounded rectangles* (rectangles with rounded corners). These appear everywhere in iOS, from homescreen icons, to modal alerts, to text fields. Given their prevalence, you might guess that there would be an easy way to create them without resorting to Photoshop. You’d be right.

`CALayer` has a `cornerRadius` property that controls the curvature of the layer’s corners. It is a floating point value that defaults to zero (sharp corners), but can be set to any value you like (specified in points). By default, this curvature affects only the background color of the layer and not the backing image or sublayers. However, when the `masksToBounds` property is set to YES (see Chapter 2), everything inside the layer is clipped to this curve.

We can demonstrate this effect with a simple project. Let’s arrange a couple of views in Interface Builder that have subviews that extend outside of their bounds (see Figure 4.1). You can’t really see from the figure that the inner views extend beyond their containing

views because Interface Builder always clips views in the editing interface. You'll just have to trust that they do.

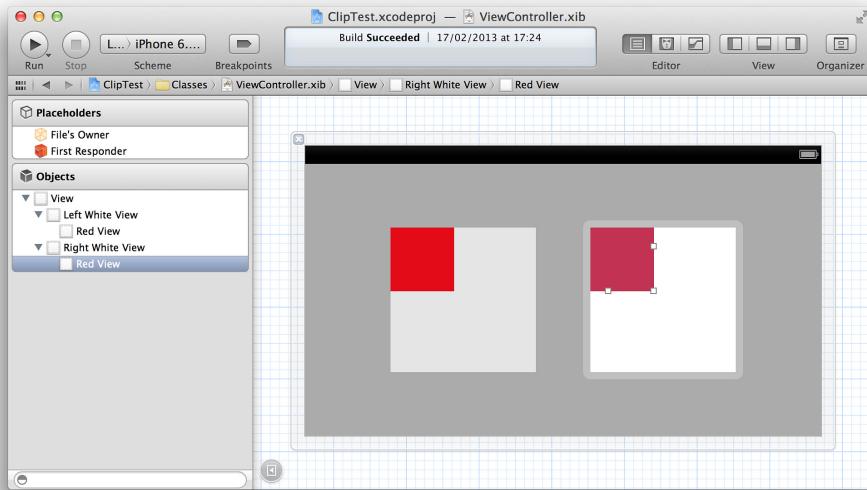


Figure 4.1 Two large white views, each containing small red views

Using code, we'll apply a 20-point radius to the corners and enable clipping on only the second view (see Listing 4.1). Technically, these properties can both be applied directly in Interface Builder by using User Defined Runtime Attributes and the Clip Subviews checkbox in the Inspector panel, respectively, but in the example this is implemented in code for clarity. Figure 4.2 shows the result.

Listing 4.1 Applying cornerRadius and masksToBounds

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView1;
@property (nonatomic, weak) IBOutlet UIView *layerView2;

@end

@implementation ViewController

- (void)viewDidLoad
{
```

```
[super viewDidLoad];

//set the corner radius on our layers
self.layerView1.layer.cornerRadius = 20.0f;
self.layerView2.layer.cornerRadius = 20.0f;

//enable clipping on the second layer
self.layerView2.layer.masksToBounds = YES;
}

@end
```

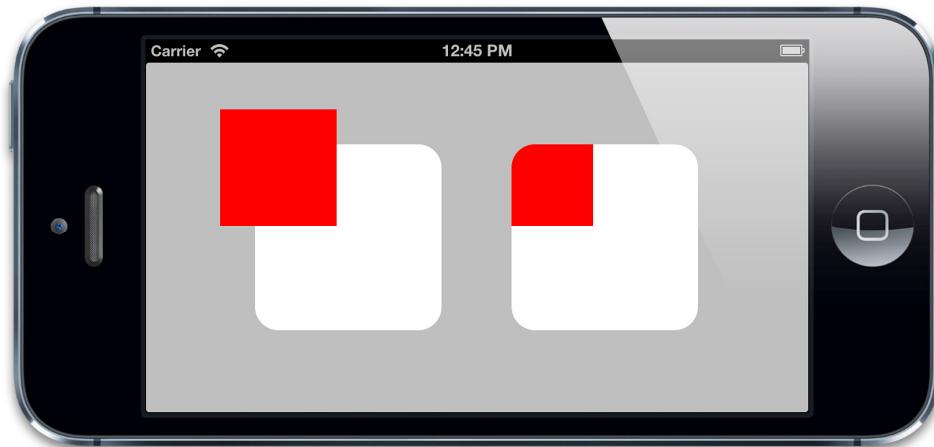


Figure 4.2 The red subview on the right is clipped to the `cornerRadius` of its superview.

As you can see, the red view on the right is clipped to the curve of its superview.

It is not possible to control the curvature of each layer corner independently, so if you want to create a layer or view that has some sharp and some rounded corners, you'll need to find a different approach, such as using a *layer mask* (as covered later in the chapter) or `CAShapeLayer` (see Chapter 6, “Specialized Layers”).

Layer Borders

Another useful pair of `CALayer` properties are `borderWidth` and `borderColor`. Together these define a line that is drawn around the edge of the layer. This line (known as a *stroke*) follows the bounds of the layer, including the corner curvature.

The `borderWidth` is a floating-point number that defines the stroke thickness in points. This defaults to zero (no border). The `borderColor` defines the color of the stroke and defaults to black.

The type of `borderColor` is `CGColorRef`, not `UIColor`, so it's not a Cocoa object per-se. However, you should be aware that the layer retains the `borderColor`, even though there's no indication of this from the property declaration. `CGColorRef` behaves like an `NSObject` in terms of retain/release, but the Objective-C syntax does not provide a way to indicate this, so even strongly retained `CGColorRef` properties must be declared using `assign`.

The border is drawn *inside* the layer bounds, and in front of any other layer contents, including sublayers. If we modify the example to include a layer border (see Listing 4.2), you can see how this works (see Figure 4.3).

Listing 4.2 Applying a Border

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set the corner radius on our layers
    self.layerView1.layer.cornerRadius = 20.0f;
    self.layerView2.layer.cornerRadius = 20.0f;

    //add a border to our layers
    self.layerView1.layer.borderWidth = 5.0f;
    self.layerView2.layer.borderWidth = 5.0f;

    //enable clipping on the second layer
    self.layerView2.layer.masksToBounds = YES;
}

@end
```

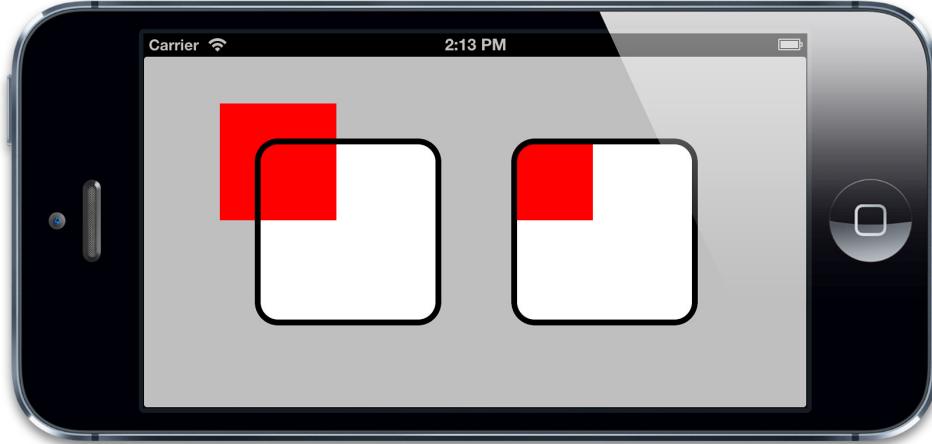


Figure 4.3 Adding a border around the layer

Note that the layer border does not take the shape of the layer backing image or sublayers into account. If the layer's sublayers overflow its bounds, or if the backing image has an alpha mask containing transparent areas, the border will still always follow a (possibly rounded) rectangle around the layer (see Figure 4.4).



Figure 4.4 The border follows the bounds of the layer, not the shape of its contents.

Drop Shadows

Another common feature in iOS is the *drop shadow*. Drop shadows are cast behind a view to imply depth. They are used to indicate layering and priority (such as when a modal alert is presented in front of another view), but they are also sometimes used for purely cosmetic purposes (to give controls a more solid appearance).

A drop shadow can be shown behind any layer by setting the `shadowOpacity` property to a value greater than zero (the default). The `shadowOpacity` is a floating-point value and should be set between 0.0 (invisible) and 1.0 (fully opaque). Setting a value of 1.0 will show a black shadow with a slight blur and a position slightly above the layer. To tweak the appearance of the shadow, you can use a trio of additional `CALayer` properties: `shadowColor`, `shadowOffset`, and `shadowRadius`.

The `shadowColor` property, as its name implies, controls the shadow color and is a `CGColorRef`, just like the `borderColor` and `backgroundColor` properties. The default shadow color is black, which is probably what you want most of the time anyway (colored shadows occur rarely in real life, and can look a bit strange).

The `shadowOffset` property controls the direction and distance to which the shadow extends. The `shadowOffset` is a `CGSize` value, with the width controlling the shadow's horizontal offset and the height controlling its vertical offset. The default `shadowOffset` is `{0, -3}`, which results in a shadow positioned 3 points above the layer along the Y axis.

Why does the default shadow point upward? Although Core Animation was adapted from Layer Kit (the private animation framework created for iOS), its first appearance as a public framework was on Mac OS, which uses an inverted coordinate system with respect to iOS (the Y axis points upward). On a Mac, the same default `shadowOffset` value results in a *downward*-pointing shadow, so the default direction makes more sense in that context (see Figure 4.5).



Figure 4.5 The default `shadowOffset` appearance on iOS (left) and Mac OS (right)

The Apple convention is that user interface shadows point vertically downward, so on iOS it's probably a good idea to use zero for the width and a positive value for the height in most cases.

The `shadowRadius` property controls the *blurriness* of the shadow. A value of zero creates a hard-edged shadow that exactly matches the shape of the view. A larger value creates a soft-edged shadow that looks more natural. Apple's own app designs tend to use soft shadows, so it's probably a good idea to stick with a nonzero value for this.

Generally, you should use a larger `shadowRadius` for something like a modal overlay than you would to make a control stand out from its background; the blurrier the shadow, the greater the illusion of depth (see Figure 4.6).

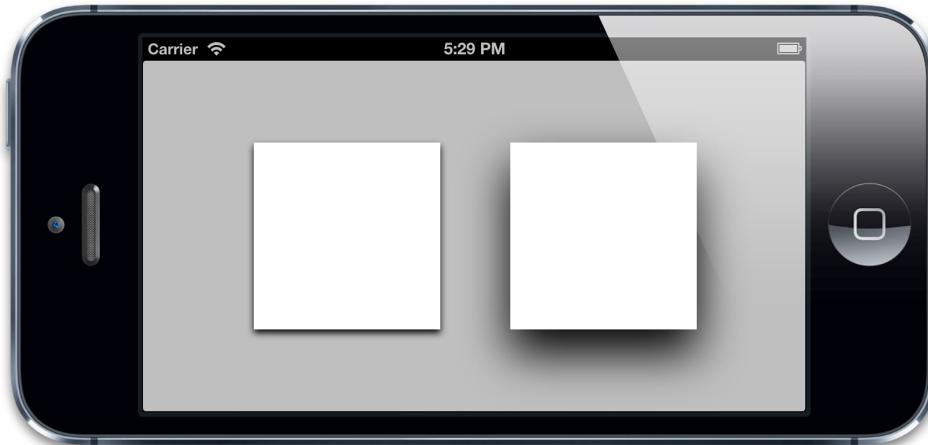


Figure 4.6 A larger shadow offset and radius increases the illusion of depth.

Shadow Clipping

Unlike the layer border, the layer's shadow derives from the *exact* shape of its contents, not just the bounds and `cornerRadius`. To calculate the shape of the shadow, Core Animation looks at the backing image (as well as the sublayers, if there are any) and uses these to create a shadow that perfectly matches the shape of the layer (see Figure 4.7).



Figure 4.7 The layer shadow follows the exact outline of the layer backing image.

Layer shadows have an annoying limitation when combined with clipping: Because the shadow is usually drawn outside the layer bounds, if you enable the `masksToBounds` property, the shadow is clipped along with any other content that protrudes outside of the layer. If we add a layer shadow to our border example project, you can see the problem (see Figure 4.8).

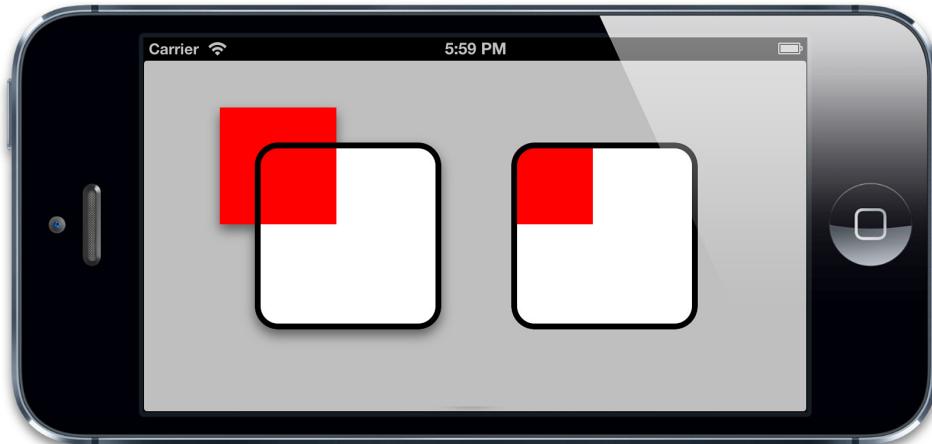


Figure 4.8 The `masksToBounds` property clips both shadow and content.

This behavior is understandable from a technical perspective, but it is unlikely to be the effect that you wanted. If you want to clip the contents *and* cast a shadow, you need to use two layers: an empty outer layer that just draws the shadow, and an inner one that has `masksToBounds` enabled for clipping content.

If we update our project to use an additional view wrapped around the clipping view on the right, we can solve the problem (see Figure 4.9).

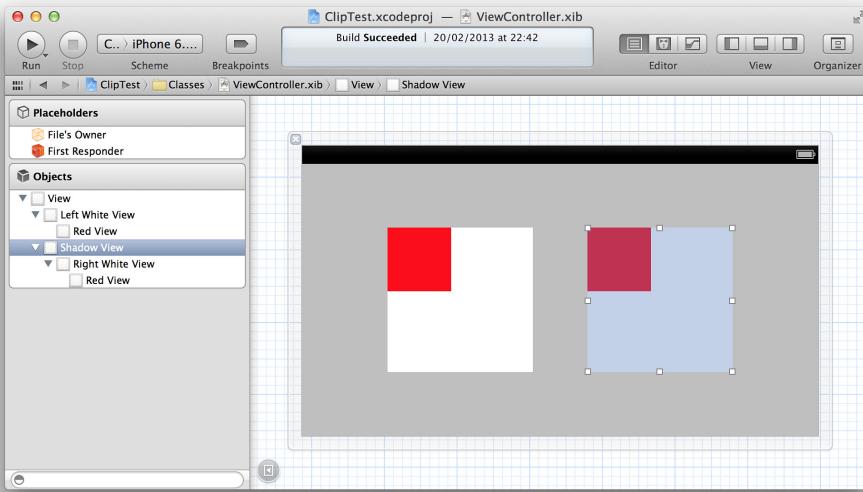


Figure 4.9 An additional shadow-casting view around the clipping view on the right

We attach the shadow only to the outermost view, and enable clipping only on the inner view. Listing 4.3 shows the updated code, and Figure 4.10 shows the result.

Listing 4.3 Using an Additional View to Solve Shadow Clipping Problems

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *layerView1;  
@property (nonatomic, weak) IBOutlet UIView *layerView2;  
@property (nonatomic, weak) IBOutlet UIView *shadowView;  
  
@end  
  
@implementation ViewController
```

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //set the corner radius on our layers
    self.layerView1.layer.cornerRadius = 20.0f;
    self.layerView2.layer.cornerRadius = 20.0f;

    //add a border to our layers
    self.layerView1.layer.borderWidth = 5.0f;
    self.layerView2.layer.borderWidth = 5.0f;

    //add a shadow to layerView1
    self.layerView1.layer.shadowOpacity = 0.5f;
    self.layerView1.layer.shadowOffset = CGSizeMake(0.0f, 5.0f);
    self.layerView1.layer.shadowRadius = 5.0f;

    //add same shadow to shadowView (not layerView2)
    self.shadowView.layer.shadowOpacity = 0.5f;
    self.shadowView.layer.shadowOffset = CGSizeMake(0.0f, 5.0f);
    self.shadowView.layer.shadowRadius = 5.0f;

    //enable clipping on the second layer
    self.layerView2.layer.masksToBounds = YES;
}

@end
```

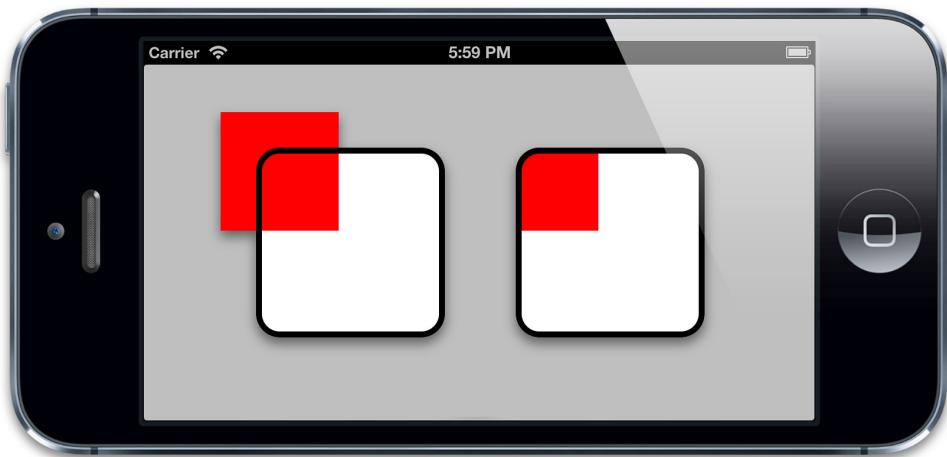


Figure 4.10 The view on the right now has a shadow, despite clipping.

The `shadowPath` Property

We've established that layer shadows are not always square, but instead derive from the shape of the contents. This looks great, but it's also very expensive to calculate in real time, especially if the layer contains multiple sublayers, each with alpha-masked backing images.

If you know in advance what the shape of your shadow needs to be, you can improve performance considerably by specifying a `shadowPath`. The `shadowPath` is a `CGPathRef` (a pointer to a `CGPath` object). `CGPath` is a Core Graphics object used to specify an arbitrary vector shape. We can use this to define the shape of our shadow independently of the layer's shape.

Figure 4.11 shows two different shadow shapes applied to the same layer image. In this case, the shapes we've used are simple, but they can be absolutely any shape you want. See Listing 4.4 for the code.

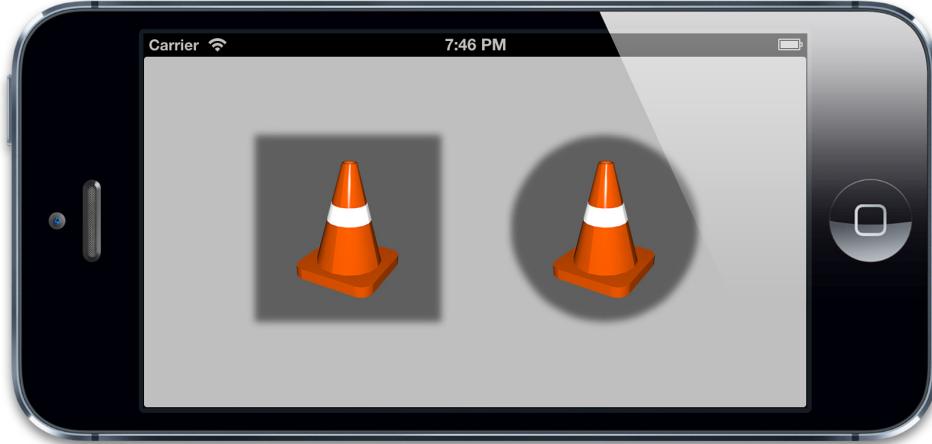


Figure 4.11 Using `shadowPath` to cast shadows with arbitrary shapes

Listing 4.4 Creating Simple Shadow Paths

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *layerView1;  
@property (nonatomic, weak) IBOutlet UIView *layerView2;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //enable layer shadows  
    self.layerView1.layer.shadowOpacity = 0.5f;  
    self.layerView2.layer.shadowOpacity = 0.5f;  
  
    //create a square shadow  
    CGMutablePathRef squarePath = CGPathCreateMutable();  
    CGPathAddRect(squarePath, NULL, self.layerView1.bounds);  
    self.layerView1.layer.shadowPath = squarePath;  
    CGPathRelease(squarePath);  
  
    //create a circular shadow
```

```
CGMutablePathRef circlePath = CGPathCreateMutable();
CGPathAddEllipseInRect(circlePath, NULL, self.layerView2.bounds);
self.layerView2.layer.shadowPath = circlePath;
CGPathRelease(circlePath);
}

@end
```

For something like a rectangle or circle, creating a `CGPath` manually is fairly straightforward. For a more complex shape like a rounded rectangle, you'll probably find it easier to use the `UIBezierPath` class, which is an Objective-C wrapper around `CGPath` provided by UIKit.

Layer Masking

Using the `masksToBounds` property, we know that it is possible to clip a layer's contents to its bounds, and using the `cornerRadius` property, we can even give it rounded corners. But sometimes you will want to represent content that is not rectangular or even rounded-rectangular in shape. For example, you might want to create a star-shaped photo frame around an image, or you might want the edges of some scrolling text to fade gracefully into the background instead of clipping to a sharp edge.

Using a 32-bit PNG image with an alpha component, you can specify a backing image that includes an arbitrary alpha mask, which is usually the simplest way to create a non-rectangular view. But that approach doesn't allow you to clip images dynamically using programmatically generated masks or to have sublayers or subviews that also clip to the same arbitrary shape.

`CALayer` has a property called `mask` that can help with this problem. The `mask` property is itself a `CALayer` and has all the same drawing and layout properties of any other layer. It is used in a similar way to a sublayer in that it is positioned relative to its parent (the layer that owns it), but it does not *appear* as a normal sublayer. Instead of being drawn inside the parent, the `mask` layer defines the part of the parent layer that is visible.

The *color* of the `mask` layer is irrelevant; all that matters is its *silhouette*. The `mask` acts like a cookie cutter; the solid part of the `mask` layer will be "cut out" of its parent layer and kept; anything else is discarded (see Figure 4.12).

If the `mask` layer is smaller than the parent layer, only the parts of the parent (or its sub-layers) that intersect the `mask` will be visible. If you are using a `mask` layer, anything outside of that layer is implicitly hidden.

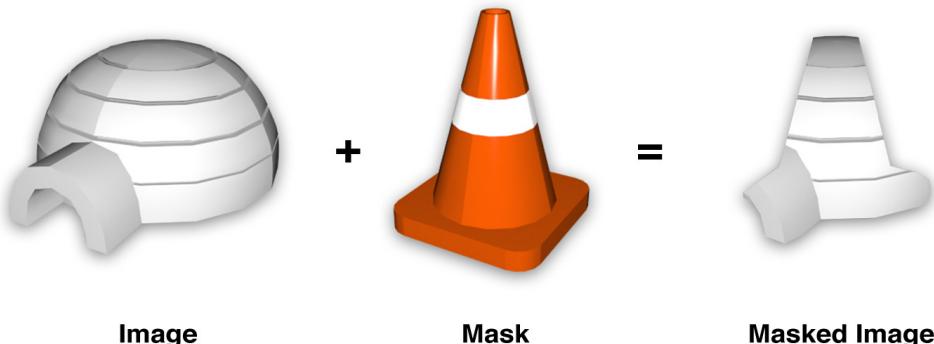


Figure 4.12 Combining separate image and mask layers to create a masked image

To demonstrate this, let's create a simple project that masks one image with another using the layer mask property. To simplify things, we'll create our image layer in Interface Builder using a UIImageView, so that only the mask layer needs to be created and applied programmatically. Listing 4.5 shows the code to do this, and Figure 4.13 shows the result.

Listing 4.5 Applying a Layer Mask

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIImageView *imageView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //create mask layer  
    CALayer *maskLayer = [CALayer layer];  
    maskLayer.frame = self.layerView.bounds;  
    UIImage *maskImage = [UIImage imageNamed:@"Cone.png"];  
    maskLayer.contents = (__bridge id)maskImage.CGImage;  
  
    //apply mask to image layer
```

```
self.imageView.layer.mask = maskLayer;  
}  
  
@end
```



Figure 4.13 Our `UIImageView`, after the `mask` layer has been applied

The *really* cool feature of `CALayer` masking is that you are not limited to using static images for your masks. Anything that can be composed out of layers can be used as the `mask` property, which means that your masks can be created dynamically using code, and even animated in real time.

Scaling Filters

The final topic we cover in this chapter is the effect of the `minificationFilter` and `magnificationFilter` properties. Generally on iOS, when you display images, you should try to display them at the correct size (that is, with a 1:1 correlation between the pixels in the image and the pixels onscreen). The reasons for this are as follows:

- It provides the best possible quality, because the pixels aren't stretched or resampled.
- It makes the best use of RAM, because you aren't storing more pixels than needed.
- It yields the best performance, because the GPU doesn't have to work as hard.

Sometimes, however, it's necessary to display an image at a larger-than-actual or smaller-than-actual size. Examples might include a thumbnail image of a person or avatar, or a very large image that the user can pan and zoom. In these cases, it might not be practical to store separate copies of the image for every size it might need to be displayed at.

When images are displayed at different sizes, an algorithm (known as a *scaling filter*) is applied to the pixels of the original image to generate the new pixels that will be displayed onscreen.

There is no universally ideal algorithm for resizing an image. The approach depends on the nature of the content being scaled, and whether you are scaling up or down. CALayer offers a choice of three scaling filters to use when resizing images. These are represented by the following string constants:

```
kCAFilerLinear  
kCAFilerNearest  
kCAFilerTrilinear
```

The default filter for both *minification* (shrinking an image) and *magnification* (expanding an image) is kCAFilerLinear. This filter uses the *bilinear* filtering algorithm, which yields good results under the majority of circumstances. Bilinear filtering works by sampling multiple pixels to create the final value. This results in nice, smooth scaling, but can make the image appear blurry if it scaled up by a large factor (see Figure 4.14).

The kCAFilerTrilinear option is very similar to kCAFilerLinear. There is no visible difference between them in most cases, but *trilinear* filtering improves on the performance of bilinear filtering by storing the image at multiple sizes (known as *mipmapping*) and then sampling in three dimensions, combining pixels from the larger and smaller stored image representations to create the final result.

The advantage of this approach is that the algorithm can work from a pair of images that are already quite close to the final size. This means that it does not need to sample as many pixels simultaneously, which improves performance and avoids the sampling glitches that can occur at very small scale factors due to rounding errors.

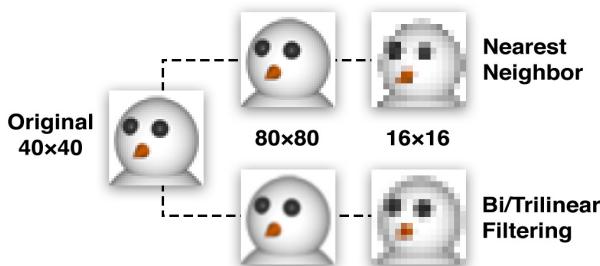


Figure 4.14 For larger images, bilinear or trilinear filtering is usually better.

The `kCAFILTERNearest` option is the crudest approach. As the name suggests, this algorithm (known as *nearest-neighbor* filtering) just samples the nearest single pixel and performs no color blending at all. This is very fast, and doesn't blur the image, but the quality is noticeably worse for shrunken images, and magnified images become blocky and pixelated.

In Figure 4.14, note how the bilinear image looks less distorted than the nearest-neighbor version when shrunk to a small size, but when enlarged it looks blurrier. Contrast this with Figure 4.15, where we've started with a very small image. In this instance, nearest-neighbor does a better job of preserving the original pixels, whereas linear filtering turns them into a blurry mess regardless of whether the image is minified or magnified.

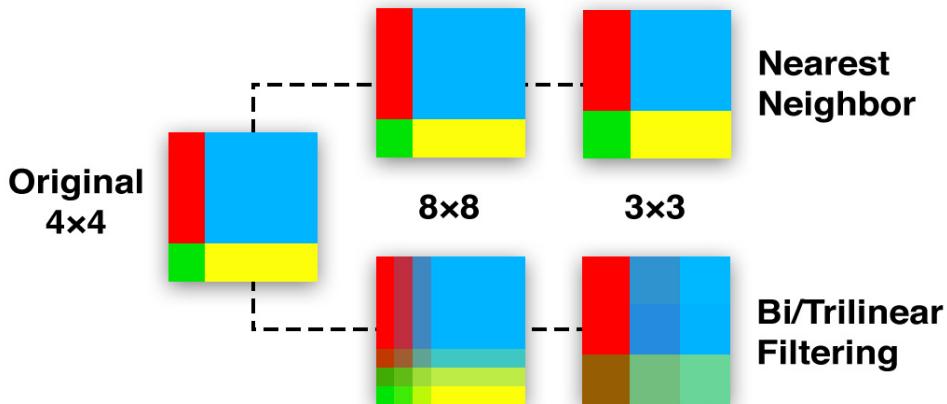


Figure 4.15 For small images without diagonals, nearest-neighbor filtering is better.

Generally speaking, for very small images or larger images with sharp contrast and few diagonal lines (for example, computer-generated images), nearest-neighbor scaling will preserve contrast and may yield better results. But for most images, especially photographs or images with diagonals or curves, nearest-neighbor will look appreciably worse than linear filtering. To put it another way, linear filtering preserves the *shape*, and nearest-neighbor filtering preserves the *pixels*.

Let's try a real-world example. We'll modify the clock project from Chapter 3 to display an LCD-style digital readout instead of an analog clock face. The digits will be created using a simple *pixel font* (a font where the characters are constructed from individual pixels rather than vector shapes), stored as a single image and displayed using the sprite sheet technique introduced in Chapter 2 (see Figure 4.16).



Figure 4.16 A simple “pixel font” sprite sheet for displaying LCD-style digits

We’ll arrange six views in Interface Builder, two each for the hours, minutes, and seconds digits. Figure 4.17 shows how the views are arranged in Interface Builder. That many views starts to get a bit unwieldy when using individual outlets, so we’ll connect them to the controller using an `IBOutletCollection`, which allows us to access the views as an array. Listing 4.6 shows the code for the clock.

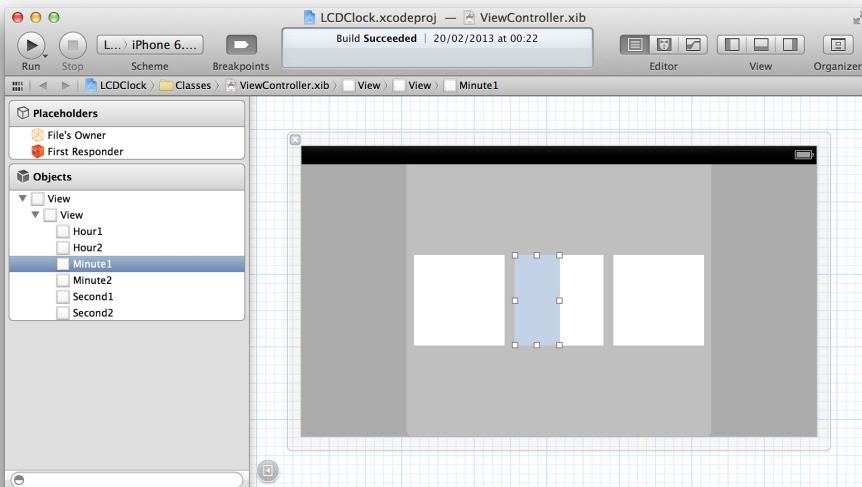


Figure 4.17 The clock digit views arranged into hours, minutes, and seconds

Listing 4.6 Displaying an LCD-Style Clock

```
@interface ViewController : UIViewController  
{  
    IBOutletCollection(UIView) NSArray *digitViews;  
    NSTimer *timer;  
}
```

```

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //get spritesheet image
    UIImage *digits = [UIImage imageNamed:@"Digits.png"];

    //set up digit views
    for (UIView *view in self.digitViews)
    {
        //set contents
        view.layer.contents = (__bridge id)digits.CGImage;
        view.layer.contentsRect = CGRectMake(0, 0, 0.1, 1.0);
        view.layer.contentsGravity = kCAGravityResizeAspect;
    }

    //start timer
    self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                target:self
                                              selector:@selector(tick)
                                              userInfo:nil
                                             repeats:YES];
}

//set initial clock time
[self tick];
}

- (void)setDigit:(NSInteger)digit forView:(UIView *)view
{
    //adjust contentsRect to select correct digit
    view.layer.contentsRect = CGRectMake(digit * 0.1, 0, 0.1, 1.0);
}

- (void)tick
{
    //convert time to hours, minutes and seconds
    NSCalendar *calendar = [[NSCalendar alloc] initWithCalendarIdentifier:
                           NSGregorianCalendar];

    NSUInteger units = NSHourCalendarUnit |
                      NSMinuteCalendarUnit |
                      NSSecondCalendarUnit;
}

```

```

NSDateComponents *components = [calendar components:units
                                             fromDate:[NSDate date]];

//set hours
[self setDigit:components.hour / 10 forView:self.digitViews[0]];
[self setDigit:components.hour % 10 forView:self.digitViews[1]];

//set minutes
[self setDigit:components.minute / 10 forView:self.digitViews[2]];
[self setDigit:components.minute % 10 forView:self.digitViews[3]];

//set seconds
[self setDigit:components.second / 10 forView:self.digitViews[4]];
[self setDigit:components.second % 10 forView:self.digitViews[5]];
}

@end

```

As you can see from Figure 4.18, it works, but the digits look blurry. It seems that the default `kCAFFilterLinear` option has failed us.



Figure 4.18 A blurry clock display, caused by the default `kCAFFilterLinear` scaling

To get the crisp digits shown in Figure 4.19, we just need to add the following line to the `for...in` loop in our program:

```
view.layer.magnificationFilter = kCAFFilterNearest;
```



Figure 4.19 A sharp clock display, achieved by using nearest-neighbor scaling

Group Opacity

`UIView` has a handy `alpha` property that can be used to vary its transparency. `CALayer` has an equivalent property called `opacity`. Both properties work hierarchically, so if you set the `opacity` of a layer it will automatically affect all of its sublayers, as well.

A common trick in iOS is to set a control's `alpha` to 0.5 (50%) to make it appear disabled. This works great for individual views, but when a control has subviews it can look a bit strange. Figure 4.20 shows a custom `UIButton` containing a nested `UILabel`; on the left is an opaque button, and on the right is the same button shown with 50% alpha. Notice how we can see the outline of the internal label against the button background.

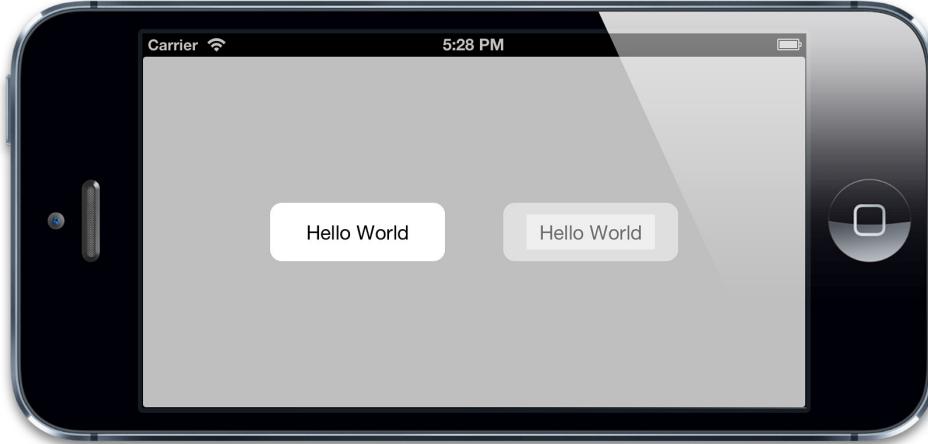


Figure 4.20 In the faded button on the right, the internal label's border is clearly visible.

This effect is due the way that alpha blending works. When you display a layer with 50% opacity, each pixel of the layer displays 50% of its own color and 50% of the layer behind it. That results in the appearance of translucency. But if the layer contains sublayers that are also displayed at 50% transparency, then when you look through the sublayer, you are seeing 50% of the sublayer's color, 25% of the containing layer's color, and only 25% of the background color.

In our example, the button and label both have white backgrounds. Even though they are both only 50% opaque, their combined opacity is 75%, and so the area where the label overlaps the button look less transparent than the surrounding part. This serves to highlight all of the sublayers that make up a control and produces a nasty visual effect.

Ideally, when you set the `opacity` of a layer, you want its entire subtree to fade as if it were a single layer, without revealing its internal structure. You can achieve this by setting `UIViewSetGroupOpacity` to YES in your `Info.plist` file, but this affects the way that blending is handled across the whole application, and introduces a small app-wide performance penalty. If the `UIViewSetGroupOpacity` key is omitted, its value defaults to NO on iOS 6 and earlier (though the default may change in a future iOS release).

Alternatively, you can implement group opacity for a specific layer subtree by using a `CALayer` property called `shouldRasterize` (see Listing 4.7). When set to YES, the `shouldRasterize` property causes the layer and its sublayers to be collapsed into a single flat image *before the opacity is applied*, thereby eliminating the blending glitch (see Figure 4.21).

In addition to enabling the `shouldRasterize` property, we've modified the layer's `rasterizationScale` property. By default, all layers are rasterized at a scale of 1.0, so if you use the `shouldRasterize` property, you should always ensure that you set the

`rasterizationScale` to match the screen to avoid views that look pixelated on a Retina display.

As with `UIViewGroupOpacity`, use of the `shouldRasterize` property has performance implications (which are explained in Chapter 12, “Tuning for Speed,” and Chapter 15, “Layer Performance”), but the performance impact is localized.

Listing 4.7 Using `shouldRasterize` to Fix the Grouped Blending Problem

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (UIButton *)customButton
{
    //create button
    CGRect frame = CGRectMake(0, 0, 150, 50);
    UIButton *button = [[UIButton alloc] initWithFrame:frame];
    button.backgroundColor = [UIColor whiteColor];
    button.layer.cornerRadius = 10;

    //add label
    frame = CGRectMake(20, 10, 110, 30);
    UILabel *label = [[UILabel alloc] initWithFrame:frame];
    label.text = @"Hello World";
    label.textAlignment = NSTextAlignmentCenter;
    [button addSubview:label];

    return button;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create opaque button
    UIButton *button1 = [self customButton];
    button1.center = CGPointMake(50, 150);
    [self.containerView addSubview:button1];

    //create translucent button
    UIButton *button2 = [self customButton];
    button2.backgroundColor = [UIColor colorWithRed:0.5 green:0.5 blue:0.5 alpha:0.5];
    button2.layer.shouldRasterize = YES;
    button2.layer.rasterizationScale = 2;
    button2.layer.zPosition = 1;
    button2.center = CGPointMake(150, 150);
    [self.containerView addSubview:button2];
}
```

```
button2.center = CGPointMake(250, 150);
button2.alpha = 0.5;
[self.containerView addSubview:button2];

//enable rasterization for the translucent button
button2.layer.shouldRasterize = YES;
button2.layer.rasterizationScale = [UIScreen mainScreen].scale;
}

@end
```

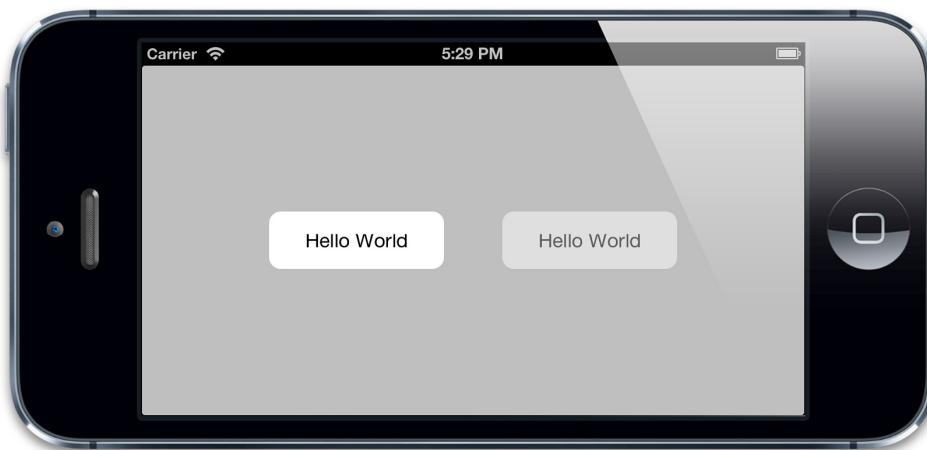


Figure 4.21 The internal structure of the faded button is no longer visible.

Summary

This chapter explored some of the visual effects that you can apply programmatically to layers, such as rounded corners, drop shadows, and masks. We also looked at scaling filters and group opacity.

In Chapter 5, “Transforms,” we investigate layer transforms and transport our layers into the third dimension.

5

Transforms

Unfortunately, no one can be told what the Matrix is. You have to see it for yourself.

Morpheus, *The Matrix*

In Chapter 4, “Visual Effects,” we looked at some techniques to enhance the appearance of layers and their contents. In this chapter, we investigate `CGAffineTransform`, which can be used to rotate, reposition, and distort our layers, and at `CATransform3D`, which can change boring flat rectangles (albeit rounded rectangles with drop-shadows) into three-dimensional surfaces.

Affine Transforms

In Chapter 3, “Layer Geometry,” we made use of the `UIView transform` property to rotate the hands on a clock, but we didn’t really explain what was going on behind the scenes. The `UIView transform` property is of type `CGAffineTransform`, and is used to represent a two-dimensional rotation, scale, or translation. `CGAffineTransform` is a 2-column-by-3-row matrix that can be multiplied by a 2D row-vector (in this case a `CGPoint`) to transform its value (see the boldface values in Figure 5.1).

This multiplication is performed by taking the values in each column of the `CGPoint` vector, multiplying them by the values in each row of the `CGAffineTransform` matrix, then adding the results together to create a new `CGPoint`. This explains the additional values shown in gray in the figure; for matrix multiplication to work, the matrix on the left must have the same number of columns as the matrix on the right has rows, so we have to pad out the matrices with so-called *identity* values—numbers that will make the sums work, but without changing the result. We don’t actually need to store those additional values because they never change, but they are required for the calculation.

For this reason, you will often see 2D transforms represented as a 3×3 matrix (instead of 2×3). You will also often see the matrix shown in a 3-column-by-2-row format instead, with the vector values stacked vertically. This is known as *column-major* format. The way we've presented it in Figure 5.1 is *row-major* format. It doesn't actually matter which representation you use as long as you are consistent.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

CGPoint	CGAffineTransform	Transformed CGPoint
----------------	--------------------------	----------------------------

Figure 5.1 `CGAffineTransform` and `CGPoint` represented as matrices

When the transform matrix is applied to a layer, each corner point of the layer rectangle is individually transformed, resulting in a new quadrilateral shape. The “affine” in `CGAffineTransform` just means that whatever values are used for the matrix, lines in the layer that were parallel before the transform will remain parallel after the transform. A `CGAffineTransform` can be used to define *any* transform that meets that criterion. Figure 5.2 shows some examples of affine and nonaffine transforms:

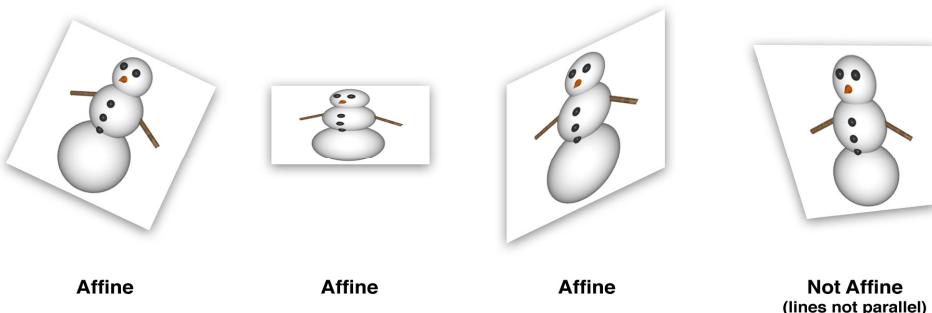


Figure 5.2 Affine and nonaffine transforms

Creating a CGAffineTransform

A full explanation of matrix mathematics is beyond the scope of this book, and if you are not already familiar with matrices, the idea of a transform matrix can seem quite daunting. Fortunately, Core Graphics provides a number of built-in functions for building up arbitrary transforms out of simple ones without requiring the developer to do any math. The following functions each create a new `CGAffineTransform` matrix from scratch:

```
CGAffineTransformMakeRotation(CGFloat angle)
CGAffineTransformMakeScale(CGFloat sx, CGFloat sy)
CGAffineTransformMakeTranslation(CGFloat tx, CGFloat ty)
```

The rotation and scale transforms are fairly self-explanatory—they rotate and scale a vector respectively. A *translation* transform just adds the specified x and y values to the vector—so if the vector represents a point, it moves the point.

Let's demonstrate the effect of these functions with a simple project. We'll start with an ordinary view and apply a 45-degree rotation transform (see Figure 5.3).

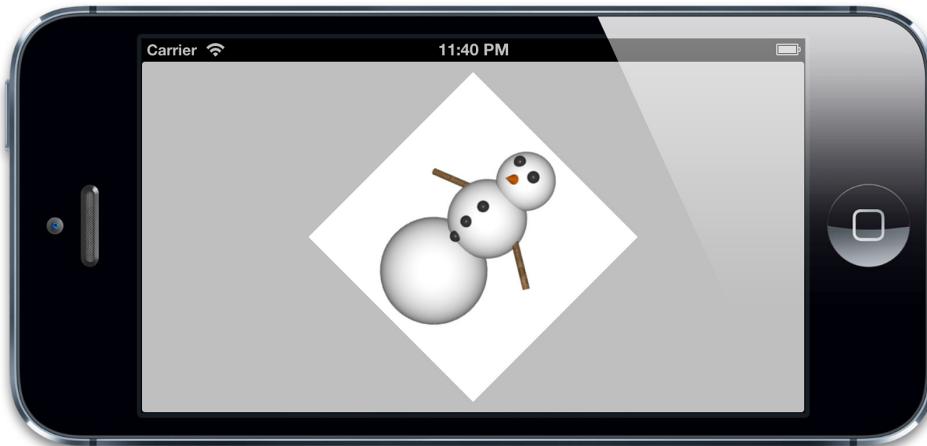


Figure 5.3 A view rotated 45 degrees using an affine transform

A `UIView` can be transformed by setting its `transform` property, but as with all layout properties, the `UIView transform` is really just a wrapper around an equivalent `CALayer` feature.

`CALayer` also has a `transform` property, but its type is `CATransform3D`, not `CGAffineTransform`. We come back to that later in the chapter, but for now it's not what we're looking for. The `CALayer` equivalent to the `UIView transform` property is called `affineTransform`. Listing 5.1 shows the code for rotating a layer by 45 degrees in the clockwise direction using the `affineTransform` property.

Listing 5.1 Rotating a Layer by 45 Degrees Using affineTransform

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //rotate the layer 45 degrees
    CGAffineTransform transform = CGAffineTransformMakeRotation(M_PI_4);
    self.layerView.layer.affineTransform = transform;
}

@end
```

Note that the value we've used for the angle is a constant called `M_PI_4`, not the number 45 as you might have expected. The transform functions on iOS use radians rather than degrees for all angular units. Radians are usually specified using multiples of the mathematical constant π (pi). π radians equates to 180 degrees, so π divided by 4 is equivalent to 45 degrees.

The C math library (which is automatically included in every iOS project) conveniently provides constants for common multiples of π , and `M_PI_4` is the constant representing π divided by 4. If you struggle to think in terms of radians, you can use these macros to convert to and from degrees:

```
#define RADIANS_TO_DEGREES(x) ((x)/M_PI*180.0)
#define DEGREES_TO_RADIANS(x) ((x)/180.0*M_PI)
```

Combining Transforms

Core Graphics also provides a second set of functions that can be used to apply a further transform on top of an existing one. This is useful if you want to create a single transform matrix that both scales *and* rotates a layer, for example. These functions are as follows:

```
CGAffineTransformRotate(CGAffineTransform t, CGFloat angle)
CGAffineTransformScale(CGAffineTransform t, CGFloat sx, CGFloat sy)
CGAffineTransformTranslate(CGAffineTransform t, CGFloat tx, CGFloat ty)
```

When you are manipulating transforms, it is often useful to be able to create a transform that does nothing at all—the `CGAffineTransform` equivalent of zero or nil. In the world of matrices, such a value is known as the *identity matrix*, and Core Graphics provides a convenient constant for this:

```
CGAffineTransformIdentity
```

Finally, if you ever want to combine two existing transform matrices, you can use the following function, which creates a new `CGAffineTransform` matrix from two existing ones:

```
CGAffineTransformConcat (CGAffineTransform t1, CGAffineTransform t2);
```

Let's use these functions in combination to build up a more complex transform. We'll start by applying a scale factor of 50%, then a 30-degree rotation, and finally a translation of 200 points to the right (see Listing 5.2). Figure 5.4 shows the result of applying these transforms to our layer.

Listing 5.2 Creating a Compound Transform Using Several Functions

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a new transform
    CGAffineTransform transform = CGAffineTransformIdentity;

    //scale by 50%
    transform = CGAffineTransformScale(transform, 0.5, 0.5);

    //rotate by 30 degrees
    transform = CGAffineTransformRotate(transform, M_PI / 180.0 * 30.0);

    //translate by 200 points
    transform = CGAffineTransformTranslate(transform, 200, 0);

    //apply transform to layer
    self.layerView.layer.affineTransform = transform;
}
```

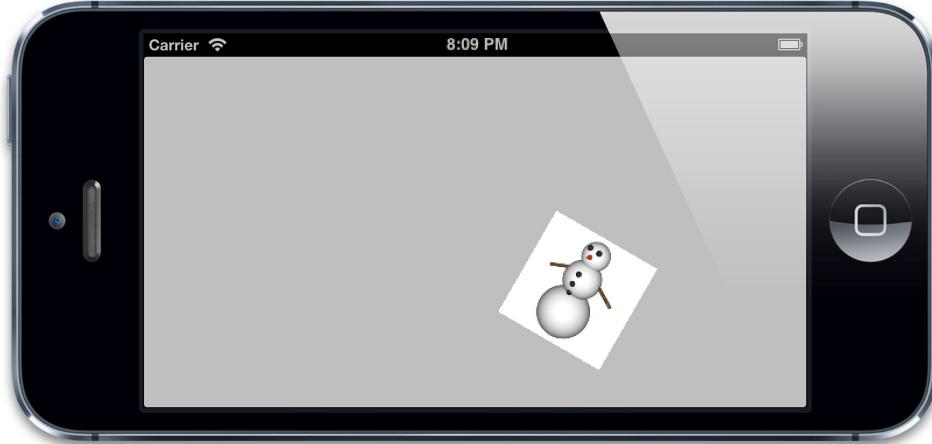


Figure 5.4 The effect of multiple affine transforms applied in sequence

There are a few things worth noting about Figure 5.4: The image has been moved to the right, but not as far as we specified (200 points), and it has also moved down instead of just sideways. The reason for this is that when you apply transforms sequentially in this way, the previous transforms affect the subsequent ones. The 200-point translation to the right has been rotated by 30 degrees and scaled by 50%, so it has actually become a translation diagonally downward by 100 points.

This means that the order in which you apply transforms affects the result; a translation followed by a rotation is not the same as a rotation followed by a translation.

The Shear Transform

Because Core Graphics provides functions to calculate the correct values for the transform matrix for you, it's rare that you need to set the fields of a `CGAffineTransform` directly. One such circumstance is when you want to create a *shear* transform, for which Core Graphics provides no built-in function.

The shear transform is a fourth type of affine transform. It is less commonly used than translation, rotation, and scaling (which is probably why Core Graphics has no built-in function for it), but it can still sometimes be useful. Its effect is probably best illustrated with a picture (see Figure 5.5). For want of a better term, it makes the layer “slanty.” Listing 5.3 shows the code for the shear transform function.

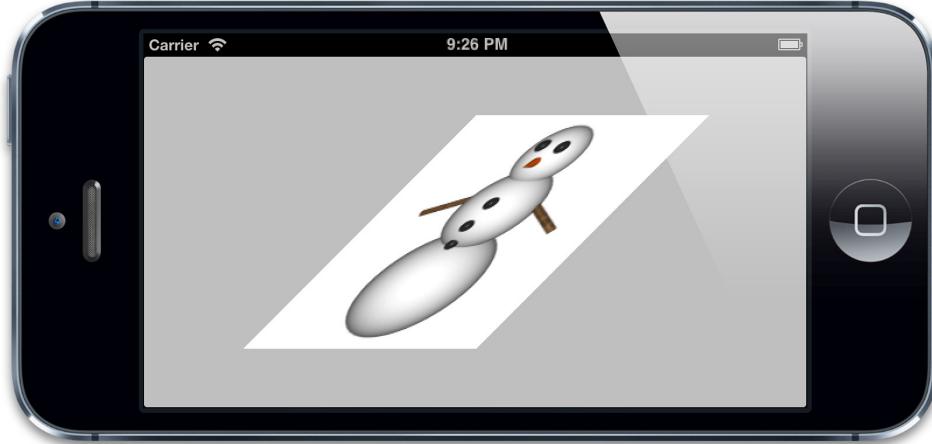


Figure 5.5 A horizontal shear transform

Listing 5.3 Implementing a Shear Transform

```
@implementation ViewController

CGAffineTransform CGAffineTransformMakeShear(CGFloat x, CGFloat y)
{
    CGAffineTransform transform = CGAffineTransformIdentity;
    transform.c = -x;
    transform.b = y;
    return transform;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //shear the layer at a 45-degree angle
    self.layerView.layer.affineTransform = CGAffineTransformMakeShear(1, 0);
}

@end
```

3D Transforms

As the CG prefix indicates, the `CGAffineTransform` type belongs to the Core Graphics framework. Core Graphics is a strictly 2D drawing API, and `CGAffineTransform` is intended only for 2D transforms (that is, ones that apply only within a two-dimensional plane).

In Chapter 3, we looked at the `zPosition` property, which enables us to move layers toward or away from the camera (the user's viewpoint). The `transform` property (which is of type `CATransform3D`) generalizes this idea, allowing us to both move *and* rotate a layer in three dimensions.

Like `CGAffineTransform`, `CATransform3D` is a matrix. But instead of a 2-by-3 matrix, `CATransform3D` is a 4-by-4 matrix that is capable of arbitrarily transforming a point in 3D (see Figure 5.6).

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}$$

`CGPoint + zPosition`

`CATransform3D`

`Transformed Point`

Figure 5.6 The `CATransform3D` matrix transforming a 3D point

Core Animation provides a number of functions that can be used to create and combine `CATransform3D` matrices in exactly the same way as with `CGAffineTransform` matrices. The functions are similar to the Core Graphics equivalents, but the 3D translation and scaling functions provide an additional `z` argument, and the rotation function accepts an `x`, `y`, and `z` argument in addition to the angle, which together form a vector that defines the axis of rotation:

```
CATransform3DMakeRotation(CGFloat angle, CGFloat x, CGFloat y, CGFloat z)
CATransform3DMakeScale(CGFloat sx, CGFloat sy, CGFloat sz)
CATransform3DMakeTranslation(Gloat tx, CGFloat ty, CGFloat tz)
```

You should now be familiar with the X and Y axes, which extend to the right and down respectively (although you may recall from Chapter 3 that this is only the case on iOS; on Mac OS, the Y axis points upward). The Z axis is perpendicular to those axes and extends outward toward the camera (see Figure 5.7).

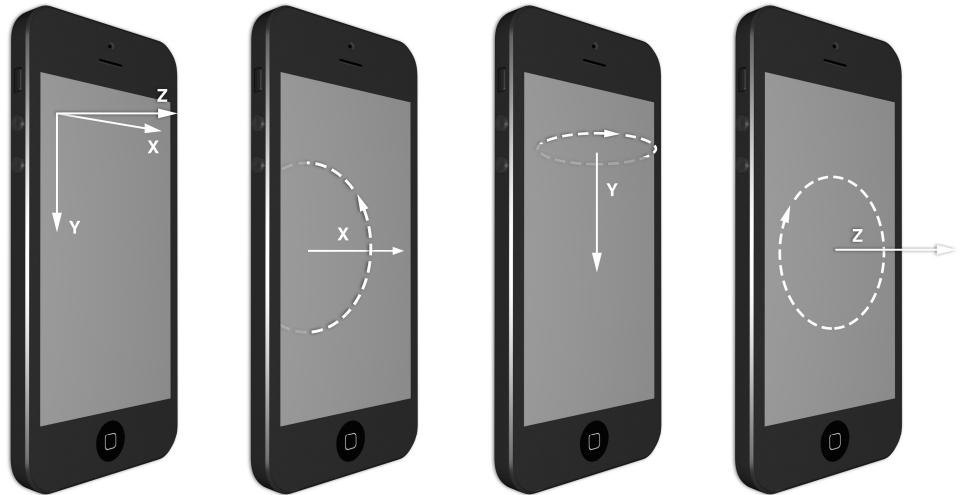


Figure 5.7 The X, Y, and Z axes, and the planes of rotation around them

As you can see from the figure, a rotation around the Z axis is equivalent to the 2D affine rotation we made earlier. A rotation around the X or Y axes, however, rotates a layer *out of* the 2D plane of the screen and tilts it away from the camera.

Let's try an example: The code in Listing 5.4 uses `CATransform3DMakeRotation` to rotate our view's backing layer by 45 degrees around the Y axis. We would expect this to tilt the view to the right, so that we are looking at it from an angle.

The result is shown in Figure 5.8, but it's not quite what we would have expected.

Listing 5.4 Rotating a Layer Around the Y Axis

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //rotate the layer 45 degrees along the Y axis
    CATransform3D transform = CATransform3DMakeRotation(M_PI_4, 0, 1, 0);
    self.layerView.layer.transform = transform;
}

@end
```

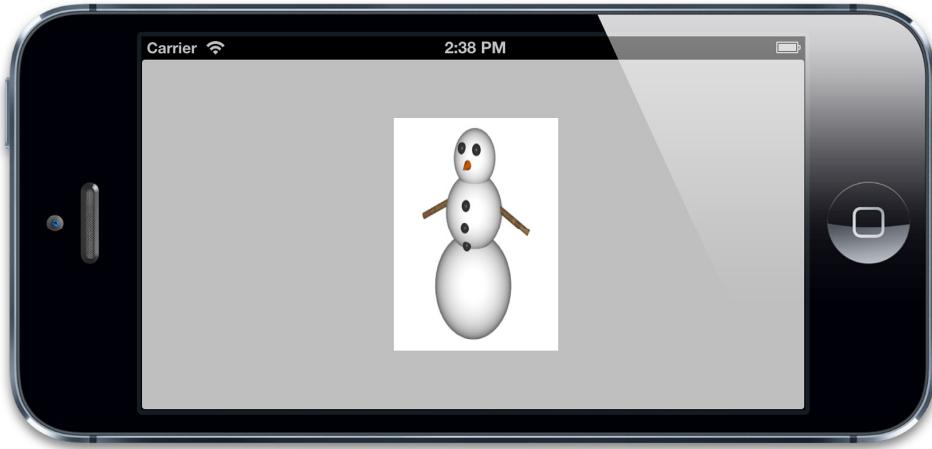


Figure 5.8 A view rotated 45 degrees around the Y axis

It doesn't really look like the view has been rotated at all; it looks more like it's been horizontally squashed using a scale transform. Did we do something wrong?

No, this is actually correct. The view looks narrower because we are looking at it diagonally, so there is less of it facing the camera. The reason it doesn't look right is because there is no *perspective*.

Perspective Projection

In real life, when things are farther away, they seem to get smaller due to perspective. We would expect the side of the view that is farther away from us to appear shorter than the side that is closer, but that isn't happening. What we are looking at currently is an *isometric projection* of our view, which is a method of 3D drawing that preserves parallel lines, much like the affine transforms we were using earlier.

In an isometric projection, objects that are farther away appear at the same scale as objects that are close to us. This kind of projection has its uses (for example, for an architectural drawing, or a top-down, pseudo-3D videogame), but it's not what we want right now.

To fix this, we need to modify our transform matrix to include a *perspective transform* (sometimes called the *z transform*) in addition to the rotation transform we've already applied. Core Animation doesn't give us any functions to set up a perspective transform, so we'll have to modify our matrix values manually. Fortunately, though, this is simple:

The perspective effect of a `CATransform3D` is controlled by a single value in the matrix: element `m34`. The `m34` value (shown in Figure 5.9) is used in the transform calculation to scale the X and Y values in proportion to how far away they are from the camera.

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}$$

CGPoint + zPosition

CATransform3D

Transformed Point

Figure 5.9 The m_{34} value of `CATransform3D`, used for perspective

By default, m_{34} has a value of zero. We can apply perspective to our scene by setting the m_{34} property of our transform to $-1.0 / d$, where d is the distance between the imaginary camera and the screen, measured in points. How do we calculate what this distance should be? We don't have to; we can just *make something up*.

Because the camera doesn't really exist, we are free to decide where it is positioned based on what looks good in our scene. A value between 500 and 1000 usually works fairly well, but you may find that smaller or larger values look better for a given layer arrangement. Decreasing the distance value increases the perspective effect, so a very small value will look extremely distorted, and a very large value will just look like there is no perspective at all (isometric). Listing 5.5 shows the code to apply perspective to our view, and Figure 5.10 shows the result.

Listing 5.5 Applying Perspective to the Transform

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a new transform
    CATransform3D transform = CATransform3DIdentity;

    //apply perspective
    transform.m34 = - 1.0 / 500.0;

    //rotate by 45 degrees along the Y axis
    transform = CATransform3DRotate(transform, M_PI_4, 0, 1, 0);

    //apply to layer
    self.layerView.layer.transform = transform;
```

```
}
```

```
@end
```



Figure 5.10 Our rotated view again, now with a perspective transform applied

The Vanishing Point

When drawn in perspective, objects get smaller as they move away from the camera. As they move *even farther*, they eventually shrink to a point. All distant objects eventually converge on a single *vanishing point*.

In real life, the vanishing point is always in the center of your view (see Figure 5.11), and generally, to create a realistic perspective effect in your app, the vanishing point should be in the center of the screen, or at least the center of the view that contains all of your 3D objects.

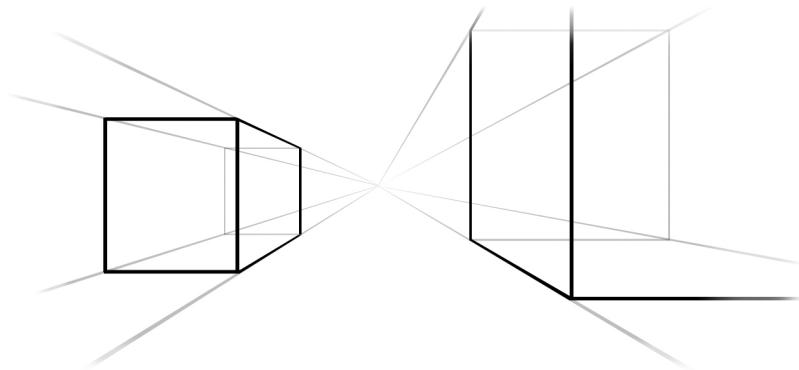


Figure 5.11 The vanishing point

Core Animation defines the vanishing point as being located at the `anchorPoint` of the layer being transformed (which is usually the center of the layer, but may not be—see Chapter 3 for details). That is to say, it's located wherever the `anchorPoint` of the view was positioned *prior* to applying the transform; if the transform includes a translation component that moves the layer to somewhere else onscreen, the vanishing point will be wherever it was located before it was transformed.

When you change the position of a layer, you also change its vanishing point. This is important to remember when you are working in 3D. If you intend to adjust the `m34` property of a layer to make it appear three-dimensional, you should position it in the center of the screen and then move it to its final location using a translation (instead of changing its `position`) so that it shares a common vanishing point with any other 3D layers on the screen.

The `sublayerTransform` Property

If you have multiple views or layers, each with 3D transforms, it is necessary to apply the same `m34` value to each individually and to ensure that they all share a common position in the center of the screen prior to being transformed. This is relatively straightforward if you define a constant or function to create and position them all for you, but it is still restrictive (for example, it prevents you from arranging your views in Interface Builder). There is a better way.

`CALayer` has another transform property called `sublayerTransform`. This is also a `CATransform3D`, but instead of transforming the layer to which it is applied, it affects only the sublayers. This means you can apply a perspective transform once and for all to a single container layer, and the sublayers will all inherit that perspective automatically.

Only having to set the perspective transform in one place is convenient in itself, but it also carries another significant benefit: The vanishing point is set as the center of the *container layer*, not set individually for each sublayer. This means that you are free to position the sublayers using their position or frame instead of having to set them all to the center of the screen and move them with transforms to keep their vanishing point consistent.

Let's demonstrate this with an example. We'll place two views side by side in Interface Builder (see Figure 5.12). Now by setting the perspective transform on their containing view, we can apply the same perspective and vanishing point to both of them. See Listing 5.6 for the code and Figure 5.13 for the result.

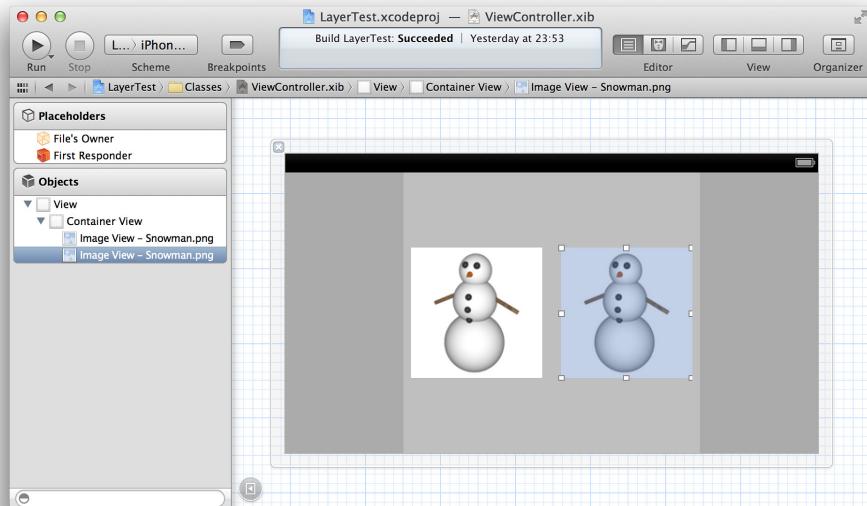


Figure 5.12 Two views arranged side by side within a container view

Listing 5.6 Applying a sublayerTransform

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic, weak) IBOutlet UIView *layerView1;  
@property (nonatomic, weak) IBOutlet UIView *layerView2;  
  
@end  
  
@implementation ViewController
```

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //apply perspective transform to container
    CATransform3D perspective = CATransform3DIdentity;
    perspective.m34 = - 1.0 / 500.0;
    self.containerView.layer.sublayerTransform = perspective;

    //rotate layerView1 by 45 degrees along the Y axis
    CATransform3D transform1 = CATransform3DMakeRotation(M_PI_4, 0, 1, 0);
    self.layerView1.layer.transform = transform1;

    //rotate layerView2 by 45 degrees along the Y axis
    CATransform3D transform2 = CATransform3DMakeRotation(-M_PI_4, 0, 1, 0);
    self.layerView2.layer.transform = transform2;
}

@end
```

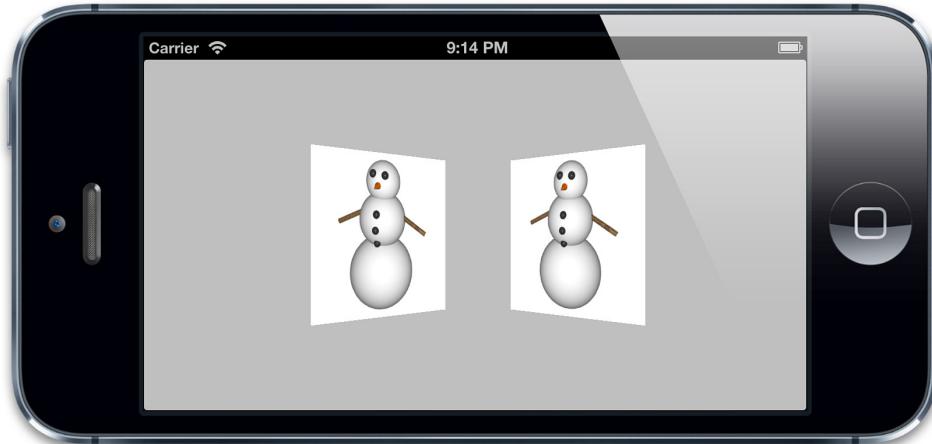


Figure 5.13 Two individually transformed views with shared perspective

Backfaces

Now that we can rotate our layers in 3D, we can look at them *from behind*. If we change the angle in Listing 5.4 to `M_PI` (180 degrees) rather than `M_PI_4` (45 degrees) as it is currently, we will have rotated the view a full half-circle, such that it is facing directly away from the camera.

What does a layer look like from the back? See Figure 5.14 to find out.



Figure 5.14 The rear side of our view, showing a mirrored snowman image

As you can see, layers are double-sided; the reverse side shows a mirror image of the front.

This isn't necessarily a desirable feature, though. If your layer contains text or controls, it's going to be very confusing if the user sees the mirror image of these. It's also potentially wasteful: Imagine a solid object such as an opaque cube formed from layers—why waste GPU cycles drawing the layers on the reverse side of the cube if we can never see them?

`CALayer` has a property called `doubleSided` that controls whether the reverse side of a layer should be drawn. The `doubleSided` property is a `BOOL` and defaults to `YES`. If you set it to `NO`, then when the layer is facing away from the camera, it will not be drawn at all.

Layer Flattening

What happens if we transform a layer containing another layer that has itself been transformed in the opposite direction? Confused? See Figure 5.15.

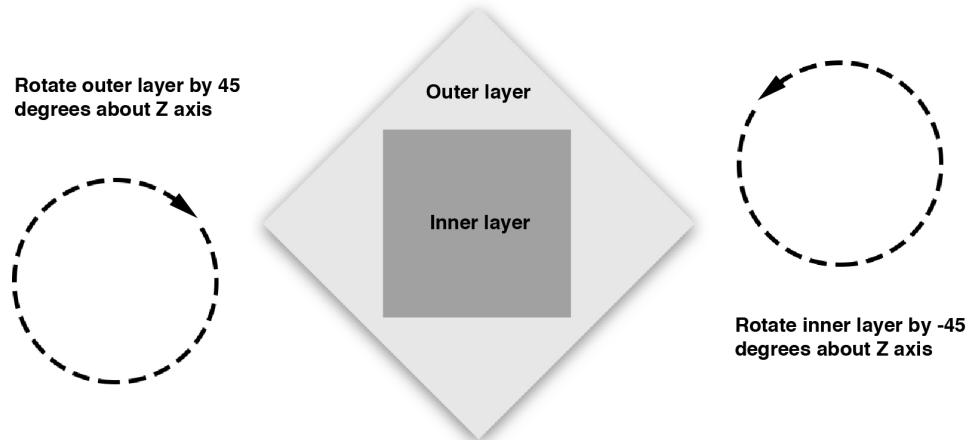


Figure 5.15 Nested layers with opposite transforms applied

Note how the negative 45-degree rotation of the inner layer cancels out the 45-degree rotation of outer layer so that the inner layer is pointing the right way up.

Logically, if an inner layer has the opposite transform to its outer layer (in this case, a rotation about the Z axis), then we would expect the two transforms to cancel each other out.

Let's verify that this is the case in practice. Listing 5.7 shows the code to do this, and Figure 5.16 shows the result.

Listing 5.7 Opposite Rotation Transforms Around Z Axis

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *outerView;  
@property (nonatomic, weak) IBOutlet UIView *innerView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //rotate the outer layer 45 degrees  
    CATransform3D outer = CATransform3DMakeRotation(M_PI_4, 0, 0, 1);
```

```

self.outerView.layer.transform = outer;

//rotate the inner layer -45 degrees
CATransform3D inner = CATransform3DMakeRotation(-M_PI_4, 0, 0, 1);
self.innerView.layer.transform = inner;
}

@end

```

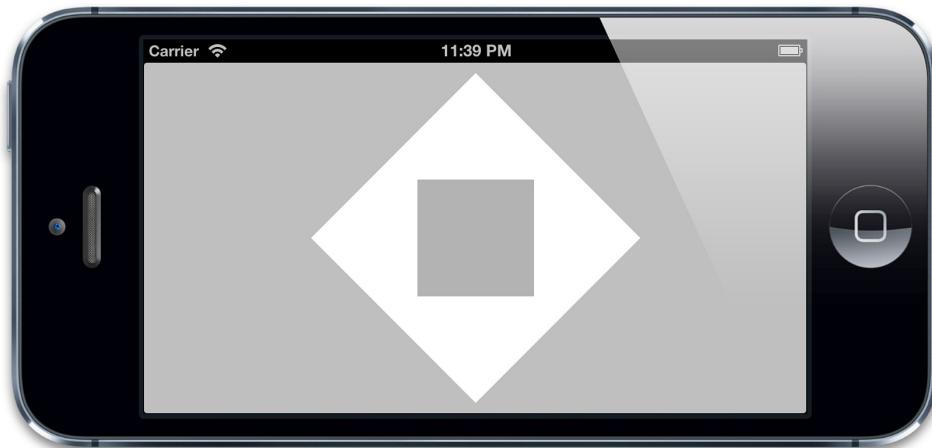


Figure 5.16 The rotated views match the predicted behavior in Figure 5.15.

That seems to have worked as expected. Now let's try it in 3D. We'll modify our code to rotate the inner and outer views along the Y axis instead of the Z axis, and also add perspective so we can see more clearly what's going on. We can't use the `sublayerTransform` trick from Listing 5.6 because our inner layer is not a direct sublayer of the container, so we'll just apply the perspective transform separately to each layer (see Listing 5.8).

Listing 5.8 Opposite Rotation Transforms Around Y Axis

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    //rotate the outer layer 45 degrees

```

```

CATransform3D outer = CATransform3DIdentity;
outer.m34 = -1.0 / 500.0;
outer = CATransform3DRotate(outer, M_PI_4, 0, 1, 0);
self.outerView.layer.transform = outer;

//rotate the inner layer -45 degrees
CATransform3D inner = CATransform3DIdentity;
inner.m34 = -1.0 / 500.0;
inner = CATransform3DRotate(inner, -M_PI_4, 0, 1, 0);
self.innerView.layer.transform = inner;
}

```

We should expect to see something like Figure 5.17.

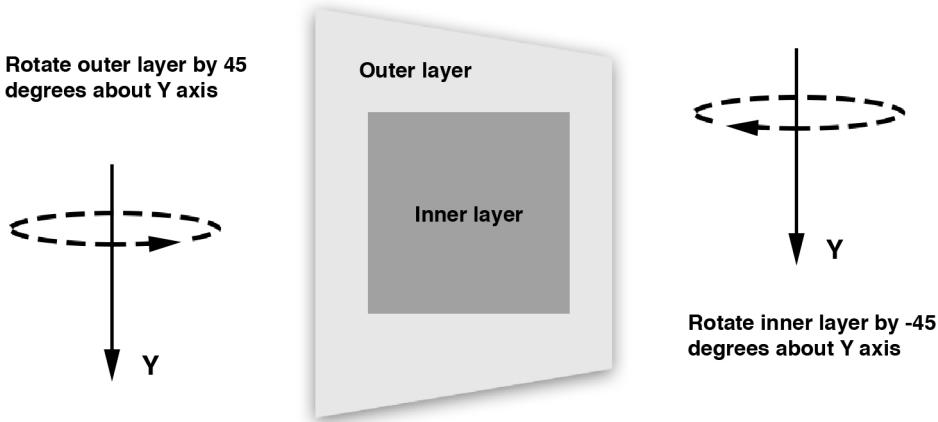


Figure 5.17 Expected result of opposite rotations around the Y axis

But that's not what we see. Instead, what we see looks like Figure 5.18. What's happened? Our inner layer is still noticeably tilted to the left, and it's also distorted; it was supposed to be face-on and square!

It turns out that although Core Animation layers exist in 3D space, they don't all exist in *the same* 3D space. The 3D scene within each layer is flattened. When you look at a layer from face on, you see the *illusion* of a 3D scene created by its sublayers, but as you tilt the layer away, you realize that 3D scene is just painted on the layer surface.

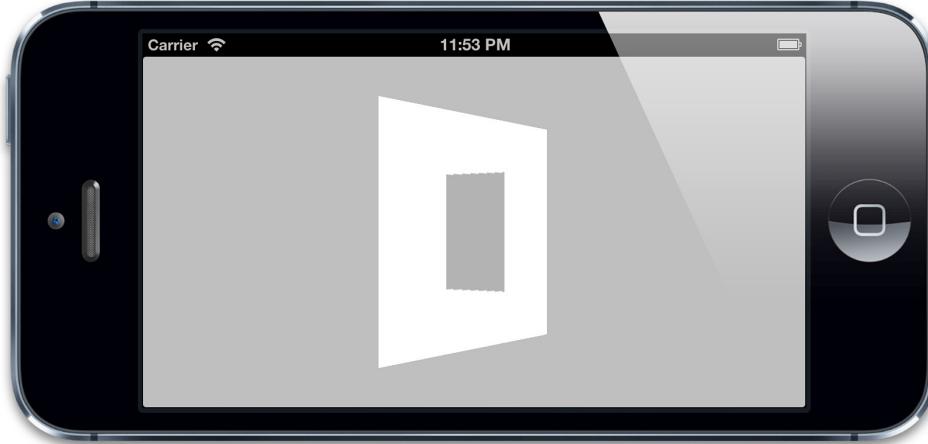


Figure 5.18 Actual result of opposite rotations around the Y axis

To draw an analogy, it's just like tilting the screen away when you are playing a 3D game. You might see a wall in front of you in the game, but tilting the screen won't allow you to peer around the wall. The scene displayed on the screen doesn't change depending on the angle at which you look at it; neither do the contents of a layer.

This makes it difficult to create very complex 3D scenes using Core Animation. You cannot use the layer tree to build a hierarchical 3D structure—any 3D surfaces in the same scene must be siblings within the same layer because each parent flattens its children.

At least, that's true if you use regular `CALayer` instances. There is a `CALayer` subclass called `CATransformLayer` designed to deal with this problem. This is covered in Chapter 6, “Specialized Layers.”

Solid Objects

Now that you understand the basics of positioning layers in 3D space, let's try constructing a solid 3D object (well, technically a *hollow* object, but it will *appear* solid). We'll create a cube by using six separate views to construct the faces.

For the purposes of our example, the cube faces are arranged in Interface Builder (see Figure 5.19). We could create the faces in code, but the advantage of using Interface Builder is that we can easily add and arrange subviews within each face. Remember that these faces are ordinary user interface elements that can contain other views and controls. They are fully fledged, interactive parts of our interface, and will remain so even after we fold them up into a cube.

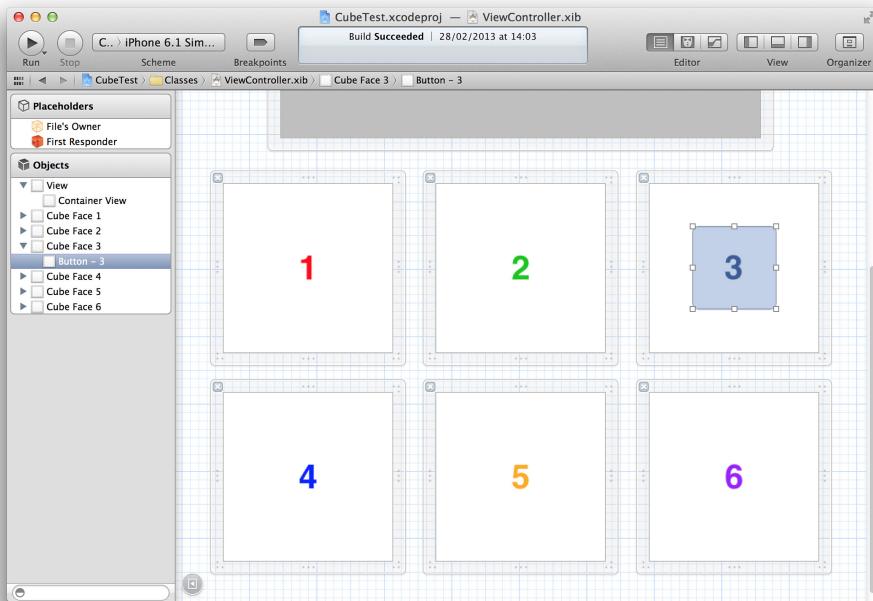


Figure 5.19 The six cube face views laid out in Interface Builder

The face views are not placed inside the main view but have been arranged loosely in the root of the nib file. We don't care about setting the location of these views within their container as we are going to position them programmatically using the layer `transform`, and it's useful to place them outside of the container view in Interface Builder so that we can easily see their contents. If they all were crammed on top of one another inside the main view, this would be awkward.

We've placed a colored `UILabel` inside each of the views so we can easily identify which is which. Note also the `UIButton` placed in the third face view; this is explained shortly.

Listing 5.9 shows the code to arrange the views in a cube, and Figure 5.20 shows the result.

Listing 5.9 Creating a Cube

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic, strong) IBOutletCollection(UIView) NSArray *faces;  
  
@end
```

```
@implementation ViewController

- (void)addFace:(NSInteger)index withTransform:(CATransform3D)transform
{
    //get the face view and add it to the container
    UIView *face = self.faces[index];
    [self.containerView addSubview:face];

    //center the face view within the container
    CGSize containerSize = self.containerView.bounds.size;
    face.center = CGPointMake(containerSize.width / 2.0,
                              containerSize.height / 2.0);

    //apply the transform
    face.layer.transform = transform;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set up the container sublayer transform
    CATransform3D perspective = CATransform3DIdentity;
    perspective.m34 = -1.0 / 500.0;
    self.containerView.layer.sublayerTransform = perspective;

    //add cube face 1
    CATransform3D transform = CATransform3DMakeTranslation(0, 0, 100);
    [self addFace:0 withTransform:transform];

    //add cube face 2
    transform = CATransform3DMakeTranslation(100, 0, 0);
    transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
    [self addFace:1 withTransform:transform];

    //add cube face 3
    transform = CATransform3DMakeTranslation(0, -100, 0);
    transform = CATransform3DRotate(transform, M_PI_2, 1, 0, 0);
    [self addFace:2 withTransform:transform];

    //add cube face 4
    transform = CATransform3DMakeTranslation(0, 100, 0);
    transform = CATransform3DRotate(transform, -M_PI_2, 1, 0, 0);
    [self addFace:3 withTransform:transform];
}
```

```

//add cube face 5
transform = CATransform3DMakeTranslation(-100, 0, 0);
transform = CATransform3DRotate(transform, -M_PI_2, 0, 1, 0);
[self addFace:4 withTransform:transform];

//add cube face 6
transform = CATransform3DMakeTranslation(0, 0, -100);
transform = CATransform3DRotate(transform, M_PI, 0, 1, 0);
[self addFace:5 withTransform:transform];
}

@end

```



Figure 5.20 The cube, displayed face-on

Our cube isn't really that impressive from this angle; it just looks like a square. To appreciate it properly, we need to look at it from a *different point of view*.

Rotating the cube itself would be cumbersome because we'd have to rotate each face individually. A simpler option is to rotate the *camera*, which we can do by adjusting the `sublayerTransform` of our container view.

Add the following lines to rotate the perspective transform matrix before applying it to the `containerView` layer:

```

perspective = CATransform3DRotate(perspective, -M_PI_4, 1, 0, 0);
perspective = CATransform3DRotate(perspective, -M_PI_4, 0, 1, 0);

```

That has the effect of rotating the camera (or rotating the entire scene relative to the camera, depending on how you look at it) 45 degrees around the Y axis, and then a further 45 degrees around the X axis. We are now viewing the cube corner-on and can see it for what it really is (see Figure 5.21).

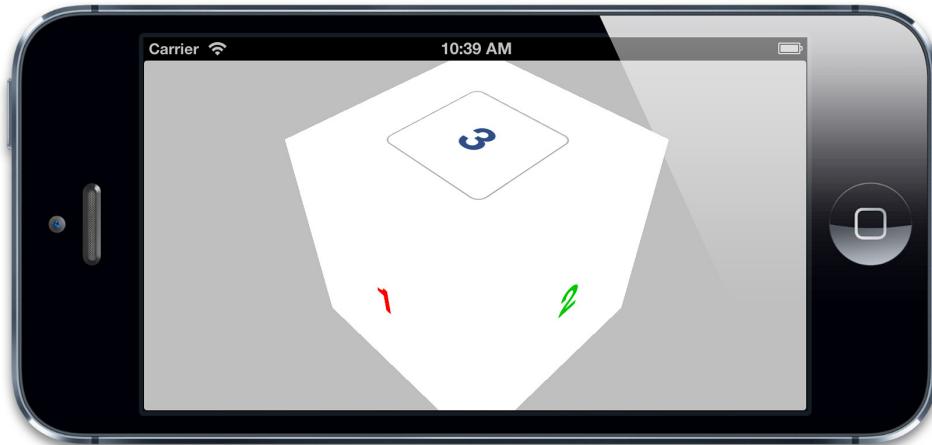


Figure 5.21 The cube, displayed corner-on

Light and Shadow

It definitely looks more like a cube now, but it's difficult to make out the joins between the faces. Core Animation can display layers in 3D, but it has no concept of *lighting*. If you want to make your cube look more realistic, you'll have to apply your own shading effects. You can do this by adjusting the different view background colors or by using images with lighting effects pre-applied to them.

If you need to create *dynamic* lighting effects, you can do so by overlaying each view with a translucent black shadow layer with varying alpha based on the view orientation. To calculate the opacity of the shadow layer, you need to get the *normal vector* for each face (a vector that points perpendicular to the surface) and then calculate the *cross product* between that vector and the vector from an imaginary light source. The cross product gives you the angle between the light source and the layer, which indicates the extent to which it should be illuminated.

An implementation of this idea is shown in Listing 5.10. We've used the GLKit framework to do the vector calculations (you'll need to include this framework in your project to run the code). The `CATransform3D` for each face is cast to a `GLKMatrix4` using some pointer trickery, and then the 3×3 *rotation matrix* is extracted using the

`GLKMatrix4GetMatrix3` function. The rotation matrix is the part of the transform that specifies the layer's orientation, and we can use it to calculate the normal vector.

Figure 5.22 shows the result. Try tweaking the `LIGHT_DIRECTION` vector and `AMBIENT_LIGHT` value to alter the lighting effect.

Listing 5.10 Applying Dynamic Lighting Effects to the Cube Faces

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>
#import <GLKit/GLKit.h>

#define LIGHT_DIRECTION 0, 1, -0.5
#define AMBIENT_LIGHT 0.5

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) IBOutletCollection(UIView) NSArray *faces;

@end

@implementation ViewController

- (void)applyLightingToFace:(CALayer *)face
{
    //add lighting layer
    CALayer *layer = [CALayer layer];
    layer.frame = face.bounds;
    [face addSublayer:layer];

    //convert the face transform to matrix
    // (GLKMatrix4 has the same structure as CATransform3D)
    CATransform3D transform = face.transform;
    GLKMatrix4 matrix4 = *(GLKMatrix4 *)&transform;
    GLKMatrix3 matrix3 = GLKMatrix4GetMatrix3(matrix4);

    //get face normal
    GLKVector3 normal = GLKVector3Make(0, 0, 1);
    normal = GLKMatrix3MultiplyVector3(matrix3, normal);
    normal = GLKVector3Normalize(normal);

    //get dot product with light direction
    GLKVector3 light = GLKVector3Normalize(GLKVector3Make(LIGHT_DIRECTION));
    float dotProduct = GLKVector3DotProduct(light, normal);
```

```

//set lighting layer opacity
CGFloat shadow = 1 + dotProduct - AMBIENT_LIGHT;
UIColor *color = [UIColor colorWithWhite:0 alpha:shadow];
layer.backgroundColor = color.CGColor;
}

- (void)addFace:(NSInteger)index withTransform:(CATransform3D)transform
{
    //get the face view and add it to the container
    UIView *face = self.faces[index];
    [self.containerView addSubview:face];

    //center the face view within the container
    CGSize containerSize = self.containerView.bounds.size;
    face.center = CGPointMake(containerSize.width / 2.0,
                             containerSize.height / 2.0);

    //apply the transform
    face.layer.transform = transform;

    //apply lighting
    [self applyLightingToFace:face.layer];
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set up the container sublayer transform
    CATransform3D perspective = CATransform3DIdentity;
    perspective.m34 = -1.0 / 500.0;
    perspective = CATransform3DRotate(perspective, -M_PI_4, 1, 0, 0);
    perspective = CATransform3DRotate(perspective, -M_PI_4, 0, 1, 0);
    self.containerView.layer.sublayerTransform = perspective;

    //add cube face 1
    CATransform3D transform = CATransform3DMakeTranslation(0, 0, 100);
    [self addFace:0 withTransform:transform];

    //add cube face 2
    transform = CATransform3DMakeTranslation(100, 0, 0);
    transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
    [self addFace:1 withTransform:transform];

    //add cube face 3
    transform = CATransform3DMakeTranslation(0, -100, 0);
}

```

```

transform = CATransform3DRotate(transform, M_PI_2, 1, 0, 0);
[self addFace:2 withTransform:transform];

//add cube face 4
transform = CATransform3DMakeTranslation(0, 100, 0);
transform = CATransform3DRotate(transform, -M_PI_2, 1, 0, 0);
[self addFace:3 withTransform:transform];

//add cube face 5
transform = CATransform3DMakeTranslation(-100, 0, 0);
transform = CATransform3DRotate(transform, -M_PI_2, 0, 1, 0);
[self addFace:4 withTransform:transform];

//add cube face 6
transform = CATransform3DMakeTranslation(0, 0, -100);
transform = CATransform3DRotate(transform, M_PI, 0, 1, 0);
[self addFace:5 withTransform:transform];
}

@end

```

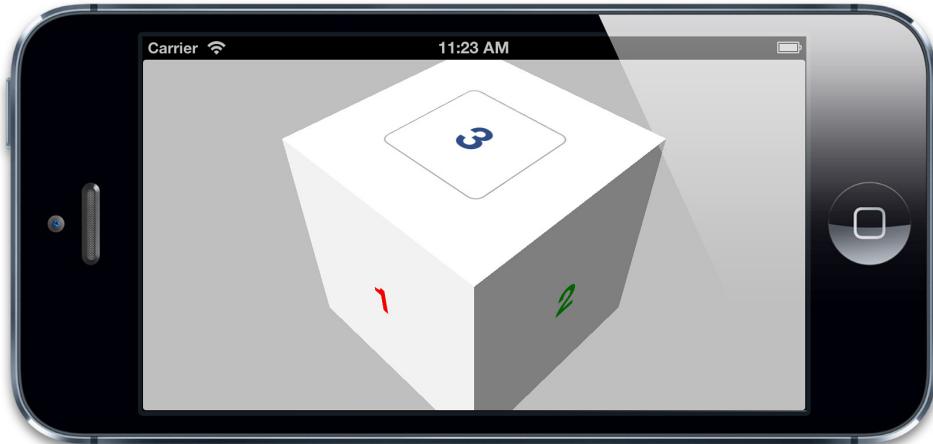


Figure 5.22 The cube, now with dynamically calculated lighting

Touch Events

You may have also noticed that we can now see the button on the third face. If you press it, nothing happens. Why is that?

It's not because iOS cannot correctly transform the touch events to match the button position in 3D; it's actually quite capable of doing that. The problem is the *view order*. As we mentioned briefly in Chapter 3, touch events are processed according to the order of views within their superview, not their Z-position in 3D space. When we added our cube face views, we added them in numeric order, so faces 4, 5, and 6 are in front of face 3 in the view/layer order.

Even though we can't see faces 4, 5, and 6 (because they are obscured by faces 1, 2, and 3), iOS still gives them first dibs on touch events. When we try to tap the button on face 3, face 5 or 6 (depending on where we tap) is intercepting the touch event, just as it would if we had placed it in front of the button in a normal 2D layout.

You might think that setting `doubleSided` to `NO` might help here, as it would render the rearward facing views invisible, but unfortunately that doesn't help; views that are hidden because they are facing away from the camera will still intercept touch events (unlike views that are hidden using the `hidden` property, or by setting their `alpha` to zero, which don't), so disabling double-sided rendering won't help with this issue (although it might be worth doing anyway for performance reasons).

There are a couple of solutions to this: We could set `userInteractionEnabled` to `NO` on all of our cube face views except face 3 so that they don't receive touches. Or we could simply add face number 3 to the view hierarchy *after* face number 6 in our program. Either way, we will then be able to press the button (see Figure 5.23).



Figure 5.23 Now the background views aren't blocking the button, we can press it.

Summary

This chapter covered 2D and 3D transforms. You learned a bit about matrix math, and how to create 3D scenes with Core Animation. You saw what the back of a layer looks like and learned that you can't peer around objects in a flat image. Finally, this chapter demonstrated that when it comes to handling touch events, the order of views or layers in the hierarchy is more significant than their apparent order onscreen.

Chapter 6 takes a look at the specialized `CALayer` subclasses provided by Core Animation and their various capabilities.

This page intentionally left blank

6

Specialized Layers

Specialization is a feature of every complex organization.

Catharine R. Stimpson

Up to this point, we have been working with the `CALayer` class, and we have seen that it has some useful image drawing and transformation capabilities. But Core Animation layers can be used for more than just images and colors. This chapter explores some of the other layer classes that you can use to extend Core Animation’s drawing capabilities.

CAShapeLayer

In Chapter 4, “Visual Effects,” you learned how to use `CGPath` to create arbitrarily shaped shadows without using images. It would be neat if we could create arbitrarily shaped layers in the same way.

`CAShapeLayer` is a layer subclass that draws itself using vector graphics instead of a bitmap image. You specify attributes such as color and line thickness, define the desired shape using a `CGPath`, and `CAShapeLayer` renders it automatically. Of course, you could use Core Graphics to draw a path directly into the contents of an ordinary `CALayer` (as in Chapter 2, “The Backing Image”), but there are several advantages to using `CAShapeLayer` instead:

- **It’s fast**—`CAShapeLayer` uses hardware-accelerated drawing and is much faster than using Core Graphics to draw an image.
- **It’s memory efficient**—A `CAShapeLayer` does not have to create a backing image like an ordinary `CALayer` does, so no matter how large it gets, it won’t consume much memory.

- **It doesn't get clipped to the layer bounds**—A `CAShapeLayer` can happily draw outside of its bounds. Your path will not get clipped like it does when you draw into a regular `CALayer` using Core Graphics (as you saw in Chapter 2).
- **There's no pixelation**—When you transform a `CAShapeLayer` by scaling it up or moving it closer to the camera with a 3D perspective transform, it does not become pixelated like an ordinary layer's backing image would.

Creating a CGPath

`CAShapeLayer` can be used to draw any shape that can be represented by a `CGPath`. The shape doesn't have to be closed, and the path doesn't have to be unbroken, so you can actually draw several distinct shapes in a single layer. You can control the `strokeColor` and `fillColor` of the path, along with other properties such as `lineWidth` (line thickness, in points), `lineCap` (how the ends of lines look), and `lineJoin` (how the joints between lines look); but you can only set these properties once, at the layer level. If you want to draw multiple shapes with different colors or styles, you have to use a separate layer for each shape.

Listing 6.1 shows the code for a simple stick figure drawing, rendered using a single `CAShapeLayer`. The `CAShapeLayer` path property is defined as a `CGPathRef`, but we've created the path using the `UIBezierPath` helper class, which saves us from having to worry about manually releasing the `CGPath`. Figure 6.1 shows the result. It's not exactly a Rembrandt, but you get the idea!

Listing 6.1 Drawing a Stick Figure Using `CAShapeLayer`

```
#import "DrawingView.h"
#import <QuartzCore/QuartzCore.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create path
    UIBezierPath *path = [[UIBezierPath alloc] init];
    [path moveToPoint:CGPointMake(175, 100)];
```

```

[path addArcWithCenter:CGPointMake(150, 100)
    radius:25
    startAngle:0
    endAngle:2*M_PI
    clockwise:YES];
[path moveToPoint:CGPointMake(150, 125)];
[path addLineToPoint:CGPointMake(150, 175)];
[path addLineToPoint:CGPointMake(125, 225)];
[path moveToPoint:CGPointMake(150, 175)];
[path addLineToPoint:CGPointMake(175, 225)];
[path moveToPoint:CGPointMake(100, 150)];
[path addLineToPoint:CGPointMake(200, 150)];

//create shape layer
CAShapeLayer *shapeLayer = [CAShapeLayer layer];
shapeLayer.strokeColor = [UIColor redColor].CGColor;
shapeLayer.fillColor = [UIColor clearColor].CGColor;
shapeLayer.lineWidth = 5;
shapeLayer.lineJoin = kCALineJoinRound;
shapeLayer.lineCap = kCALineCapRound;
shapeLayer.path = path.CGPath;

//add it to our view
[self.containerView.layer addSublayer:shapeLayer];
}

@end

```



Figure 6.1 A simple stick figure displayed using CAShapeLayer

Rounded Corners, Redux

Chapter 2 mentioned that `CAShapeLayer` provides an alternative way to create a view with rounded corners, as opposed to using the `CALayer.cornerRadius` property. Although using a `CAShapeLayer` is a bit more work, it has the advantage that it allows us to specify the radius of each corner independently.

We could create a rounded rectangle path manually using individual straight lines and arcs, but `UIBezierPath` actually has some convenience constructors for creating rounded rectangles automatically. The following code snippet produces a path with three rounded corners and one sharp:

```
//define path parameters
CGRect rect = CGRectMake(50, 50, 100, 100);
CGSize radii = CGSizeMake(20, 20);
UIRectCorner corners = UIRectCornerTopRight |
    UIRectCornerBottomRight |
    UIRectCornerBottomLeft;

//create path
UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:rect
    byRoundingCorners:corners
    cornerRadii:radii];
```

We can use a `CAShapeLayer` with this path to create a view with mixed sharp and rounded corners. If we want to clip the view's contents to this shape, we can use our `CAShapeLayer` as the `mask` property of the view's backing layer instead of adding it as a sublayer. (See Chapter 4, “Visual Effects,” for a full explanation of layer masks.)

CATextLayer

A user interface cannot be constructed from images alone. A well-designed icon can do a great job of conveying the purpose of a button or control, but sooner or later you're going to need a good old-fashioned text label.

If you want to display text in a layer, there is nothing stopping you from using the layer delegate to draw a string directly into the layer contents with Core Graphics (which is essentially how `UILabel` works). This is a cumbersome approach if you are working directly with layers, though, instead of layer-backed views. You would need to create a class that can act as the layer delegate for each layer that displays text, and then write logic to determine which layer should display which string, not to mention keeping track of the different fonts, colors, and so on.

Fortunately, this is unnecessary. Core Animation provides a subclass of `CALayer` called `CATextLayer` that encapsulates most of the string drawing features of `UILabel` in layer form and adds a few extra features for good measure.

CATextLayer also renders much faster than UILabel. It's a little-known fact that on iOS6 and earlier, UILabel actually uses WebKit to do its text drawing, which carries a significant performance overhead when you are drawing a lot of text. CATextLayer uses Core Text and is significantly faster.

Let's try displaying some text using a CATextLayer. Listing 6.2 shows the code to set up and display a CATextLayer, and Figure 6.2 shows the result.

Listing 6.2 Implementing a Text Label Using CATextLayer

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *labelView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a text layer
    CATextLayer *textLayer = [CATextLayer layer];
    textLayer.frame = self.labelView.bounds;
    [self.labelView.layer addSublayer:textLayer];

    //set text attributes
    textLayer.foregroundColor = [UIColor blackColor].CGColor;
    textLayer.alignmentMode = kCAAlignmentJustified;
    textLayer.wrapped = YES;

    //choose a font
    UIFont *font = [UIFont systemFontOfSize:15];

    //set layer font
    CFStringRef fontName = (_bridge CFStringRef)font.fontName;
    CGFontRef fontRef = CGFontCreateWithName(fontName);
    textLayer.font = fontRef;
    textLayer.fontSize = font.pointSize;
    CGFontRelease(fontRef);

    //choose some text
    NSString *text = @"Lorem ipsum dolor sit amet, consectetur adipiscing \
elit. Quisque massa arcu, eleifend vel varius in, facilisis pulvinar \
leo. Nunc quis nunc at mauris pharetra condimentum ut ac neque. Nunc \
```

```
elementum, libero ut porttitor dictum, diam odio congue lacus, vel \
fringilla sapien diam at purus. Etiam suscipit pretium nunc sit amet \
lobortis";

//set layer text
textLayer.string = text;
}

@end
```

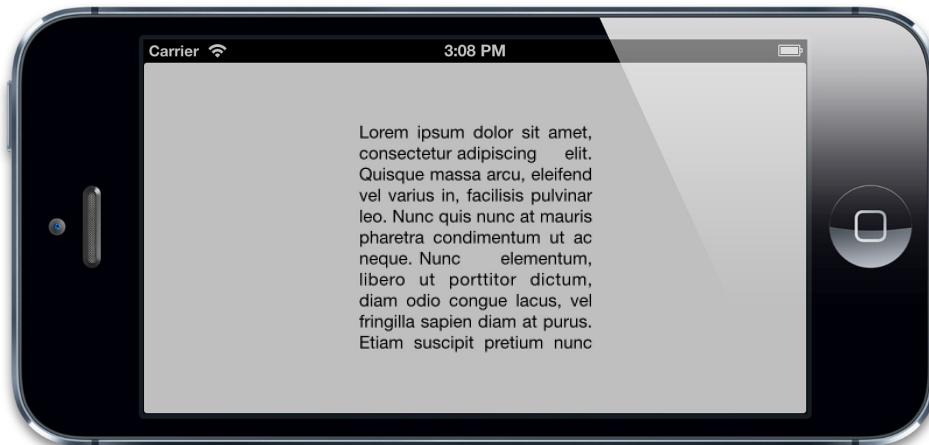


Figure 6.2 A plain text label implemented using `CATextLayer`

If you look at this text closely, you'll see that something is a bit odd; the text is pixelated. That's because it's not being rendered at Retina resolution. Chapter 2 mentioned the `contentsScale` property, which is used to determine the resolution at which the layer contents are rendered. The `contentsScale` property always defaults to 1.0 instead of the screen scale factor. If we want Retina-quality text, we have to set the `contentsScale` of our `CATextLayer` to match the screen scale using the following line of code:

```
textLayer.contentsScale = [UIScreen mainScreen].scale;
```

This solves the pixelation problem (see Figure 6.3).

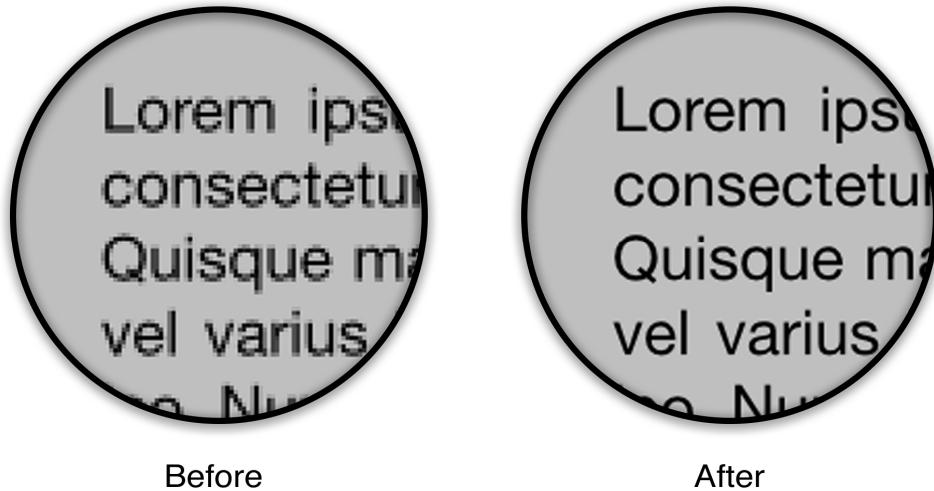


Figure 6.3 The effect of setting the `contentsScale` to match the screen

The `CATextLayer` font property is not a `UIFont`, it's a `CFTyperef`. This allows you to specify the font using either a `CGFontRef` or a `CTFontRef` (a Core Text font), depending on your requirements. The font size is also set independently using the `fontSize` property, because `CTFontRef` and `CGFontRef` do not encapsulate the point size like `UIFont` does. The example shows how to convert from a `UIFont` to a `CGFontRef`.

Also, the `CATextLayer` string property is not an `NSString` as you might expect, but is typed as `id`. This is to allow you the option of using an `NSAttributedString` instead of an `NSString` to specify the text (`NSAttributedString` is not a subclass of `NSString`). Attributed strings are the mechanism that iOS uses for rendering styled text. They specify *style runs*, which are specific ranges of the string to which metadata such as font, color, bold, italic, and so forth are attached.

Rich Text

In iOS 6, Apple added direct support for attributed strings to `UILabel` and to other UIKit text views. This is a handy feature that makes attributed strings much easier to work with, but `CATextLayer` has supported attributed strings since its introduction in iOS 3.2; so if you still need to support earlier iOS versions with your app, `CATextLayer` is a great way to add simple rich text labels to your interface without having to deal with the complexity of Core Text or the hassle of using a `UIWebView`.

Let's modify the example to use an `NSAttributedString` (see Listing 6.3). On iOS 6 and above we could use the new `NSTextAttributeName` constants to set up our string

attributes, but because the point of the exercise is to demonstrate that this feature also works on iOS 5 and below, we've used the Core Text equivalents instead. This means that you'll need to add the Core Text framework to your project; otherwise, the compiler won't recognize the attribute constants.

Figure 6.4 shows the result. (Note the red, underlined text.)

Listing 6.3 Implementing a Rich Text Label Using `NSAttributedString`

```
#import "DrawingView.h"
#import <QuartzCore/QuartzCore.h>
#import <CoreText/CoreText.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *labelView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a text layer
    CATextLayer *textLayer = [CATextLayer layer];
    textLayer.frame = self.labelView.bounds;
    textLayer.contentsScale = [UIScreen mainScreen].scale;
    [self.labelView.layer addSublayer:textLayer];

    //set text attributes
    textLayer.alignmentMode = kCAAlignmentJustified;
    textLayer.wrapped = YES;

    //choose a font
    UIFont *font = [UIFont systemFontOfSize:15];

    //choose some text
    NSString *text = @"Lorem ipsum dolor sit amet, consectetur adipiscing \
elit. Quisque massa arcu, eleifend vel varius in, facilisis pulvinar \
leo. Nunc quis nunc at mauris pharetra condimentum ut ac neque. Nunc \
elementum, libero ut porttitor dictum, diam odio congue lacus, vel \
fringilla sapien diam at purus. Etiam suscipit pretium nunc sit amet \
lobortis";
```

```
//create attributed string
NSMutableAttributedString *string = nil;
string = [[NSMutableAttributedString alloc] initWithString:text];

//convert UIFont to a CTFont
CFStringRef fontName = (_bridge CFStringRef)font.fontName;
CGFloat fontSize = font.pointSize;
CTFontRef fontRef = CTFontCreateWithName(fontName, fontSize, NULL);

//set text attributes
NSDictionary *attribs = @{
    (_bridge id)kCTForegroundColorAttributeName:
        (_bridge id)[UIColor blackColor].CGColor,
    (_bridge id)kCTFontAttributeName: (_bridge id)fontRef
};
[string setAttributes:attribs range:NSMakeRange(0, [text length])];
attribs = @{
    (_bridge id)kCTForegroundColorAttributeName:
        (_bridge id)[UIColor redColor].CGColor,
    (_bridge id)kCTUnderlineStyleAttributeName:
        @(kCTUnderlineStyleSingle),
    (_bridge id)kCTFontAttributeName: (_bridge id)fontRef
};
[string setAttributes:attribs range:NSMakeRange(6, 5)];

//release the CTFont we created earlier
CFRelease(fontRef);

//set layer text
textLayer.string = string;
}

@end
```

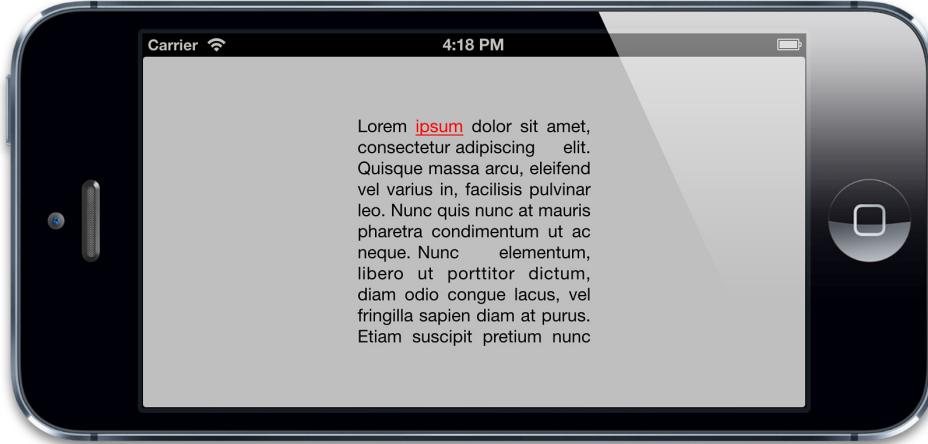


Figure 6.4 A rich text label implemented using `CATextLayer`

Leading and Kerning

It's worth mentioning that the *leading* (line spacing) and *kerning* (spacing between letters) for text rendered using `CATextLayer` is not completely identical to that of the string rendering used by `UILabel` due to the different drawing implementations (Core Text and WebKit, respectively).

The extent of the discrepancy varies (depending on the specific font and characters used) and is generally fairly minor, but you should keep this mind if you are trying to exactly match appearance between regular labels and a `CATextLayer`.

A `UILabel` Replacement

We've established that `CATextLayer` has performance benefits over `UILabel`, as well as some additional layout options and support for rich text on iOS 5. But it's fairly cumbersome to use by comparison to a regular label. If we want to make a truly usable replacement for `UILabel`, we should be able to create our labels in Interface Builder, and they should behave as much as possible like regular views.

We could subclass `UILabel` and override its methods to display the text in a `CATextLayer` that we've added as a sublayer, but we'd still have the redundant empty backing image created by the presence of `UILabel -drawRect:` method. And because `CALayer` doesn't support autoresizing or autolayout, a sublayer wouldn't track the size of the view bounds automatically, so we would need to manually update the sublayer bounds every time the view is resized.

What we really want is a `UILabel` subclass that actually uses a `CATextLayer` as its *backing layer*, then it would automatically resize with the view and there would be no redundant backing image to worry about.

As we discussed in Chapter 1, “The Layer Tree,” every `UIView` is backed by an instance of `CALayer`. That layer is automatically created and managed by the view, so how can we substitute a different layer type? We can’t replace the layer once it has been created, but if we subclass `UIView`, we can override the `+layerClass` method to return a different layer subclass at creation time. `UIView` calls the `+layerClass` method during its initialization, and uses the class it returns to create its backing layer.

Listing 6.4 shows the code for a `UILabel` subclass called `LayerLabel` that draws its text using a `CATextLayer` instead of the using the slower `-drawRect:` approach that an ordinary `UILabel` uses. `LayerLabel` instances can either be created programmatically or via Interface Builder by adding an ordinary label to the view and setting its class to `LayerLabel`.

Listing 6.4 `LayerLabel`, a `UILabel` Subclass That Uses `CATextLayer`

```
#import "LayerLabel.h"
#import <QuartzCore/QuartzCore.h>

@implementation LayerLabel

+ (Class)layerClass
{
    //this makes our label create a CATextLayer
    //instead of a regular CALayer for its backing layer
    return [CATextLayer class];
}

- (CATextLayer *)textLayer
{
    return (CATextLayer *)self.layer;
}

- (void)setUp
{
    //set defaults from UILabel settings
    self.text = self.text;
    self.textColor = self.textColor;
    self.font = self.font;

    //we should really derive these from the UILabel settings too
    //but that's complicated, so for now we'll just hard-code them
    [self.textLayer].alignmentMode = kCAAlignmentJustified;
}
```

```
[self.textLayer].wrapped = YES;

[self.layer display];
}

- (id)initWithFrame:(CGRect)frame
{
    //called when creating label programmatically
    if (self = [super initWithFrame:frame])
    {
        [self setUp];
    }
    return self;
}

- (void)awakeFromNib
{
    //called when creating label using Interface Builder
    [self setUp];
}

- (void)setText:(NSString *)text
{
    super.text = text;

    //set layer text
    [self.textLayer].string = text;
}

- (void)setTextColor:(UIColor *)textColor
{
    super.textColor = textColor;

    //set layer text color
    [self.textLayer].foregroundColor = textColor.CGColor;
}

- (void)setFont:(UIFont *)font
{
    super.font = font;

    //set layer font
    CFStringRef fontName = (__bridge CFStringRef)font.fontName;
    CGFontRef fontRef = CGFontCreateWithName(fontName);
    [self.textLayer].font = fontRef;
    [self.textLayer].fontSize = font.pointSize;
}
```

```
CGFontRelease(fontRef);  
}  
  
@end
```

If you run the sample code, you'll notice that the text isn't pixelated even though we aren't setting the `contentsScale` anywhere. Another benefit of implementing `CATextLayer` as a backing layer is that its `contentsScale` is automatically set by the view.

In this simple example, we've only implemented a few of the styling and layout properties of `UILabel`, but with a bit more work we could create a `LayerLabel` class that supports the full functionality of `UILabel` and more (you will find several such classes already available as open source projects online).

If you only intend to support iOS 6 and above, a `CATextLayer`-based label may be of limited use. But in general, using `+layerClass` to create views backed by different layer types is a clean and reusable way to utilize `CALayer` subclasses in your apps.

CATransformLayer

When constructing complex objects in 3D, it is convenient to be able to organize the individual elements hierarchically. For example, suppose you were making an arm: You would want the hand to be a child of the wrist, which would be a child of the forearm, which would be a child of the elbow, which would be a child of the upper arm, which would be a child of the shoulder, and so on.

The reason for this is that it allows you to move each section independently. Pivoting the elbow would move the lower arm and hand but not the shoulder. Core Animation layers easily allow for this kind of hierarchical arrangement in 2D, but in 3D it's not possible because each layer flattens its children into a single plane (as explained in Chapter 5, "Transforms").

`CATransformLayer` solves this problem. A `CATransformLayer` is unlike a regular `CALayer` in that it cannot display any content of its own; it exists only to host a transform that can be applied to its sublayers. `CATransformLayer` does not flatten its sublayers, so it can be used to construct a hierarchical 3D structure, such as our arm example.

Creating an arm programmatically would require rather a lot of code, so we'll demonstrate this with something a bit simpler: In the cube example in Chapter 5, we worked around the layer-flattening problem by rotating the *camera* instead of the cube by using the `sublayerTransform` of the containing layer. This is a neat trick, but only works for a single object. If our scene contained two cubes, we would not be able to rotate them independently using this technique.

So, let's try it using a `CATransformLayer` instead. The first problem to address is that we constructed our cube in Chapter 5 using views rather than standalone layers. We cannot place a view-backing layer inside another layer that is not itself a view-backing layer without messing up the view hierarchy. We could create a new `UIView` subclass backed by a `CATransformLayer` (using the `+layerClass` method), but to keep things simple for our example, let's just re-create the cube using standalone layers instead of views. This means we can't display buttons and labels on our cube faces like we did in Chapter 5, but we don't need to do that right now.

Listing 6.5 contains the code. We position each cube face using the same basic logic we used in Chapter 5. But instead of adding the cube faces directly to the container view's backing layer as we did before, we place them inside a `CATransformLayer` to create a standalone cube object, and then place two such cubes into our container. We've colored the cube faces randomly so as to make it easier to distinguish them without labels or lighting. Figure 6.5 shows the result.

Listing 6.5 Assembling a 3D Layer Hierarchy Using `CATransformLayer`

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (CALayer *)faceWithTransform:(CATransform3D)transform
{
    //create cube face layer
    CALayer *face = [CALayer layer];
    face.frame = CGRectMake(-50, -50, 100, 100);

    //apply a random color
    CGFloat red = (rand() / (double)INT_MAX);
    CGFloat green = (rand() / (double)INT_MAX);
    CGFloat blue = (rand() / (double)INT_MAX);
    face.backgroundColor = [UIColor colorWithRed:red
                                         green:green
                                         blue:blue
                                         alpha:1.0].CGColor;

    //apply the transform and return
    face.transform = transform;
    return face;
}
```

```

- (CALayer *)cubeWithTransform:(CATransform3D)transform
{
    //create cube layer
    CATransformLayer *cube = [CATransformLayer layer];

    //add cube face 1
    CATransform3D ct = CATransform3DMakeTranslation(0, 0, 50);
    [cube addSublayer:[self faceWithTransform:ct]];

    //add cube face 2
    ct = CATransform3DMakeTranslation(50, 0, 0);
    ct = CATransform3DRotate(ct, M_PI_2, 0, 1, 0);
    [cube addSublayer:[self faceWithTransform:ct]];

    //add cube face 3
    ct = CATransform3DMakeTranslation(0, -50, 0);
    ct = CATransform3DRotate(ct, M_PI_2, 1, 0, 0);
    [cube addSublayer:[self faceWithTransform:ct]];

    //add cube face 4
    ct = CATransform3DMakeTranslation(0, 50, 0);
    ct = CATransform3DRotate(ct, -M_PI_2, 1, 0, 0);
    [cube addSublayer:[self faceWithTransform:ct]];

    //add cube face 5
    ct = CATransform3DMakeTranslation(-50, 0, 0);
    ct = CATransform3DRotate(ct, -M_PI_2, 0, 1, 0);
    [cube addSublayer:[self faceWithTransform:ct]];

    //add cube face 6
    ct = CATransform3DMakeTranslation(0, 0, -50);
    ct = CATransform3DRotate(ct, M_PI, 0, 1, 0);
    [cube addSublayer:[self faceWithTransform:ct]];

    //center the cube layer within the container
    CGSize containerSize = self.containerView.bounds.size;
    cube.position = CGPointMake(containerSize.width / 2.0,
                                containerSize.height / 2.0);

    //apply the transform and return
    cube.transform = transform;
    return cube;
}

- (void)viewDidLoad
{

```

```

[super viewDidLoad];

//set up the perspective transform
CATransform3D pt = CATransform3DIdentity;
pt.m34 = -1.0 / 500.0;
self.containerView.layer.sublayerTransform = pt;

//set up the transform for cube 1 and add it
CATransform3D c1t = CATransform3DIdentity;
c1t = CATransform3DTranslate(c1t, -100, 0, 0);
CALayer *cube1 = [self cubeWithTransform:c1t];
[self.containerView.layer addSublayer:cube1];

//set up the transform for cube 2 and add it
CATransform3D c2t = CATransform3DIdentity;
c2t = CATransform3DTranslate(c2t, 100, 0, 0);
c2t = CATransform3DRotate(c2t, -M_PI_4, 1, 0, 0);
c2t = CATransform3DRotate(c2t, -M_PI_4, 0, 1, 0);
CALayer *cube2 = [self cubeWithTransform:c2t];
[self.containerView.layer addSublayer:cube2];

@end

```

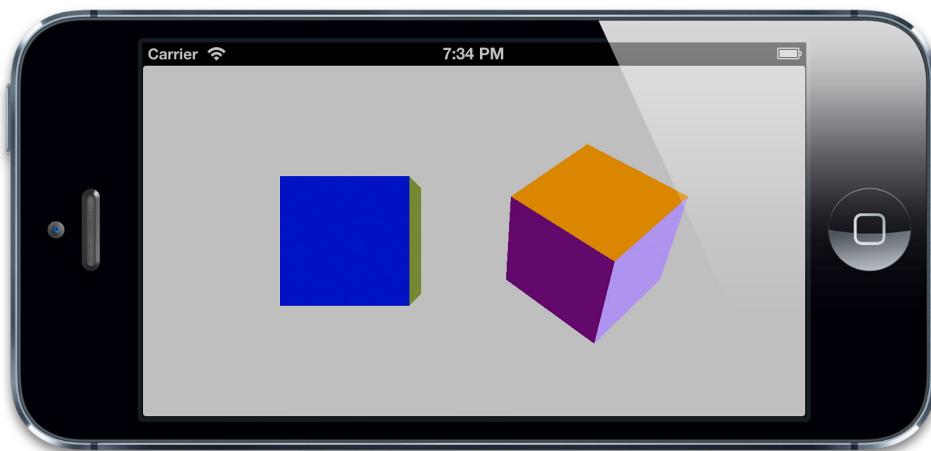


Figure 6.5 Two cubes with shared perspective but different transforms applied

CAGradientLayer

`CAGradientLayer` is used to generate a smooth gradient between two or more colors. It's possible to replicate the appearance of a `CAGradientLayer` using Core Graphics to draw into an ordinary layer's backing image, but the advantage of using a `CAGradientLayer` instead is that the drawing is hardware accelerated.

Basic Gradients

We'll start with a simple diagonal gradient from red to blue (see Listing 6.6). The gradient colors are specified using the `colors` property, which is an array. The `colors` array expects values of type `CGColorRef` (which is not an `NSObject` derivative), so we need to use the bridging trick that we first saw in Chapter 2 to keep the compiler happy.

`CAGradientLayer` also has `startPoint` and `endPoint` properties that define the direction of the gradient. These are specified in *unit coordinates*, not points, so the top-left corner of the layer is specified with `{0, 0}` and the bottom-right corner is `{1, 1}`. The resulting gradient is shown in Figure 6.6.

Listing 6.6 A Simple Two-Color Diagonal Gradient

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create gradient layer and add it to our container view
    CAGradientLayer *gradientLayer = [CAGradientLayer layer];
    gradientLayer.frame = self.containerView.bounds;
    [self.containerView.layer addSublayer:gradientLayer];

    //set gradient colors
    gradientLayer.colors = @[(__bridge id)[UIColor redColor].CGColor,
                           (__bridge id)[UIColor blueColor].CGColor];

    //set gradient start and end points
    gradientLayer.startPoint = CGPointMake(0, 0);
    gradientLayer.endPoint = CGPointMake(1, 1);
```

```
}
```

```
@end
```

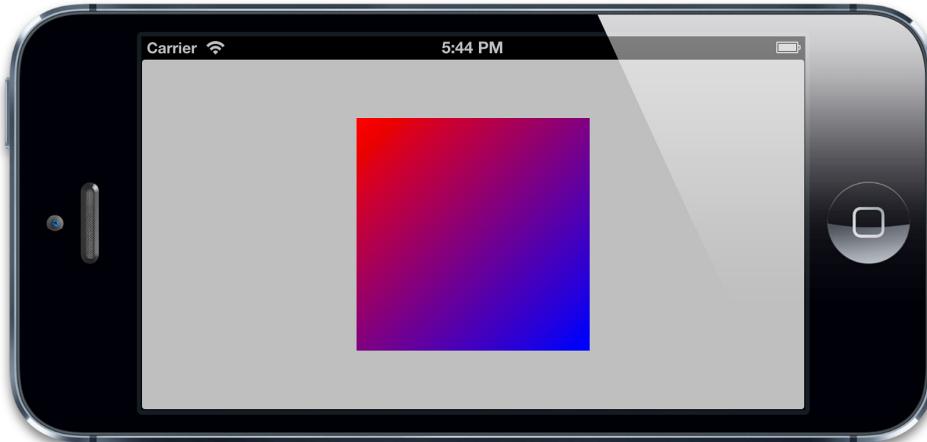


Figure 6.6 A two-color diagonal gradient using `CAGradientLayer`

Multipart Gradients

The `colors` array can contain as many colors as you like, so it is simple to create a multipart gradient such as a rainbow. By default, the colors in the gradient will be evenly spaced, but we can adjust the spacing using the `locations` property.

The `locations` property is an array of floating-point values (boxed as `NSNumber` objects). These values define the positions for each distinct color in the `colors` array, and are specified in unit coordinates, with 0.0 representing the start of the gradient and 1.0 representing the end.

It is not obligatory to supply a `locations` array, but if you do, you must ensure that the number of locations matches the number of colors or you'll get a blank gradient.

Listing 6.7 shows a modified version of the diagonal gradient code from Listing 6.6. We now have a three-part gradient from red to yellow to green. A `locations` array has been specified with the values 0.0, 0.25, and 0.5, which causes the gradient to be squashed up toward the top-left corner of the view (see Figure 6.7).

Listing 6.7 Using the locations Array to Offset a Gradient

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //create gradient layer and add it to our container view
    CAGradientLayer *gradientLayer = [CAGradientLayer layer];
    gradientLayer.frame = self.containerView.bounds;
    [self.containerView.layer addSublayer:gradientLayer];

    //set gradient colors
    gradientLayer.colors = @[(__bridge id)[UIColor redColor].CGColor,
                            (__bridge id)[UIColor yellowColor].CGColor,
                            (__bridge id)[UIColor greenColor].CGColor];

    //set locations
    gradientLayer.locations = @[@0.0, @0.25, @0.5];

    //set gradient start and end points
    gradientLayer.startPoint = CGPointMake(0, 0);
    gradientLayer.endPoint = CGPointMake(1, 1);
}
```

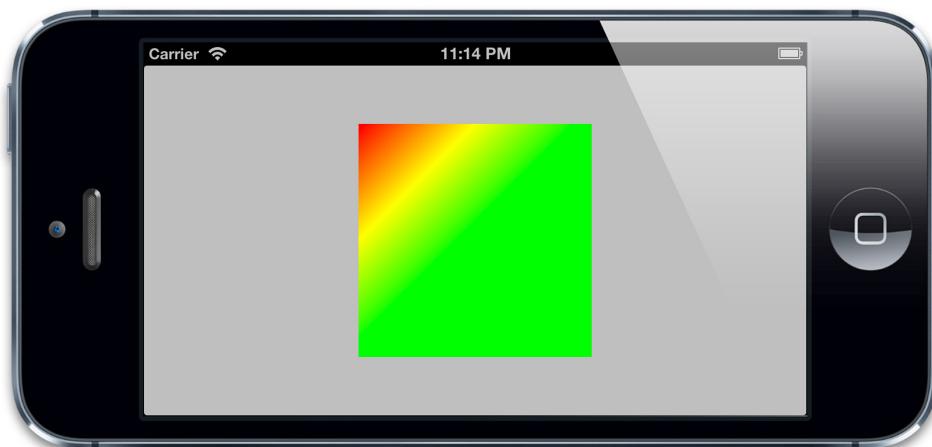


Figure 6.7 A three-color gradient, offset to the top left using the `locations` array

CAReplicatorLayer

The CAReplicatorLayer class is designed to efficiently generate collections of similar layers. It works by drawing one or more duplicate copies of each of its sublayers, applying a different transform to each duplicate. This is probably easier to demonstrate than to explain, so let's construct an example.

Repeating Layers

In Listing 6.8, we create a small white square layer in the middle of the screen, then turn it into a ring of ten layers using CAReplicatorLayer. The instanceCount property specifies how many times the layer should be repeated. The instanceTransform applies a CATransform3D (in this case, a translation and rotation that moves the layer to the next point in the circle).

The transform is applied incrementally, with each instance positioned relative to the previous one. This is why the duplicates don't all end up in the same place. Figure 6.8 shows the result.

Listing 6.8 Repeating Layers Using CAReplicatorLayer

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a replicator layer and add it to our view
    CAReplicatorLayer *replicator = [CAReplicatorLayer layer];
    replicator.frame = self.containerView.bounds;
    [self.containerView.layer addSublayer:replicator];

    //configure the replicator
    replicator.instanceCount = 10;

    //apply a transform for each instance
    CATransform3D transform = CATransform3DIdentity;
    transform = CATransform3DTranslate(transform, 0, 200, 0);
    transform = CATransform3DRotate(transform, M_PI / 5.0, 0, 0, 1);
```

```

transform = CATransform3DTranslate(transform, 0, -200, 0);
replicator.instanceTransform = transform;

//apply a color shift for each instance
replicator.instanceBlueOffset = -0.1;
replicator.instanceGreenOffset = -0.1;

//create a sublayer and place it inside the replicator
CALayer *layer = [CALayer layer];
layer.frame = CGRectMake(100.0f, 100.0f, 100.0f, 100.0f);
layer.backgroundColor = [UIColor whiteColor].CGColor;
[replicator addSublayer:layer];
}

@end

```

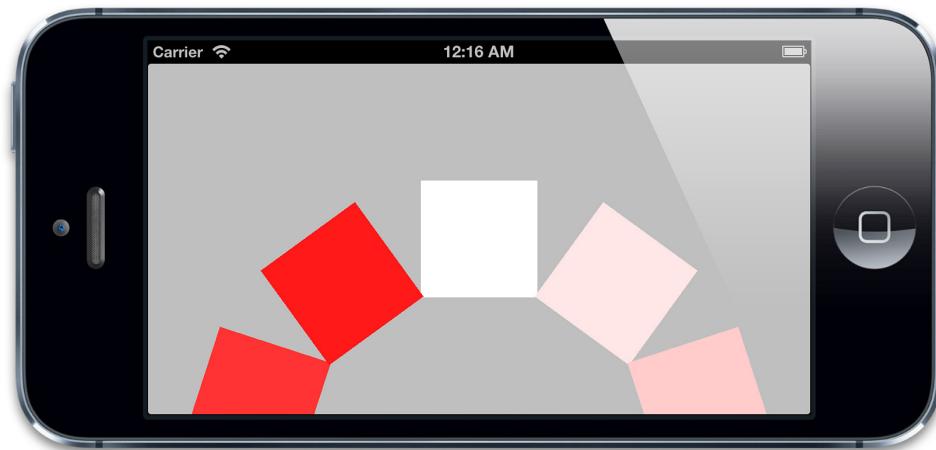


Figure 6.8 A ring of layers, created using CAReplicatorLayer

Note how the color of the layers changes as they are repeated: This was achieved using the `instanceBlueOffset` and `instanceGreenOffset` properties. By decreasing the intensity of the blue and green color components for each repetition, we've caused the layers to color-shift toward red.

This replication effect may look cool, but what are the practical applications? CAReplicatorLayer is useful for special effects such as drawing a contrail behind a missile in a game, or a particle explosion (although iOS 5 introduced CAEmitterLayer,

which is more suited to creating arbitrary particle effects). There is another more practical use however: reflections.

Reflections

By using `CAResuplicatorLayer` to apply a transform with a negative scale factor to a single duplicate layer, you can create a mirror image of the contents of a given view (or an entire view hierarchy), creating a real-time “reflection” effect.

Let’s try implementing this idea in the form of a reusable `UIView` subclass called `ReflectionView` that will automatically generate a reflection of its contents. The code to create this is very simple (see Listing 6.9), and actually using the `ReflectionView` is even simpler; we can just drop an instance of our `ReflectionView` into Interface Builder (see Figure 6.9), and it will generate a reflection of its subviews at runtime without the need to add any setup code to the view controller (see Figure 6.10).

Listing 6.9 Automatically Drawing a Reflection with `CAResuplicatorLayer`

```
#import "ReflectionView.h"
#import <QuartzCore/QuartzCore.h>

@implementation ReflectionView

+ (Class)layerClass
{
    return [CAResuplicatorLayer class];
}

- (void)setUp
{
    //configure replicator
    CAResuplicatorLayer *layer = (CAResuplicatorLayer *)self.layer;
    layer.instanceCount = 2;

    //move reflection instance below original and flip vertically
    CATransform3D transform = CATransform3DIdentity;
    CGFloat verticalOffset = self.bounds.size.height + 2;
    transform = CATransform3DTranslate(transform, 0, verticalOffset, 0);
    transform = CATransform3DScale(transform, 1, -1, 0);
    layer.instanceTransform = transform;

    //reduce alpha of reflection layer
    layer.instanceAlphaOffset = -0.6;
}
```

```

- (id)initWithFrame:(CGRect)frame
{
    //this is called when view is created in code
    if ((self = [super initWithFrame:frame]))
    {
        [self setUp];
    }
    return self;
}

- (void)awakeFromNib
{
    //this is called when view is created from a nib
    [self setUp];
}

@end

```

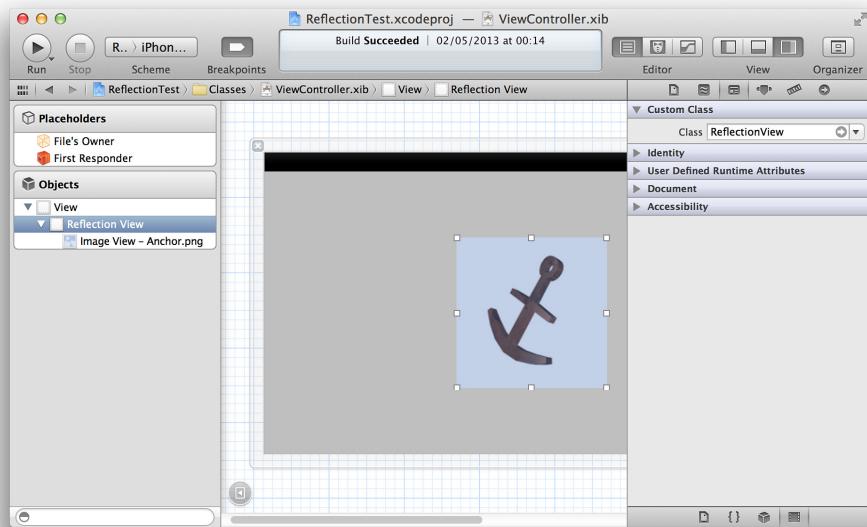


Figure 6.9 Using a `ReflectionView` in Interface Builder

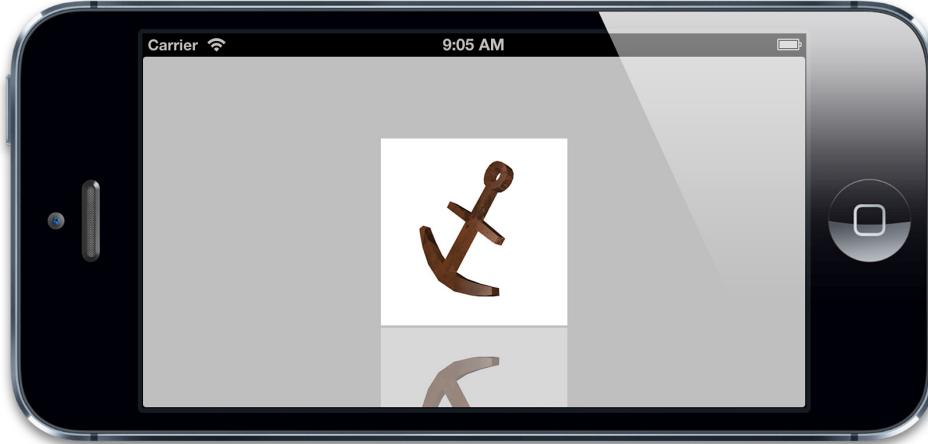


Figure 6.10 The `ReflectionView` automatically creates a reflection at runtime

A more complete, open source implementation of the `ReflectionView` class, complete with an adjustable gradient fade effect (implemented using `CAGradientLayer` and layer masking), can be found at <https://github.com/nicklockwood/ReflectionView>.

CAScrollLayer

For an untransformed layer, the size of a layer's bounds will match the size of its frame. The frame is calculated automatically from the bounds, so changing either property will update the other.

But what if you want to display only a small part of a larger layer? For example, you might have a large image that you want the user to be able to scroll around, or a long list of data or text. In a typical iOS application, you might use a `UITableView` or `UIScrollView`, for this, but what's the equivalent when working with standalone layers?

In Chapter 2, we explored the use of the layer `contentsRect` property, which is a good solution for displaying only a small part of a larger image in a layer. It's not a very good solution if your layer contains sublayers, though, as you would need to manually recalculate and update all the sublayer positions every time you wanted to "scroll" the visible area.

That's where `CAScrollLayer` comes in. `CAScrollLayer` has a `-scrollToPoint:` method that automatically adjusts the origin of the bounds so that the layer contents appear to scroll. Note, however, that that is *all* it does. As discussed earlier, Core Animation does not handle user input, so `CAScrollLayer` is not responsible for turning touch events into a scrolling action, nor does it render scrollbars or

implement any other iOS-specific behavior such as *scroll bouncing* (when a view elastically snaps back into place after scrolling past its bounds).

Let's use a `CAScrollLayer` to create a very basic `UIScrollView` replacement. We'll create a custom `UIView` that uses a `CAScrollLayer` as its backing layer, and then use `UIPanGestureRecognizer` to implement the touch handling. The code is shown in Listing 6.10. Figure 6.11 shows the `ScrollView` being used to pan around a `UIImageView` that is larger than the `ScrollView` frame.

Listing 6.10 Implementing a Scrolling View Using `CAScrollLayer`

```
#import "ScrollView.h"
#import <QuartzCore/QuartzCore.h>

@implementation ScrollView

+ (Class)layerClass
{
    return [CAScrollLayer class];
}

- (void)setUp
{
    //enable clipping
    self.layer.masksToBounds = YES;

    //attach pan gesture recognizer
    UIPanGestureRecognizer *recognizer = nil;
    recognizer = [[UIPanGestureRecognizer alloc]
                  initWithTarget:self action:@selector(pan:)];
    [self addGestureRecognizer:recognizer];
}

- (id)initWithFrame:(CGRect)frame
{
    //this is called when view is created in code
    if ((self = [super initWithFrame:frame]))
    {
        [self setUp];
    }
    return self;
}

- (void)awakeFromNib
{
    //this is called when view is created from a nib
```

```

    [self setUp];
}

- (void)pan:(UIPanGestureRecognizer *)recognizer
{
    //get the offset by subtracting the pan gesture
    //translation from the current bounds origin
    CGPoint offset = self.bounds.origin;
    offset.x -= [recognizer translationInView:self].x;
    offset.y -= [recognizer translationInView:self].y;

    //scroll the layer
    [(CAScrollLayer *)self.layer scrollToPoint:offset];

    //reset the pan gesture translation
    [recognizer setTranslation:CGPointZero inView:self];
}

@end

```

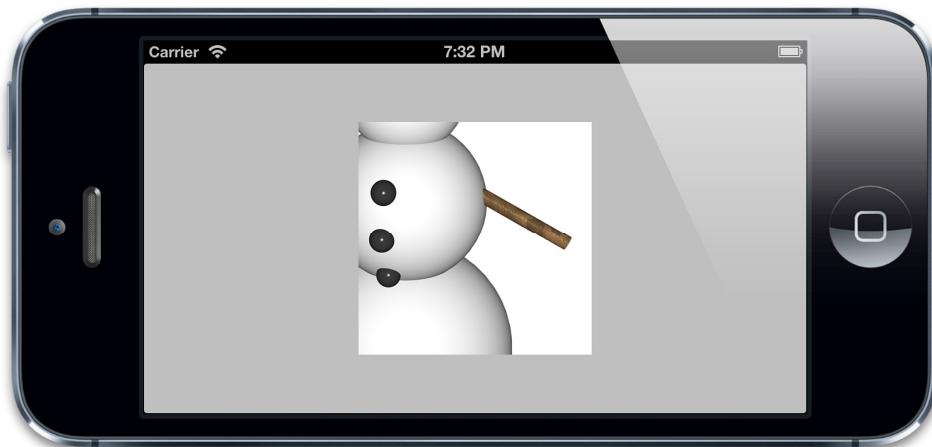


Figure 6.11 Using `CAScrollLayer` to create a makeshift scroll view

Unlike `UIScrollView`, our custom `ScrollView` class doesn't implement any sort of bounds checking. It's quite possible to scroll the layer contents right off the edge of the view and keep going indefinitely. `CAScrollLayer` has no equivalent to the

`UIScrollView` `contentSize` property, and so has no concept of the *total scrollable area*—all that is really happening when you scroll a `CAScrollLayer` is that it adjusts its bounds origin to the value you specify. It doesn't adjust the bounds *size* at all because it doesn't need to; contents can overflow the bounds without consequence.

The astute among you may wonder then what the point of using a `CAScrollLayer` is at all, as you could simply use a regular `CALayer` and adjust the bounds origin yourself. The truth is that there isn't much point really. `UIScrollView` doesn't use a `CAScrollLayer`, in fact, and simply implements scrolling by manipulating the layer bounds directly.

`CAScrollLayer` does have one potentially useful feature, though. If you look in the `CAScrollLayer` header file, you will notice that it includes a category that extends `CALayer` with a couple of extra methods and a property:

```
- (void)scrollPoint:(CGPoint)p;
- (void)scrollRectToVisible:(CGRect)r;
@property(nonatomic) CGRect visibleRect;
```

You might assume from their names that these methods add scrolling functionality to every `CALayer` instance, but in fact they are just utility methods to be used with layers that are placed inside of a `CAScrollLayer`. The `scrollPoint:` method searches up through the layer tree to find the first available `CAScrollLayer` and then scrolls it so that the specified point is visible. The `scrollRectToVisible:` method does the same for a rect. The `visibleRect` property determines the portion of the layer (if any) that is currently visible within the containing `CAScrollLayer`.

It would be relatively straightforward to implement these methods yourself, but `CAScrollLayer` saves you the trouble, and so (just about) justifies its existence when it comes to implementing layer scrolling.

CATiledLayer

Sometimes you will find that you need to draw a *really huge* image. A typical example might be a photograph taken with a high-megapixel camera or a detailed map of the surface of the Earth. iOS applications usually run on quite memory-constrained devices, so loading an entire such image into memory is generally a bad idea. Loading large images may also be quite slow, and doing so in the conventional way (by calling the `UIImage` `-imageNamed:` or `-imageWithContentsOfFile:` methods on the main thread) is likely to freeze your interface for a while, or at least cause animations to stutter.

There is also an upper limit on the size of image that can be efficiently drawn on iOS. All images displayed onscreen have to eventually be converted to an OpenGL texture, and OpenGL has a maximum texture size (usually 2048×2048 or 4096×4096 , depending on the device model). If you try to display an image that is larger than can fit into a single texture, even if the image is already resident in RAM, you can expect to see some very poor

performance indeed, as Core Animation is forced to process the image using the CPU instead of the much faster GPU. (See Chapter 12, “Tuning for Speed,” and Chapter 13, “Efficient Drawing,” for a more detailed explanation of software versus hardware drawing.)

`CATiledLayer` offers a solution for loading large images that solves the performance problems with large images by splitting the image up into multiple small tiles and loading them individually as needed. Let’s test this out with an example.

Tile Cutting

We’ll start with a fairly large image—in this case, a 2048×2048 image of a snowman. To get any benefit from `CATiledLayer`, we’ll need to slice this into several smaller images. You can do this programmatically, but if you load the entire image and then cut it up at runtime, you will lose most of the loading performance advantage that `CATiledLayer` is designed to provide. Ideally, you want to do this as a preprocessing step instead.

Listing 6.11 shows the code for a simple Mac OS command-line app that can cut an image into tiles and save them as individual files for use with a `CATiledLayer`.

Listing 6.11 A Terminal App for Slicing an Image into Tiles

```
#import <AppKit/AppKit.h>

int main(int argc, const char * argv[])
{
    @autoreleasepool
    {
        //handle incorrect arguments
        if (argc < 2)
        {
            NSLog(@"TileCutter arguments: inputFile");
            return 0;
        }

        //input file
        NSString *inputFile = [NSString stringWithCString:argv[1]
                                                encoding:NSUTF8StringEncoding];

        //tile size
        CGFloat tileSize = 256;

        //output path
        NSString *outputPath = [inputFile stringByDeletingPathExtension];

        //load image
        NSImage *image = [[NSImage alloc] initWithContentsOfFile:inputFile];
```

```
NSSize size = [image size];
NSArray *representations = [image representations];
if ([representations count])
{
    NSBitmapImageRep *representation = representations[0];
    size.width = [representation pixelsWide];
    size.height = [representation pixelsHigh];
}
NSRect rect = NSMakeRect(0.0, 0.0, size.width, size.height);
CGImageRef imageRef = [image CGImageForProposedRect:&rect
                                         context:NULL hints:nil];

//calculate rows and columns
NSInteger rows = ceil(size.height / tileSize);
NSInteger cols = ceil(size.width / tileSize);

//generate tiles
for (int y = 0; y < rows; ++y)
{
    for (int x = 0; x < cols; ++x)
    {
        //extract tile image
        CGRect tileRect = CGRectMake(x*tileSize, y*tileSize,
                                     tileSize, tileSize);
        CGImageRef tileImage = CGImageCreateWithImageInRect(imageRef,
                                                          tileRect);

        //convert to jpeg data
        NSBitmapImageRep *imageRep = [[NSBitmapImageRep alloc]
                                      initWithCGImage:titleImage];
        NSData *data = [imageRep representationUsingType:
                        NSJPEGFileType properties:nil];
        CGImageRelease(tileImage);

        //save file
        NSString *path = [outputPath stringByAppendingFormat:
                          @"_%02i_%02i.jpg", x, y];
        [data writeToFile:path atomically:NO];
    }
}
return 0;
}
```

We will use this to convert our 2048×2048 snowman image into 64 individual 256×256 tiles. (256×256 is the default tile size for `CATiledLayer`, although this can be changed using the `tileSize` property.) Our app expects to receive the path to the input image file as the first command-line parameter. We could hard-code this path argument in the build scheme so that we can run it from within Xcode, but that's not very useful if we want to use a different image in future. Instead, we'll build the app and save it somewhere sensible, and then invoke it from the Terminal, like this:

```
> path/to/TileCutterApp path/to/Snowman.jpg
```

The app is very basic, but could easily be extended to support additional arguments such as tile size, or to export images in formats other than JPEG. The result of running it is a sequence of 64 new images, named as follows:

```
Snowman_00_00.jpg  
Snowman_00_01.jpg  
Snowman_00_02.jpg  
...  
Snowman_07_07.jpg
```

Now that we have the tile images, we need to set up an iOS application to consume them. `CATiledLayer` integrates nicely with `UIScrollView`, so for the purposes of this example, we'll place the `CATiledLayer` inside a `UIScrollView`. Other than setting the layer and scrollview bounds to match our total image size, all we really need to do is implement the `-drawLayer:inContext:` method, which is called by `CATiledLayer` whenever it needs to load a new tile image.

Listing 6.12 shows the code, and Figure 6.12 shows the result.

Listing 6.12 A Simple Scrolling `CATiledLayer` Implementation

```
#import "ViewController.h"  
#import <QuartzCore/QuartzCore.h>  
  
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //add the tiled layer  
    CATiledLayer *tileLayer = [CATiledLayer layer];
```

```
tileLayer.frame = CGRectMake(0, 0, 2048, 2048);
tileLayer.delegate = self;
[self.scrollView.layer addSublayer:tileLayer];

//configure the scroll view
self.scrollView.contentSize = tileLayer.frame.size;

//draw layer
[tileLayer setNeedsDisplay];
}

- (void)drawLayer:(CATiledLayer *)layer inContext:(CGContextRef)ctx
{
    //determine tile coordinate
    CGRect bounds = CGContextGetClipBoundingBox(ctx);
    NSInteger x = floor(bounds.origin.x / layer.tileSize.width);
    NSInteger y = floor(bounds.origin.y / layer.tileSize.height);

    //load tile image
    NSString *imageName = [NSString stringWithFormat:
                           @"Snowman_%02i_%02i", x, y];
    NSString *imagePath = [[NSBundle mainBundle] pathForResource:imageName
                           ofType:@"jpg"];
    UIImage *tileImage = [UIImage imageWithContentsOfFile:imagePath];

    //draw tile
    UIGraphicsPushContext(ctx);
    [tileImage drawInRect:bounds];
    UIGraphicsPopContext();
}

@end
```

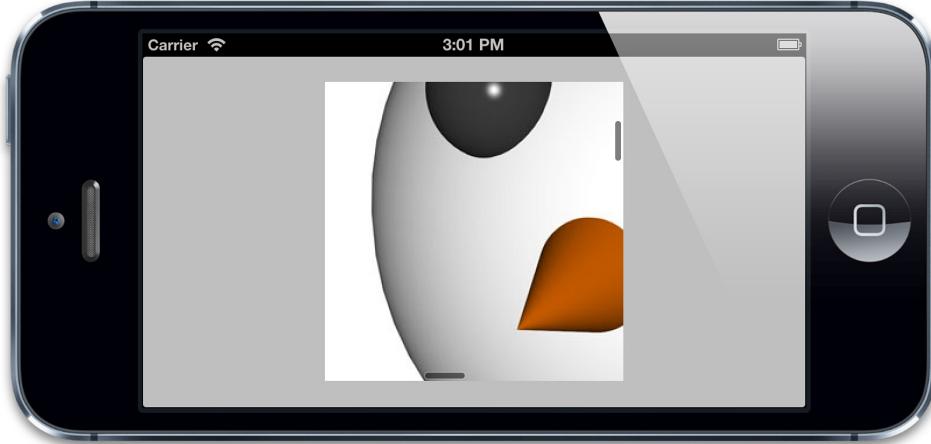


Figure 6.12 Scrolling a `CATiledLayer` using `UIScrollView`

As you scroll around the image, you will notice the tiles fading in as `CATiledLayer` loads them. This is the default behavior of `CATiledLayer`. (You might have seen this effect before in the [pre-iOS 6] Apple Maps app.) You can vary the fade duration or disable it altogether using the `fadeDuration` property.

`CATiledLayer` (unlike most UIKit and Core Animation methods) supports multi-threaded drawing. The `-drawLayer:inContext:` method may be called concurrently on multiple threads at the same time, so be careful to ensure that any drawing code you implement within that method is thread-safe.

Retina Tiles

You might also have noticed that the tile images are not being displayed at Retina resolution. To render the `CATiledLayer` at the device's native resolution, we need to set the layer `contentsScale` to match the `UIScreen` scale, as follows:

```
tileLayer.contentsScale = [UIScreen mainScreen].scale;
```

Interestingly, the `tileSize` is specified in *pixels*, not points, so by increasing the `contentsScale`, we automatically halve the default tile size (it now equates to 128×128 points onscreen instead of 256×256). Therefore, we don't need to update the tile size manually or supply a separate set of tiles at Retina resolution. We *will* need to adjust the tile rendering code slightly to accommodate the change in scale, however:

```
//determine tile coordinate
CGRect bounds = CGContextGetClipBoundingBox(ctx);
CGFloat scale = [UIScreen mainScreen].scale;
```

```
NSInteger x = floor(bounds.origin.x / layer.tileSize.width * scale);  
NSInteger y = floor(bounds.origin.y / layer.tileSize.height * scale);
```

Correcting the scale in this way also means that our snowman image will be rendered at half the size on Retina devices (at a total size of only 1024×1024 points instead of 2048×2048 as before). This usually doesn't matter for the types of images normally displayed with `CATiledLayer` (such photographs and maps, which are designed to be zoomed and viewed at various scales), but it's worth bearing in mind.

CAEmitterLayer

In iOS 5, Apple introduced a new `CALayer` subclass called `CAEmitterLayer`. `CAEmitterLayer` is a high-performance particle engine designed to let you create real-time particle animations such as smoke, fire, rain, and so on.

`CAEmitterLayer` acts as a container for a collection of `CAEmitterCell` instances that define a particle effect. You will create one or more `CAEmitterCell` objects as *templates* for the different particle types, and the `CAEmitterLayer` is responsible for instantiating a stream of particles based on these templates.

A `CAEmitterCell` is similar to a `CALayer`: It has a `contents` property that can be set using a `CGImage`, as well as dozens of configurable properties that control the appearance and behavior of the particles. We won't attempt to describe each and every property in detail, but they are well documented in the `CAEmitterCell` class header file.

Let's try an example: We'll create a fiery explosion effect by emitting particles in a circle with varying velocity and alpha. Listing 6.13 contains the code for generating the explosion. You can see the result in Figure 6.13.

Listing 6.13 Creating an Explosion Effect with `CAEmitterLayer`

```
#import "ViewController.h"  
#import <QuartzCore/QuartzCore.h>  
  
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];
```

```
//create particle emitter layer
CAEmitterLayer *emitter = [CAEmitterLayer layer];
emitter.frame = self.containerView.bounds;
[self.containerView.layer addSublayer:emitter];

//configure emitter
emitter.renderMode = kCAEmitterLayerAdditive;
emitter.emitterPosition = CGPointMake(emitter.frame.size.width / 2.0,
                                      emitter.frame.size.height / 2.0);

//create a particle template
CAEmitterCell *cell = [[CAEmitterCell alloc] init];
cell.contents = (_bridge id)[UIImage imageNamed:@"Spark.png"].CGImage;
cell.birthRate = 150;
cell.lifetime = 5.0;
cell.color = [UIColor colorWithRed:1
                           green:0.5
                             blue:0.1
                            alpha:1.0].CGColor;
cell.alphaSpeed = -0.4;
cell.velocity = 50;
cell.velocityRange = 50;
cell.emissionRange = M_PI * 2.0;

//add particle template to emitter
emitter.emitterCells = @[cell];
}

@end
```

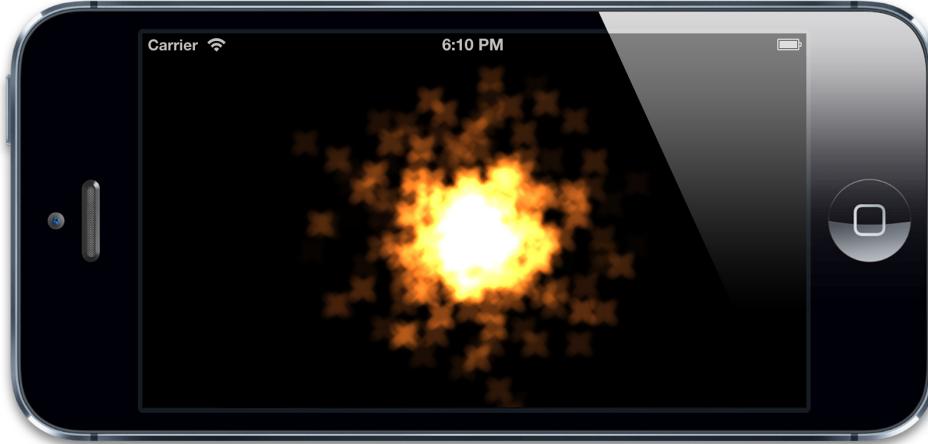


Figure 6.13 A fiery explosion effect

The properties of `CAEmitterCell` generally break down into three categories:

- A starting value for a particular attribute of the particle. For example, the `color` property specifies a blend color that will be multiplied by the colors in the contents image. In our explosion project, we've set this to an orangey color.
- A range by which a value will vary from particle to particle. For example, the `emissionRange` property is set to 2π in our project, indicating that particles can be emitted in any direction within a 360-degree radius. By specifying a smaller value, we could create a conical funnel for our particles.
- A change over time for a particular value. For example, in the explosion project, we set `alphaSpeed` to `-0.4`, meaning that the `alpha` value of the particle will reduce by `0.4` every second, creating a fadeout effect for the particles as they travel away from the emitter.

The properties of the `CAEmitterLayer` itself control the position and general shape of the entire particle system. Some properties such as `birthRate`, `lifetime`, and `velocity` duplicate values that are specified on the `CAEmitterCell`. These act as multipliers so that you can speed up or amplify the entire particle system using a single value. Other notable properties include the following:

- `preservesDepth`, which controls whether a 3D particle system is flattened into a single layer (the default) or can intermingle with other layers in the 3D space of its container layer.

- `renderMode`, which controls how the particle images are blended visually. You may have noted in our example that we set this to `kCAEmitterLayerAdditive`, which has the effect of combining the brightness of overlapping particles so that they appear to glow. If we were to leave this as the default value of `kCAEmitterLayerUnordered`, the result would be a lot less pleasing (see Figure 6.14).



Figure 6.14 The fire particles with additive blending disabled

CAEAGLLayer

When it comes to high-performance graphics on iOS, the last word is OpenGL. It should probably also be the *last resort*, at least for nongaming applications, because it's phenomenally complicated to use compared to the Core Animation and UIKit frameworks.

OpenGL provides the underpinning for Core Animation. It is a low-level C API that communicates directly with the graphics hardware on the iPhone and iPad, with minimal abstraction. OpenGL has no notion of a hierarchy of objects or layers; it simply deals with triangles. In OpenGL everything is made of triangles that are positioned in 3D space and have colors and textures associated with them. This approach is extremely flexible and powerful, but it's a lot of work to replicate something like the iOS user interface from scratch using OpenGL.

To get good performance with Core Animation, you need to determine what sort of content you are drawing (vector shapes, bitmaps, particles, text, and so on) and then select an appropriate layer type to represent that content. Only some types of content have been

optimized in Core Animation; so if the thing you want to draw isn't a good match for any of the standard layer classes, you will struggle to achieve good performance.

Because OpenGL makes no assumptions about your content, it can be blazingly fast. With OpenGL, you can draw *anything you like* provided you know how to write the necessary geometry and shader logic. This makes it a popular choice for games (where Core Animation's limited repertoire of optimized content types doesn't always meet the requirements), but it's generally overkill for applications with a conventional interface.

In iOS 5, Apple introduced a new framework called GLKit that takes away some of the complexity of setting up an OpenGL drawing context by providing a `UIView` subclass called `GLKView` that handles most of the setup and drawing for you. Prior to that it was necessary to do all the low-level configuration of the various OpenGL drawing buffers yourself using `CAEAGLLayer`, which is a `CALayer` subclass designed for displaying arbitrary OpenGL graphics.

It is rare that you will need to manually set up a `CAEAGLLayer` any more (as opposed to just using a `GLKView`), but let's give it a go for old time's sake. Specifically, we'll set up an *OpenGL ES 2.0* context, which is the standard for all modern iOS devices.

Although it is possible to do this entirely without using GLKit, it involves a lot of additional work in the form of setting up *vertex* and *fragment shaders*, which are self-contained programs that are written in a C-like language called GLSL, and are loaded at runtime into the graphics hardware. Writing GLSL code is not really related to the task of setting up an `EAGLLayer`, so we'll use the `GLKBaseEffect` class to abstract away the shader logic for us. Everything else, we'll do the old-fashioned way.

Before we begin, you need to add the GLKit and OpenGL ES frameworks to your project. You should then be able to implement the code in Listing 6.14, which does the bare minimum necessary to set up a `CAEAGLLayer` with an OpenGL ES 2.0 drawing context and render a colored triangle (see Figure 6.15).

Listing 6.14 Drawing a Triangle Using `CAEAGLLayer`

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>
#import <GLKit/GLKit.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *glView;
@property (nonatomic, strong) EAGLContext *glContext;
@property (nonatomic, strong) CAEAGLLayer *glLayer;
@property (nonatomic, assign) GLuint framebuffer;
@property (nonatomic, assign) GLuint colorRenderbuffer;
@property (nonatomic, assign) GLint framebufferWidth;
@property (nonatomic, assign) GLint framebufferHeight;
@property (nonatomic, strong) GLKBaseEffect *effect;
```

```

@end

@implementation ViewController

- (void)setUpBuffers
{
    //set up frame buffer
    glGenFramebuffers(1, &_framebuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);

    //set up color render buffer
    glGenRenderbuffers(1, &_colorRenderbuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                             GL_RENDERBUFFER, _colorRenderbuffer);
    [self.context renderbufferStorage:GL_RENDERBUFFER
        fromDrawable:self.gllayer];
    glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH,
                                &_framebufferWidth);
    glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT,
                                &_framebufferHeight);

    //check success
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    {
        NSLog(@"Failed to make complete framebuffer object: %i",
              glCheckFramebufferStatus(GL_FRAMEBUFFER));
    }
}

- (void)tearDownBuffers
{
    if (_framebuffer)
    {
        //delete framebuffer
        glDeleteFramebuffers(1, &_framebuffer);
        _framebuffer = 0;
    }

    if (_colorRenderbuffer)
    {
        //delete color render buffer
        glDeleteRenderbuffers(1, &_colorRenderbuffer);
        _colorRenderbuffer = 0;
    }
}

```

```
}

- (void)drawFrame
{
    //bind framebuffer & set viewport
    glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);
    glViewport(0, 0, _framebufferWidth, _framebufferHeight);

    //bind shader program
    [self.effect prepareToDraw];

    //clear the screen
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    //set up vertices
    GLfloat vertices[] =
    {
        -0.5f, -0.5f, -1.0f,
        0.0f, 0.5f, -1.0f,
        0.5f, -0.5f, -1.0f,
    };

    //set up colors
    GLfloat colors[] =
    {
        0.0f, 0.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
    };

    //draw triangle
    glEnableVertexAttribArray(GLKVertexAttribPosition);
    glEnableVertexAttribArray(GLKVertexAttribAttribColor);
    glVertexAttribPointer(GLKVertexAttribPosition,
                         3, GL_FLOAT, GL_FALSE, 0, vertices);
    glVertexAttribPointer(GLKVertexAttribAttribColor,
                         4, GL_FLOAT, GL_FALSE, 0, colors);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    //present render buffer
    glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
    [self.glContext presentRenderbuffer:GL_RENDERBUFFER];
}

- (void)viewDidLoad
```

```
{  
    [super viewDidLoad];  
  
    //set up context  
    self.glContext = [[EAGLContext alloc] initWithAPI:  
                      kEAGLRenderingAPIOpenGL2];  
    [EAGLContext setCurrentContext:self.glContext];  
  
    //set up layer  
    self.glLayer = [CAEAGLLayer layer];  
    self.glLayer.frame = self.glView.bounds;  
    [self.glView.layer addSublayer:self.glLayer];  
    self.glLayer.drawableProperties = @ {kEAGLDrawablePropertyRetainedBacking:  
                                         @NO,  
                                         kEAGLDrawablePropertyColorFormat:  
                                         kEAGLColorFormatRGBA8};  
  
    //set up base effect  
    self.effect = [[GLKBaseEffect alloc] init];  
  
    //set up buffers  
    [self setUpBuffers];  
  
    //draw frame  
    [self drawFrame];  
}  
  
- (void)viewDidUnload  
{  
    [self tearDownBuffers];  
    [super viewDidUnload];  
}  
  
- (void)dealloc  
{  
    [self tearDownBuffers];  
    [EAGLContext setCurrentContext:nil];  
}  
  
@end
```

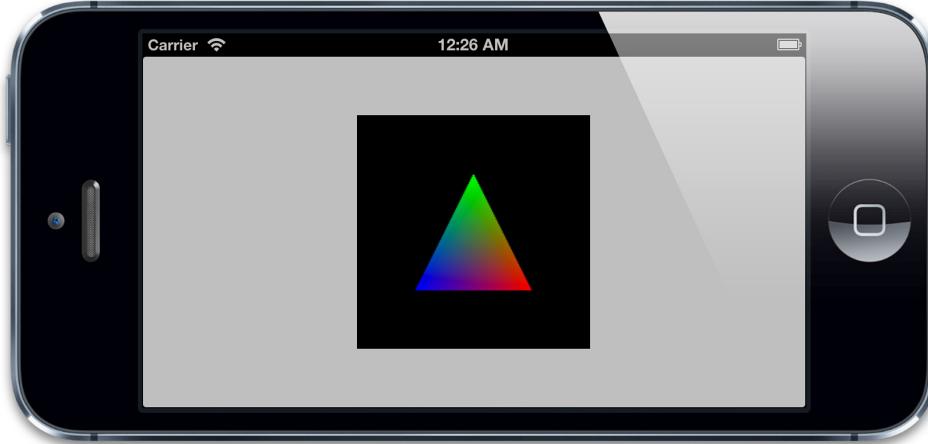


Figure 6.15 A single triangle rendered in a `CAEAGLLayer` using OpenGL

In a real OpenGL application, we would probably want to call the `-drawFrame` method 60 times per second using an `NSTimer` or `CADisplayLink` (see Chapter 11, “Timer-Based Animation”), and we would separate out geometry generation from drawing so that we aren’t regenerating our triangle’s vertices every frame (and also so that we can draw something other than a single triangle), but this should be enough to demonstrate the principle.

AVPlayerLayer

The last layer type we will look at is `AVPlayerLayer`. Although not part of the Core Animation framework (the AV prefix is a bit of a giveaway), `AVPlayerLayer` is an example of another framework (in this case, AVFoundation) tightly integrating with Core Animation by providing a `CALayer` subclass to display a custom content type.

`AVPlayerLayer` is used for playing video on iOS. It is the underlying implementation used by high-level APIs such as `MPMoviePlayer`, and provides lower-level control over the display of video. Usage of `AVPlayerLayer` is actually pretty straightforward: You can either create a layer with a video player already attached using the `+playerLayerWithPlayer:` method, or you can create the layer first and attach an `AVPlayer` instance using the `player` property.

Before we begin, we need to add the AVFoundation framework to our project, because it’s not included in the default project template. After you’ve done that, see Listing 6.15 for the code to create a simple movie player. Figure 6.16 shows the movie player in action.

Listing 6.15 Playing a Video with AVPlayerLayer

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>
#import <AVFoundation/AVFoundation.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //get video URL
    NSURL *URL = [[NSBundle mainBundle] URLForResource:@"Ship"
                                                withExtension:@"mp4"];

    //create player and player layer
    AVPlayer *player = [AVPlayer playerWithURL:URL];
    AVPlayerLayer *playerLayer = [AVPlayerLayer playerLayerWithPlayer:player];

    //set player layer frame and attach it to our view
    playerLayer.frame = self.containerView.bounds;
    [self.containerView.layer addSublayer:playerLayer];

    //play the video
    [player play];
}

@end
```



Figure 6.16 Still-frame from a video playing inside an `AVPlayerLayer`

Although we're creating the `AVPlayerLayer` programmatically, we are still adding it to a container view instead of directly inside the main view for the controller. This is so that we can use ordinary autolayout constraints to center the layer; otherwise, we would have to reposition it programmatically every time the device is rotated due to Core Animation not supporting autoresizing or autolayout (see Chapter 3, “Layer Geometry,” for details).

Of course, because `AVPlayerLayer` is a subclass of `CALayer`, it inherits all of its cool features. We're not restricted to playing our video in a simple rectangle; with the few extra lines of code in Listing 6.16, we can rotate our video in 3D and add rounded corners, a colored border, mask, drop shadow, and so on (see Figure 6.17).

Listing 6.16 Adding a Transform, Border, and Corner Radius to the Video

```
- (void)viewDidLoad
{
    ...
    //set player layer frame and attach it to our view
    playerLayer.frame = self.containerView.bounds;
    [self.containerView.layer addSublayer:playerLayer];

    //transform layer
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / 500.0;
    transform = CATransform3DRotate(transform, M_PI_4, 1, 1, 0);
    playerLayer.transform = transform;
```

```
//add rounded corners and border
playerLayer.masksToBounds = YES;
playerLayer.cornerRadius = 20.0;
playerLayer.borderColor = [UIColor redColor].CGColor;
playerLayer.borderWidth = 5.0;

//play the video
[player play];
}
```

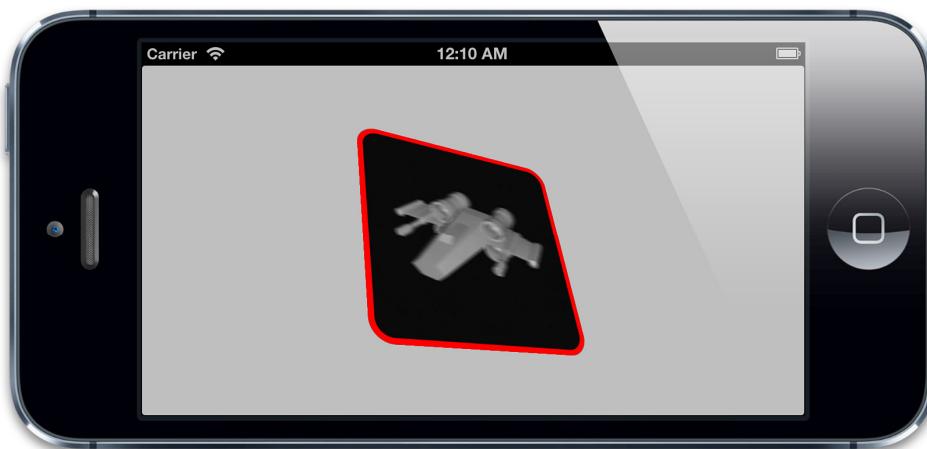


Figure 6.17 AVPlayerLayer rotated in 3D and displaying a border and corner radius

Summary

This chapter provided an overview of the many specialized layer types and the effects that can be achieved by using them. We have only scratched the surface in many cases; classes such as CATiledLayer or CAEmitterLayer could fill a chapter on their own. However, the key point to remember is that CALayer is a jack-of-all-trades, and is not optimized for every possible drawing scenario. To get the best performance out of Core Animation, you need to choose the right tool for the job, and hopefully you have been inspired to dig deeper into the various CALayer subclasses and their capabilities.

We touched on animation a little bit in this chapter with the `CAEmitterLayer` and `AVPlayerLayer` classes. In Part II, we dive into animation properly, starting with *implicit animations*.

This page intentionally left blank



Setting Things in Motion

- 7** Implicit Animations
- 8** Explicit Animations
- 9** Layer Time
- 10** Easing
- 11** Timer-Based Animation

This page intentionally left blank

Implicit Animations

Do what I mean, not what I say.

Edna Krabappel, *The Simpsons*

Part I covered just about everything that Core Animation can do, apart from *animation*. Animation is a pretty significant part of the Core Animation framework. In this chapter, we take a look at how it works. Specifically, we explore *implicit animations*, which are animations that the framework performs automatically (unless you tell it not to).

Transactions

Core Animation is built on the assumption that everything you do onscreen will (or at least *may*) be animated. Animation is not something that you *enable* in Core Animation. Animations have to be explicitly *turned off*; otherwise, they happen all the time.

Whenever you change an animatable property of a `CALayer`, the change is not reflected immediately onscreen. Instead, the layer property animates smoothly from the previous value to the new one. You don't have to do anything to make this happen; it's the default behavior.

This might seem a bit too good to be true, so let's demonstrate it with an example: We'll take the blue square project from Chapter 1, "The Layer Tree," and add a button that will set the layer to a random color. Listing 7.1 shows the code for this. Tap the button and you will see that the color changes smoothly instead of jumping to its new value (see Figure 7.1).

Listing 7.1 Randomizing the Layer Color

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *layerView;  
@property (nonatomic, weak) IBOutlet CALayer *colorLayer;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //create sublayer  
    self.colorLayer = [CALayer layer];  
    self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);  
    self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;  
  
    //add it to our view  
    [self.layerView.layer addSublayer:self.colorLayer];  
}  
  
- (IBAction)changeColor  
{  
    //randomize the layer background color  
    CGFloat red = arc4random() / (CGFloat)INT_MAX;  
    CGFloat green = arc4random() / (CGFloat)INT_MAX;  
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;  
    self.colorLayer.backgroundColor = [UIColor colorWithRed:red  
                                              green:green  
                                              blue:blue  
                                              alpha:1.0].CGColor;  
}  
  
@end
```

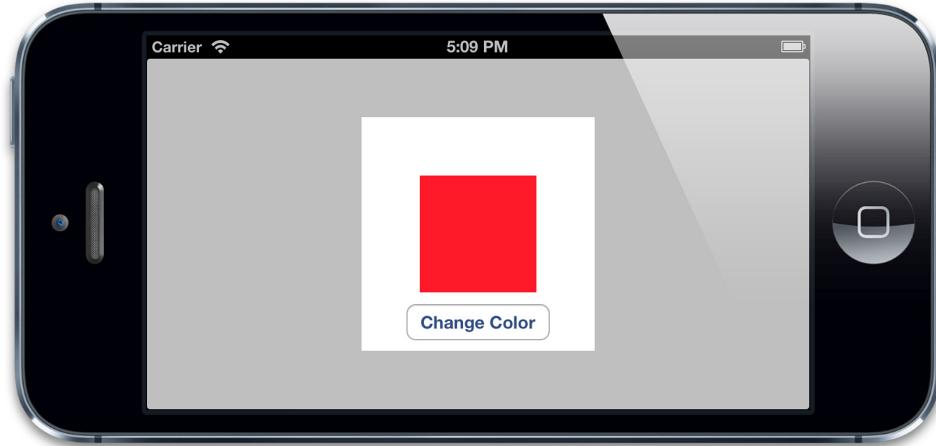


Figure 7.1 Adding a button to change the layer color

This kind of animation is known as *implicit* animation. It is implicit because we are not specifying what kind of animation we want to happen; we just change a property, and Core Animation decides how and when to animate it. Core Animation also supports *explicit* animation, which is covered in the next chapter.

When you change a property, how does Core Animation determine the type and duration of the animation that it will perform? The duration of the animation is specified by the settings for the current *transaction*, and the animation type is controlled by *layer actions*.

Transactions are the mechanism that Core Animation uses to encapsulate a particular set of property animations. Any animatable layer properties that are changed within a given transaction will not change immediately, but instead will begin to animate to their new value as soon as that transaction is *committed*.

Transactions are managed using the `CATransaction` class. The `CATransaction` class has a peculiar design in that it does not represent a single transaction as you might expect from the name, but rather it manages a stack of transactions without giving you direct access to them. `CATransaction` has no properties or instance methods, and you can't create a transaction using `+alloc` and `-init` as normal. Instead, you use the class methods `+begin` and `+commit` to push a new transaction onto the stack or pop the current one, respectively.

Any layer property change that can be animated will be added to the topmost transaction in the stack. You can set the duration of the current transaction's animations by using the `+setAnimationDuration:` method, or you can find out the current duration using the `+animationDuration` method. (The default is 0.25 seconds.)

Core Animation automatically begins a new transaction with each iteration of the *run loop*. (The run loop is where iOS gathers user input, handles any outstanding timer or network events, and then eventually redraws the screen.) Even if you do not explicitly begin a transaction using `[CATransaction begin]`, any property changes that you make within a given run loop iteration will be grouped together and then animated over a 0.25-second period.

Armed with this knowledge, we can easily change the duration of our color animation. It would be sufficient to change the animation duration of the current (default) transaction by using the `+setAnimationDuration:` method, but we will start a new transaction first so that changing the duration doesn't have any unexpected side effects. Changing the duration of the current transaction might possibly affect other animations that are incidentally happening at the same time (such as screen rotation), so it is always a good idea to push a new transaction explicitly before adjusting the animation settings.

Listing 7.2 shows the modified code. If you run the app, you will notice that the color fade happens much more slowly than before.

Listing 7.2 Controlling Animation Duration Using CATransaction

```
- (IBAction)changeColor
{
    //begin a new transaction
    [CATransaction begin];

    //set the animation duration to 1 second
    [CATransaction setAnimationDuration:1.0];

    //randomize the layer background color
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    self.colorLayer.backgroundColor = [UIColor colorWithRed:red
                                                green:green
                                                blue:blue
                                               alpha:1.0].CGColor;

    //commit the transaction
    [CATransaction commit];
}
```

If you've ever done any animation work using the `UIView` animation methods, this pattern should look familiar. `UIView` has two methods, `+beginAnimations:context:` and `+commitAnimations`, that work in a similar way to the `+begin` and `+commit`

methods on `CATransaction`. Any view or layer properties you change between calls to `+beginAnimations:context:` and `+commitAnimations` will be animated automatically because what those `UIView` animation methods are actually doing is setting up a `CATransaction`.

In iOS 4, Apple added a new block-based animation method to `UIView`, `+animateWithDuration:animations:`. This is syntactically a bit cleaner than having separate methods to begin and end a block of property animations, but really it's just doing the same thing behind the scenes.

The `CATransaction` `+begin` and `+commit` methods are called internally by the `+animateWithDuration:animations:` method, with the body of the animations block executed in between them so that any property changes that you make inside the block will be encapsulated by the transaction. This has the benefit of avoiding any risk of mismatched `+begin` and `+commit` calls due to developer error.

Completion Blocks

The `UIView` block-based animation allows you to supply a completion block to be called when the animation has finished. This same feature is available when using the `CATransaction` interface by calling the `+setCompletionBlock:` method. Let's adapt our example again so that it performs an action when the color change has completed. We'll attach a completion block and use it to trigger a second animation that spins the layer 90 degrees each time the color has changed. Listing 7.3 shows the code, and Figure 7.2 shows the result.

Listing 7.3 Adding a Callback When the Color Animation Completes

```
- (IBAction)changeColor
{
    //begin a new transaction
    [CATransaction begin];

    //set the animation duration to 1 second
    [CATransaction setAnimationDuration:1.0];

    //add the spin animation on completion
    [CATransaction setCompletionBlock:^{
        //rotate the layer 90 degrees
        CGAffineTransform transform = self.colorLayer.affineTransform;
        transform = CGAffineTransformRotate(transform, M_PI_2);
        self.colorLayer.affineTransform = transform;
    }];
}
```

```

//randomize the layer background color
CGFloat red = arc4random() / (CGFloat)INT_MAX;
CGFloat green = arc4random() / (CGFloat)INT_MAX;
CGFloat blue = arc4random() / (CGFloat)INT_MAX;
self.colorLayer.backgroundColor = [UIColor colorWithRed:red
                                                 green:green
                                                 blue:blue
                                                 alpha:1.0].CGColor;

//commit the transaction
[CATransaction commit];
}

```

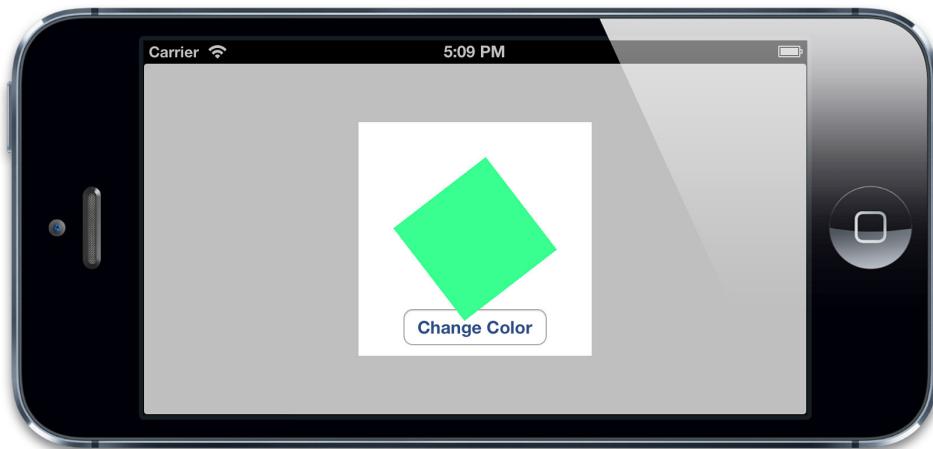


Figure 7.2 A rotation animation applied after the color fade has finished

Notice that our rotation animation is much faster than our color fade animation. That's because the completion block that applies the rotation animation is executed *after* the color fade animation's transaction has been committed and popped off the stack. It is, therefore, using the default transaction, with the default animation duration of 0.25 seconds.

Layer Actions

Now let's try an experiment: Instead of animating a standalone sublayer, we'll try directly animating the backing layer of our view. Listing 7.4 shows an adapted version of the code

from Listing 7.2 that removes the `colorLayer` and sets the `layerView` backing layer's background color directly.

Listing 7.4 Setting the Property of the Backing Layer Directly

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set the color of our layerView backing layer directly
    self.layerView.layer.backgroundColor = [UIColor blueColor];
}

- (IBAction)changeColor
{
    //begin a new transaction
    [CATransaction begin];

    //set the animation duration to 1 second
    [CATransaction setAnimationDuration:1.0];

    //randomize the layer background color
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    self.layerView.layer.backgroundColor = [UIColor colorWithRed:red
                                                green:green
                                                blue:blue
                                               alpha:1.0].CGColor;

    //commit the transaction
    [CATransaction commit];
}
```

If you run the project, you'll notice that the color snaps to its new value immediately when the button is pressed instead of animating smoothly as before. What's going on? The implicit animation seems to have been disabled for the `UIView` backing layer.

Come to think of it, we'd probably have noticed if `UIView` properties always animated automatically whenever we modified them. So, if UIKit is built on top of Core Animation (which always animates everything by default), how come implicit animations are *disabled* by default in UIKit?

We know that Core Animation normally animates any property change of a `CALayer` (provided it *can* be animated) and that `UIView` somehow turns this behavior off for its backing layer. To understand how it does that, we need to understand how implicit animations are implemented in the first place.

The animations that `CALayer` automatically applies when properties are changed are called *actions*. When a property of a `CALayer` is modified, it calls its `-actionForKey:` method, passing the name of the property in question. What happens next is quite nicely documented in the header file for `CALayer`, but it essentially boils down to this:

1. The layer first checks whether it has a delegate and if the delegate implements the `-actionForLayer:forKey` method specified in the `CALayerDelegate` protocol. If it does, it will call it and return the result.
2. If there is no delegate, or the delegate does not implement `-actionForLayer:forKey`, the layer checks in its `actions` dictionary, which contains a mapping of property names to actions.
3. If the `actions` dictionary does not contain an entry for the property in question, the layer searches inside its `style` dictionary hierarchy for any actions that match the property name.
4. Finally, if it fails to find a suitable action anywhere in the `style` hierarchy, the layer will fall back to calling the `-defaultActionForKey:` method, which defines standard actions for known properties.

The result of this exhaustive search will be that `-actionForKey:` either returns `nil` (in which case, no animation will take place and the property value will change immediately) or an object that conforms to the `CAAction` protocol, which `CALayer` will then use to animate between the previous and current property values.

And that explains how UIKit disables implicit animations: Every `UIView` acts as the delegate for its backing layer and provides an implementation for the `-actionForLayer:forKey` method. When not inside an animation block, `UIView` returns `nil` for all layer actions, but within the scope of an animation block it returns non-`nil` values. We can demonstrate this with a simple experiment (see Listing 7.5).

Listing 7.5 Testing `UIView`'s `actionForLayer:forKey:` Implementation

```
@interface ViewController ()
```

```

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //test layer action when outside of animation block
    NSLog(@"Outside: %@", [self.layerView actionForLayer:self.layerView.layer
                                                forKey:@"backgroundColor"]);

    //begin animation block
    [UIView beginAnimations:nil context:nil];

    //test layer action when inside of animation block
    NSLog(@"Inside: %@", [self.layerView actionForLayer:self.layerView.layer
                                                forKey:@"backgroundColor"]);

    //end animation block
    [UIView commitAnimations];
}

@end

```

When we run the project, we see this in the console:

```

$ LayerTest[21215:c07] Outside: <null>
$ LayerTest[21215:c07] Inside: <CABasicAnimation: 0x757f090>

```

So as predicted, `UIView` is disabling implicit animation when properties are changed outside of an animation block by returning `nil` for the property actions. The action that it returns when animation is enabled depends on the property type, but in this case, it's a `CABasicAnimation`. (You'll learn what that is in Chapter 8, "Explicit Animations".)

Returning `nil` for the action is not the only way to disable implicit animations; `CATransaction` has a method called `+setDisableActions:` that can be used to enable or disable implicit animation for all properties simultaneously. If we modify the code in Listing 7.2 by adding the following line after `[CATransaction begin]`, it will prevent any animations from taking place:

```
[CATransaction setDisableActions:YES];
```

So to recap, we've learned the following:

- `UIView` backing layers do not have implicit animation enabled. The only ways to animate the properties of a backing layer are to use the `UIView` animation methods (instead of relying on `CATransition`), to subclass `UIView` itself and override the `-actionForLayer:forKey:` method, or to create an explicit animation (see Chapter 8 for details).
- For hosted (that is, nonbacking) layers, we can control the animation that will be selected for an implicit property animation by either implementing the `-actionForLayer:forKey:` layer delegate method, or by providing an `actions` dictionary.

Let's specify a different action for our color fade example. We'll modify Listing 7.1 by setting a custom `actions` dictionary for `colorLayer`. We could implement this using the delegate instead, but the `actions` dictionary approach requires marginally less code. So how can we create a suitable action object?

Actions are usually specified using an *explicit* animation object that will be called *implicitly* by Core Animation when it is needed. The animation we are using here is a *push transition*, which is implemented with an instance of `CATransition` (see Listing 7.6).

Transitions are explained fully in Chapter 8, but suffice to say for now that `CATransition` conforms to the `CAAction` protocol, and can therefore be used as a layer action. The result is pretty cool; whenever we change our layer color, the new value slides in from the left instead of using the default crossfade effect (see Figure 7.3).

Listing 7.6 Implementing a Custom Action

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) IBOutlet CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;
}
```

```

//add a custom action
CATransition *transition = [CATransition animation];
transition.type = kCATransitionPush;
transition.subtype = kCATransitionFromLeft;
self.colorLayer.actions = @{@"backgroundColor": transition};

//add it to our view
[self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
    //randomize the layer background color
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    self.colorLayer.backgroundColor = [UIColor colorWithRed:red
                                                green:green
                                                blue:blue
                                               alpha:1.0].CGColor;
}

@end

```

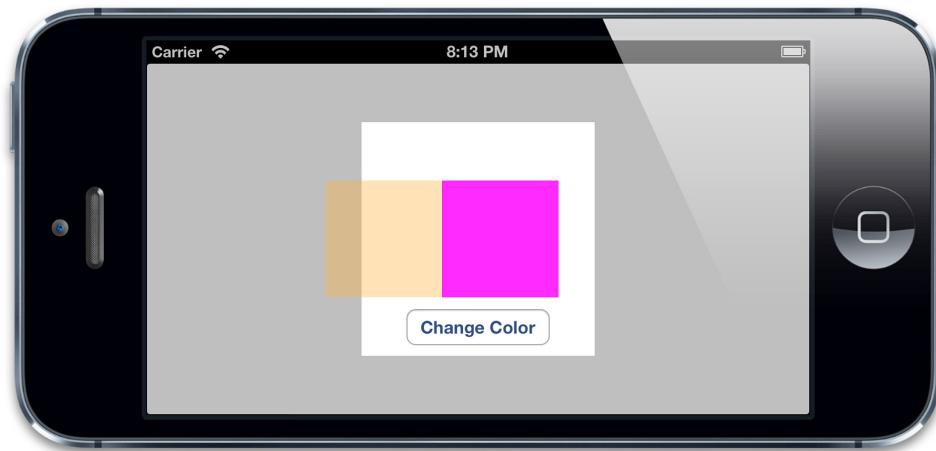


Figure 7.3 The color value animation implemented using a push transition

Presentation Versus Model

The behavior of properties on a `CALayer` is unusual, in that changing a layer property does not have an immediate effect, but gradually updates over time. How does that work?

When you change a property of a layer, the property value *is* actually updated immediately (if you try to read it, you'll find that the value is whatever you just set it to), but that change is not reflected onscreen. That's because the property you set doesn't adjust the appearance of the layer directly; instead, it defines the appearance that the layer is *going to have* when that property's animation has completed.

When you set the properties of a `CALayer`, you are really defining a *model* for how you want the display to look at the end of the current transaction. Core Animation then acts as a *controller* and takes responsibility for updating the *view* state of these properties onscreen based on the layer actions and transaction settings.

What we are talking about is effectively the *MVC pattern in miniature*. `CALayer` is a visual class that you would normally associate with the user interface (aka *view*) part of the MVC (Model-View-Controller) pattern, but in the context of the user interface itself, `CALayer` behaves more like a model for how the view is going to look when all animations have completed. In fact, in Apple's own documentation, the layer tree is sometimes referred to as the *model* layer tree.

In iOS, the screen is redrawn 60 times per second. If the animation duration is longer than one 60th of a second, Core Animation is therefore required to recompose the layer onto the screen multiple times between when you set the new value for an animatable property and when that new value is eventually reflected onscreen. This implies that `CALayer` must somehow maintain a record of the current *display* value of the property in addition to its "actual" value (the value that you've set it to).

The display values of each layer's properties are stored in a separate layer called the *presentation layer*, which is accessed via the `-presentationLayer` method. The presentation layer is essentially a duplicate of the model layer, except that its property values always represent the *current appearance* at any given point in time. In other words, you can access a property of the presentation layer to find out the current *onscreen* value of the equivalent model layer property (see Figure 7.4).

We mentioned in Chapter 1 that in addition to the layer tree there is a *presentation tree*. The presentation tree is the tree formed by the presentation layers of all the layers in the layer tree. Note that the presentation layer is only created when a layer is first *committed* (that is, when it's first displayed onscreen), so attempting to call `-presentationLayer` before then will return `nil`.

You may notice that there is also a `-modelLayer` method. Calling `-modelLayer` on a presentation layer will return the underlying `CALayer` that it is presenting. Calling `-modelLayer` on a regular layer just returns `-self`. (We already established that ordinary layers are in fact a type of model.)

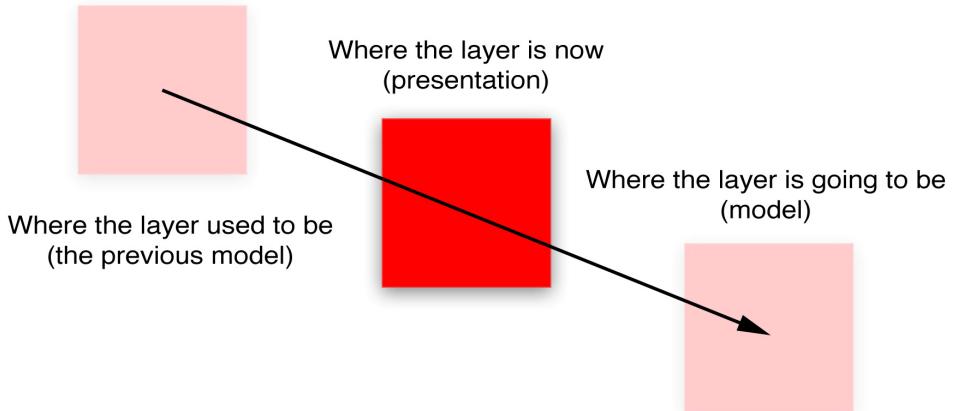


Figure 7.4 How model relates to presentation for a moving layer

Most of the time, you do not need to access the presentation layer directly; you can just interact with the properties of the model layer and let Core Animation take care of updating the display. Two cases where the presentation layer *does* become useful are for synchronizing animations and for handling user interaction:

- If you are implementing timer-based animations (see Chapter 11, “Timer-Based Animation”) in addition to ordinary transaction-based animations, it can be useful to be able to find out exactly where a given layer appears to be onscreen at a given point in time so that you can position other elements to correctly align with the animation.
- If you want your animated layers to respond to user input, and you are using the `-hitTest:` method (see Chapter 3, “Layer Geometry”) to determine whether a given layer is being touched, it makes sense to call `-hitTest:` against the *presentation* layer rather than the *model* layer because that represents the layer’s position as the user currently sees it, not as it will be when the current animation has finished.

We can demonstrate the latter case with a simple example (see Listing 7.7). In the example, tapping anywhere onscreen animates the layer to the position you’ve touched. Tapping on the layer itself sets its color to a random value. We determine whether the tap is inside the layer by calling the `-hitTest:` method on the layer’s presentation layer.

If you modify the code so that `-hitTest:` is called directly on the `colorLayer` instead of its presentation layer, you will see that it doesn’t work correctly when the layer is moving. Instead of tapping the layer, you now have to tap the location that the layer is moving toward for it to register the hit (which is why we used the presentation layer for hit testing originally).

Listing 7.7 Using presentationLayer to Determine Current Layer Position

```
@interface ViewController : UIViewController

@property (nonatomic, strong) CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a red layer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(0, 0, 100, 100);
    self.colorLayer.position = CGPointMake(self.view.bounds.size.width / 2,
                                           self.view.bounds.size.height / 2);
    self.colorLayer.backgroundColor = [UIColor redColor].CGColor;
    [self.view.layer addSublayer:self.colorLayer];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the touch point
    CGPoint point = [[touches anyObject] locationInView:self.view];

    //check if we've tapped the moving layer
    if ([self.colorLayer.presentationLayer hitTest:point])
    {
        //randomize the layer background color
        CGFloat red = arc4random() / (CGFloat)INT_MAX;
        CGFloat green = arc4random() / (CGFloat)INT_MAX;
        CGFloat blue = arc4random() / (CGFloat)INT_MAX;
        self.colorLayer.backgroundColor = [UIColor colorWithRed:red
                                                       green:green
                                                       blue:blue
                                                       alpha:1.0].CGColor;
    }
    else
    {
        //otherwise (slowly) move the layer to new position
        [CATransaction begin];
        [CATransaction setAnimationDuration:4.0];
        self.colorLayer.position = point;
    }
}
```

```
[CATransaction commit];  
}  
}  
  
@end
```

Summary

This chapter covered implicit animations and the mechanism that Core Animation uses to select an appropriate animation action for a given property. You also learned how UIKit utilizes Core Animation's implicit animation mechanism to power its own *explicit* system, where animations are disabled by default and only enabled when requested. Finally, you learned about the presentation and model layers and how they allow Core Animation to keep track of both where a layer is *and* where it's going to be.

In the next chapter, we look at the *explicit* animation types provided by Core Animation, which can be used either to directly animate layer properties, or to override the default layer actions.

This page intentionally left blank

Explicit Animations

If you want something done right, do it yourself.

Charles-Guillaume Étienne

The previous chapter introduced the concept of implicit animations. Implicit animations are a straightforward way to create animated user interfaces on iOS, and they are the mechanism on which UIKit's own animation methods are based, but they are not a completely general-purpose animation solution. In this chapter, we will look at *explicit* animations, which allow us to specify custom animations for particular properties or create nonlinear animations, such as a movement along an arbitrary curve.

Property Animations

The first type of explicit animation we will look at is the *property animation*. Property animations target a single property of a layer and specify a target value or range of values for that property to animate between. Property animations come in two flavors: *basic* and *keyframe*.

Basic Animations

An animation is a change that happens over time, and the simplest form of change is when one value changes to another, which is exactly what `CABasicAnimation` is designed to model.

`CABasicAnimation` is a concrete subclass of the abstract `CAPropertyAnimation` class, which in turn is a subclass of `CAAnimation`, the abstract base class for all animation types supported by Core Animation. As an abstract class, `CAAnimation` doesn't actually do very much on its own. It provides a timing function (as explained in Chapter 10, “Easing”), a

delegate (used to get feedback about the animation state), and a `removedOnCompletion` flag, used to indicate whether the animation should automatically be released after it has finished (this defaults to YES, which prevents your application's memory footprint from spiraling out of control). `CAAnimation` also implements a number of protocols, including `CAACTION` (allowing any `CAAnimation` subclass to be supplied as a layer action) and `CAMediaTiming` (which is explained in detail in Chapter 9, "Layer Time").

`CAPropertyAnimation` acts upon a single property, specified by the animation's `keyPath` value. A `CAAnimation` is always applied to a specific `CALayer`, so the `keyPath` is relative to that layer. The fact that this is a key *path* (a sequence of dot-delimited keys that can point to an arbitrarily nested object within a hierarchy) rather than just a property name is interesting because it means that animations can be applied not only to properties of the layer itself, but to properties of its member objects, and even *virtual* properties (more on this later).

`CABasicAnimation` extends `CAPropertyAnimation` with three additional attributes:

```
id fromValue  
id toValue  
id byValue
```

These are fairly self-explanatory: `fromValue` represents the value of the property at the start of the animation; `toValue` represents its value at the end of the animation; `byValue` represents the relative amount by which the value changes during the animation.

By combining these three attributes, you can specify a value change in various different ways. The type is defined as `id` (as opposed to something more specific) because property animations can be used with a number of different property types, including numeric values, vectors, transform matrices, and even colors and images.

A property of type `id` can contain any `NSObject` derivative, but often you will want to animate property types that do not actually inherit from `NSObject`, which means that you will need to either wrap the value in an object (known as *boxing*) or cast it to an object (known as *toll-free bridging*), which is possible for certain Core Foundation types that *behave* like Objective-C classes, even though they aren't. It's not always obvious how to convert the expected data type to an `id`-compatible value, but the common cases are listed in Table 8.1.

Table 8.1 Boxing Primitive Values for Use in a `CAPropertyAnimation`

Type	Object Type	Code Example
<code>CGFloat</code>	<code>NSNumber</code>	<code>id obj = @(float);</code>
<code>CGPoint</code>	<code>NSValue</code>	<code>id obj = [NSValue valueWithCGPoint:point];</code>
<code>CGSize</code>	<code>NSValue</code>	<code>id obj = [NSValue valueWithCGSize:size];</code>
<code>CGRect</code>	<code>NSValue</code>	<code>id obj = [NSValue valueWithCGRect:rect];</code>

```

CATransform3D NSValue           id obj = [NSValue
                                         valueWithCATransform3D:transform];
CGImageRef      id             id obj = (__bridge id)imageRef;
CGColorRef     id             id obj = (__bridge id)colorRef;

```

The `fromValue`, `toValue`, and `byValue` properties can be used in various combinations, but you should not specify all three at once because that could result in a contradiction. For example, if you were to specify a `fromValue` of 2, a `toValue` of 4, and a `byValue` of 3, Core Animation would not know whether the final value should be 4 (as specified by `toValue`) or 5 (`fromValue + byValue`). The exact rules around how these properties can be used are neatly documented in the `CABasicAnimation` header file, so we won't repeat them here. In general, you will only need to specify either the `toValue` or `byValue`; the other values can be determined automatically from context.

Let's try an example: We'll modify our color fade animation from Chapter 7, "Implicit Animations," to use an explicit `CABasicAnimation` instead of an implicit one. Listing 8.1 shows the code.

Listing 8.1 Setting the Layer Background Color with `CABasicAnimation`

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, strong) IBOutlet CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //Create sublayer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;

    //Add it to our view
    [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
    //Create a new random color
}

```

```

CGFloat red = arc4random() / (CGFloat)INT_MAX;
CGFloat green = arc4random() / (CGFloat)INT_MAX;
CGFloat blue = arc4random() / (CGFloat)INT_MAX;
UIColor *color = [UIColor colorWithRed:red
                                green:green
                                blue:blue
                               alpha:1.0];

//create a basic animation
CABasicAnimation *animation = [CABasicAnimation animation];
animation.keyPath = @"backgroundColor";
animation.toValue = (_bridge id)color.CGColor;

//apply animation to layer
[self.colorLayer addAnimation:animation forKey:nil];
}

@end

```

When we run the example, it doesn't work as expected. Tapping the button causes the layer to animate to a new color, but it then immediately snaps back to its original value.

The reason for this is that animations do not modify the layer's *model*, only its *presentation* (see Chapter 7). Once the animation finishes and is removed from the layer, the layer reverts back to the appearance defined by its model properties. We never changed the `backgroundColor` property, so the layer returns to its original color.

When we were using implicit animation before, the underlying action was implemented using a `CABasicAnimation` exactly like the one we have just used. (You might recall that in Chapter 7, we logged the result of the `-actionForLayer:forKey:` delegate method and saw that the action type was a `CABasicAnimation`.) However, in that case, we triggered the animation by setting the property. Now we are performing the same animation directly, but we aren't setting the property any more (hence the snap-back problem).

Assigning our animation as a layer action (and then simply triggering the animation by changing the property value) is by far the easiest approach to keeping property values and animation states in sync, but assuming that we cannot do that for some reason (usually because the layer we need to animate is a `UIView` backing layer), we have two choices for when we can update the property value: immediately before the animation starts or immediately after it finishes.

Updating the property before the animation has started is the simpler of those options, but it means that we cannot take advantage of the implicit `fromValue`, so we will need to manually set the `fromValue` in our animation to match the current value in the layer.

Taking that into account, if we insert the following two lines between where we create our animation and where we add it to the layer, it should get rid of the snap-back:

```
animation.fromValue = (_bridge id)self.colorLayer.backgroundColor;
self.colorLayer.backgroundColor = color.CGColor;
```

That works, but is potentially unreliable. We should really derive the `fromValue` from the *presentation* layer (if it exists) rather than the model layer, in case there is already an animation in progress. Also, because the layer in this case is *not* a backing layer, we should disable implicit animations using a `CATransaction` before setting the property, or the default layer action may interfere with our explicit animation. (In practice, the explicit animation always seems to override the implicit one, but this behavior is not documented, so it's better to be safe than sorry.)

If we make those changes, we end up with the following:

```
CALayer *layer = self.colorLayer.presentationLayer ?: self.colorLayer;
animation.fromValue = (_bridge id)layer.backgroundColor;
[CATransaction begin];
[CATransaction setDisableActions:YES];
self.colorLayer.backgroundColor = color.CGColor;
[CATransaction commit];
```

That's quite a lot of code to have to add to each and every animation. Fortunately, we can derive this information automatically from the `CABasicAnimation` object itself, so we can create a reusable method. Listing 8.2 shows a modified version of our first example that includes a method for applying a `CABasicAnimation` without needing to repeat this boilerplate code each time.

Listing 8.2 A Reusable Method for Fixing Animation Snap-Back

```
- (void)applyBasicAnimation:(CABasicAnimation *)animation
                      toLayer:(CALayer *)layer
{
    //set the from value (using presentation layer if available)
    animation.fromValue = [layer.presentationLayer ?: layer
                           valueForKeyPath:animation.keyPath];

    //update the property in advance
    //note: this approach will only work if toValue != nil
    [CATransaction begin];
    [CATransaction setDisableActions:YES];
    [layer setValue:animation.toValue forKeyPath:animation.keyPath];
    [CATransaction commit];

    //apply animation to layer
    [layer addAnimation:animation forKey:nil];
}
```

```

- (IBAction)changeColor
{
    //create a new random color
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    UIColor *color = [UIColor colorWithRed:red
                                    green:green
                                      blue:blue
                                         alpha:1.0];

    //create a basic animation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"backgroundColor";
    animation.toValue = (__bridge id)color.CGColor;

    //apply animation without snap-back
    [self applyBasicAnimation:animation toLayer:self.colorLayer];
}

```

This simple implementation only handles animations with a `toValue`, not a `byValue`, but it's a good start toward a general solution. You could package it up as a category method on `CALayer` to make it more convenient and reusable.

This might all seem like a lot of trouble to solve such a seemingly simple problem, but the alternative is considerably *more* complex. If we don't update the target property *before* we begin the animation, we cannot update it until after the animation has fully completed or we will cancel the `CABasicAnimation` in progress. That means we need to update the property at the exact point when the animation has finished, but before it gets removed from the layer and the property snaps back to its original value. How can we determine that point?

CAAnimationDelegate

When using implicit animations in Chapter 7, we were able to detect when an animation finished by using the `CATransaction` completion block. That approach isn't available when using explicit animations, however, because the animation isn't associated with a transaction.

To find out when an explicit animation has finished, we need to make use of the animation's `delegate` property, which is an object conforming to the `CAAnimationDelegate` protocol.

`CAAnimationDelegate` is an ad hoc protocol, so you won't find a `CAAnimationDelegate` `@protocol` defined in any header file, but you can find the supported methods in the `CAAnimation` header or in Apple's developer documentation. In this case, we use the `-animationDidStop:finished:` method to update our layer's `backgroundColor` immediately after the animation has finished.

We need to set up a new transaction and disable layer actions when we update the property; otherwise, the animation will occur twice—once due to our explicit `CABasicAnimation`, and then again afterward due to the implicit animation action for that property. See Listing 8.3 for the complete implementation.

Listing 8.3 Fixing the Background Color Value Once Animation Completes

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;

    //add it to our view
    [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
    //create a new random color
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    UIColor *color = [UIColor colorWithRed:red
                                    green:green
                                    blue:blue
                                   alpha:1.0];

    //create a basic animation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"backgroundColor";
    animation.toValue = (__bridge id)color.CGColor;
    animation.delegate = self;

    //apply animation to layer
}
```

```

        [self.colorLayer addAnimation:animation forKey:nil];
    }

- (void)animationDidStop:(CABasicAnimation *)anim finished:(BOOL)flag
{
    //set the backgroundColor property to match animation toValue
    [CATransaction begin];
    [CATransaction setDisableActions:YES];
    self.colorLayer.backgroundColor = (__bridge CGColorRef)anim.toValue;
    [CATransaction commit];
}

@end

```

The problem with `CAAnimation` using a delegate pattern instead of a completion block is that it makes it quite awkward when you have multiple animations or animated layers to keep track of. When creating animations in a view controller, you would usually use the controller itself as the animation delegate (as we did in Listing 8.3), but since all the animations will be calling the same delegate method, you need some way to determine which completion call relates to which layer.

Consider the clock from Chapter 3, “Layer Geometry”; we originally implemented the clock without animation by simply updating the angle of the hands every second. It would look nicer if the hands animated to their new position realistically.

We can’t animate the hands using implicit animation because the hands are represented by `UIView` instances, and implicit animation is disabled for their backing layers. We could animate them easily using `UIView` animation methods, but there is a benefit to using an explicit property animation if we want more control over the animation timing (more on this in Chapter 10). Animating those hands using `CABasicAnimation` is potentially quite complex because we would need to detect which hand the animation relates to in the `-animationDidStop:finished:` method (so we can set its finishing position).

The animation itself is passed as a parameter to the delegate method. You might be thinking that you can store the animations as properties in the controller and compare them with the parameter in the delegate method, but this won’t work because the animation returned by the delegate is an immutable copy of the original, not the same object.

When we attached the animations to our layers using `-addAnimation:forKey:`, there was a `key` parameter that we’ve so far always set to `nil`. The key is an `NSString` that is used to uniquely identify the animation if you later want to retrieve it using the layer’s `-animationForKey:` method. The keys for all animations currently attached to a layer can be retrieved using the `animationKeys` property. If we were to use a unique key for each animation, we could loop through the animation keys for each animated layer and compare

the result of calling `-animationForKey:` with the animation object passed to our delegate method. It's not exactly an elegant solution, though.

Fortunately, there is an easier way. Like all `NSObject` subclasses, `CAAnimation` conforms to the KVC (Key-Value Coding) ad hoc protocol, which allows you to set and get properties by name using the `-setValue:forKey:` and `-valueForKey:` methods. But `CAAnimation` has an unusual feature: It acts like an `NSDictionary`, allowing you to set arbitrary key/value pairs even if they do not match up to any of the declared properties of the animation class that you are using.

This means that you can tag an animation with additional data for your own use. In this case, we will attach the clock hand `UIImageView` to the animation, so we can easily determine which view each animation relates to. We can then use this information in the delegate method to update the correct hand (see Listing 8.4).

Listing 8.4 Using KVC to Tag an Animation with Additional Data

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *hourHand;
@property (nonatomic, weak) IBOutlet UIImageView *minuteHand;
@property (nonatomic, weak) IBOutlet UIImageView *secondHand;
@property (nonatomic, weak) NSTimer *timer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //adjust anchor points
    self.secondHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.minuteHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.hourHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);

    //start timer
    self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                target:self
                                              selector:@selector(tick)
                                              userInfo:nil
                                             repeats:YES];

    //set initial hand positions
    [self updateHandsAnimated:NO];
}
```

```

}

- (void)tick
{
    [self updateHandsAnimated:YES];
}

- (void)updateHandsAnimated:(BOOL)animated
{
    //convert time to hours, minutes and seconds
    NSCalendar *calendar =
        [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];

   NSUInteger units = NSHourCalendarUnit |
        NSMinuteCalendarUnit |
        NSSecondCalendarUnit;

    NSDateComponents *components = [calendar components:units
                                                fromDate:[NSDate date]];

    //calculate hour hand angle
    CGFloat hourAngle = (components.hour / 12.0) * M_PI * 2.0;

    //calculate minute hand angle
    CGFloat minuteAngle = (components.minute / 60.0) * M_PI * 2.0;

    //calculate second hand angle
    CGFloat secondAngle = (components.second / 60.0) * M_PI * 2.0;

    //rotate hands
    [self setAngle:hourAngle forHand:self.hourHand animated:animated];
    [self setAngle:minuteAngle forHand:self.minuteHand animated:animated];
    [self setAngle:secondAngle forHand:self.secondHand animated:animated];
}

- (void)setAngle:(CGFloat)angle
            forHand:(UIView *)handView
            animated:(BOOL)animated
{
    //generate transform
    CATransform3D transform = CATransform3DMakeRotation(angle, 0, 0, 1);

    if (animated)
    {
        //create transform animation
        CABasicAnimation *animation = [CABasicAnimation animation];

```

```

        animation.keyPath = @"transform";
        animation.toValue = [NSValue valueWithCATransform3D:transform];
        animation.duration = 0.5;
        animation.delegate = self;
        [animation setValue:handView forKey:@"handView"];
        [handView.layer addAnimation:animation forKey:nil];
    }
}
else
{
    //set transform directly
    handView.layer.transform = transform;
}
}

- (void)animationDidStop:(CABasicAnimation *)anim finished:(BOOL)flag
{
    //set final position for hand view
    UIView *handView = [anim valueForKey:@"handView"];
    handView.layer.transform = [anim.toValue CATransform3DValue];
}

@end

```

We've successfully identified when each layer has finished animating and updated its transform to the correct value. So far, so good.

Unfortunately, even after taking those steps, we have another problem. Listing 8.4 works fine on the simulator, but if we run it on an iOS device, we can see our clock hand snap back to its original value briefly before our `-animationDidStop:finished:` delegate method is called. The same thing happens with the layer color in Listing 8.3.

The problem is that although the callback method is called after the animation has finished, there is no guarantee that it will be called before the property has been reset to its pre-animation state. This is a good example of why you should always test animation code on a device, not just on the simulator.

We can work around this by using a property called `fillMode`, which we explore in the next chapter, but the lesson here is that setting the `animated` property to its final value immediately *before* applying the animation is a much simpler approach than trying to update it after the animation has finished.

Keyframe Animations

`CABasicAnimation` is interesting in that it shows us the underlying mechanism behind most of the implicit animations on iOS, but adding a `CABasicAnimation` to a layer explicitly is a lot of work for little benefit when there are simpler ways to achieve the same effect (either using implicit animations for hosted layers, or `UIView` animation for views and backing layers).

`CAKeyframeAnimation`, however, is considerably more powerful and has no equivalent interface exposed in UIKit. `CAKeyframeAnimation` is, like `CABasicAnimation`, a subclass of `CAPropertyAnimation`. It still operates on a single property, but unlike `CABasicAnimation` it is not limited to just a single start and end value, and instead can be given an arbitrary sequence of values to animate between.

The term *keyframe* originates from traditional animation, where a lead animator would draw only the frames where something significant happens (the *key frames*), and then the less highly skilled artists would draw the frames in between (which could be easily inferred from the keyframes). The same principle applies with `CAKeyframeAnimation`: You provide the significant frames, and Core Animation fills in the gaps using a process called *interpolation*.

We can demonstrate this using our colored layer from the earlier example. We'll set up an array of colors and play them back with a single command using a keyframe animation (see Listing 8.5).

Listing 8.5 Applying a Sequence of Colors Using `CAKeyframeAnimation`

```
- (IBAction)changeColor
{
    //create a keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"backgroundColor";
    animation.duration = 2.0;
    animation.values = @[
        (__bridge id)[UIColor blueColor].CGColor,
        (__bridge id)[UIColor redColor].CGColor,
        (__bridge id)[UIColor greenColor].CGColor,
        (__bridge id)[UIColor blueColor].CGColor
    ];

    //apply animation to layer
    [self.colorLayer addAnimation:animation forKey:nil];
}
```

Note that we've specified blue as both the start and end color in the sequence. That is necessary because `CAKeyframeAnimation` does not have an option to automatically use the current value as the first frame (as we were doing with the `CABasicAnimation` by leaving `fromValue` as `nil`). The animation will immediately jump to the first keyframe value when it begins, and immediately revert to the original property value once it finishes, so for a smooth animation, we need both the start and end keyframes to match the current value of the property.

Of course, it's possible to create animations that end on a different value than they begin. In that case, we would need to manually update the property value to match the last keyframe before we trigger the animation, just as we discussed earlier.

We've increased the duration of our animation from the default of 0.25 seconds to 2 seconds using the `duration` property so that the animation doesn't take place too quickly to follow. If you run the animation, you'll see that the layer cycles through the colors, but the effect seems a bit... *strange*. The reason for this is that the animation runs at a *constant pace*. It does not slow down as it transitions through each color, and this results in a slightly surreal effect. To make the animation appear more natural, we will need to adjust the *easing*, which will be explained in Chapter 10.

Supplying an array of values makes sense for animating something like a color change, but it's not a very intuitive way to describe motion in general. `CAKeyframeAnimation` has an alternative way to specify an animation, by using a `CGPath`. The `path` property allows you to define a motion sequence in an intuitive way by using Core Graphics functions to *draw* your animation.

To demonstrate this, let's animate a spaceship image moving along a simple curve. To create the path, we'll use a *cubic Bézier curve*, which is a special type of curve that is defined using a start and end point with two additional *control points* to guide the shape. It's possible to create such a path purely using C-based Core Graphics drawing commands, but it's easier to do this using the high-level `UIBezierPath` class provided by UIKit.

Although it's not actually necessary for the animation, we're going to draw the curve onscreen using a `CAShapeLayer`. This makes it easier to visualize what our animation is going to do. After we've drawn the `CGPath`, we'll use it to create a `CAKeyframeAnimation`, then apply it to our spaceship. Listing 8.6 shows the code, and Figure 8.1 shows the result.

Listing 8.6 Animating a Layer Along a Cubic Bézier Curve

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad
```

```
{  
    [super viewDidLoad];  
  
    //create a path  
    UIBezierPath *bezierPath = [[UIBezierPath alloc] init];  
    [bezierPath moveToPoint:CGPointMake(0, 150)];  
    [bezierPath addCurveToPoint:CGPointMake(300, 150)  
        controlPoint1:CGPointMake(75, 0)  
        controlPoint2:CGPointMake(225, 300)];  
  
    //draw the path using a CAShapeLayer  
    CAShapeLayer *pathLayer = [CAShapeLayer layer];  
    pathLayer.path = bezierPath.CGPath;  
    pathLayer.fillColor = [UIColor clearColor].CGColor;  
    pathLayer.strokeColor = [UIColor redColor].CGColor;  
    pathLayer.lineWidth = 3.0f;  
    [self.containerView.layer addSublayer:pathLayer];  
  
    //add the ship  
    CALayer *shipLayer = [CALayer layer];  
    shipLayer.frame = CGRectMake(0, 0, 64, 64);  
    shipLayer.position = CGPointMake(0, 150);  
    shipLayer.contents = (__bridge id)[UIImage imageNamed:  
        @"Ship.png"].CGImage;  
    [self.containerView.layer addSublayer:shipLayer];  
  
    //create the keyframe animation  
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];  
    animation.keyPath = @"position";  
    animation.duration = 4.0;  
    animation.path = bezierPath.CGPath;  
    [shipLayer addAnimation:animation forKey:nil];  
}  
  
@end
```

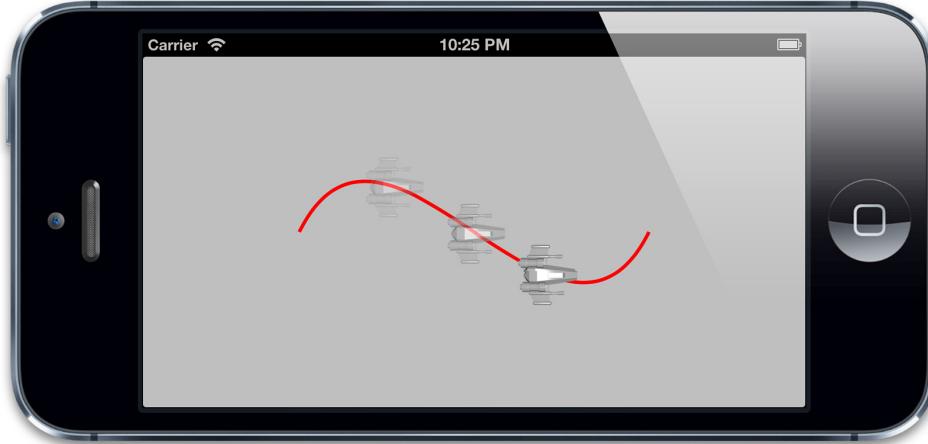


Figure 8.1 The spaceship image layer moving along a Bézier curve

If you run the example, you may notice that the ship animation looks a bit unrealistic because it's always pointing directly to the right as it moves rather than turning to match the tangent of the curve. You could animate the orientation of the ship as it moves by adjusting its `affineTransform`, but that would be tricky to synchronize with the other animation.

Fortunately, Apple anticipated this scenario, and added a property to `CAKeyframeAnimation` called `rotationMode`. Set `rotationMode` to the constant value `kCAAnimationRotateAuto` (see Listing 8.7), and the layer will automatically rotate to follow the tangent of the curve as it animates (see Figure 8.2).

Listing 8.7 Automatically Aligning Layer to Curve with `rotationMode`

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a path
    ...

    //create the keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"position";
    animation.duration = 4.0;
    animation.path = bezierPath.CGPath;
    animation.rotationMode = kCAAnimationRotateAuto;
```

```
[shipLayer addAnimation:animation forKey:nil];  
}
```

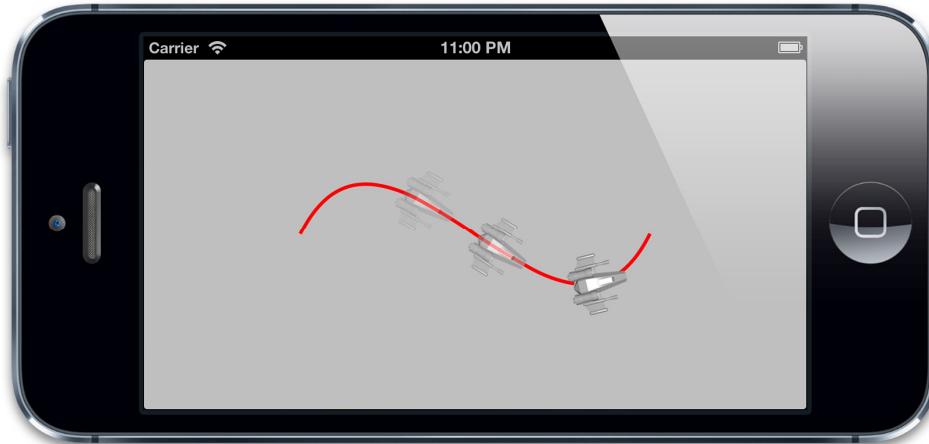


Figure 8.2 The spaceship layer rotating to match the tangent of the curve

Virtual Properties

We mentioned earlier that the fact that property animations work on key *paths* instead of keys means that we can animate subproperties *and even virtual properties*. But what is a *virtual* property?

Consider a rotation animation: If we wanted to animate a rotating object, we'd have to animate the `transform` because there is no explicit angle/orientation property on a `CALayer`. We might do that as shown in Listing 8.8.

Listing 8.8 Rotating a Layer by Animating the `transform` Property

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
  
@end  
  
@implementation ViewController
```

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add the ship
    CALayer *shipLayer = [CALayer layer];
    shipLayer.frame = CGRectMake(0, 0, 128, 128);
    shipLayer.position = CGPointMake(150, 150);
    shipLayer.contents = (__bridge id)[UIImage imageNamed:@"Ship.png"].CGImage;
    [self.containerView.layer addSublayer:shipLayer];

    //animate the ship rotation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform";
    animation.duration = 2.0;
    animation.toValue = [NSValue valueWithCATransform3D:
        CATransform3DMakeRotation(M_PI, 0, 0, 1)];
    [shipLayer addAnimation:animation forKey:nil];
}

@end

```

This works, but it turns out that this is more by luck than design. If we were to change the rotation value from `M_PI` (180 degrees) to `2 * M_PI` (360 degrees) and run the animation, we'd find that the ship doesn't move at all. That's because the matrix representation for a rotation of 360 degrees is the same as for 0 degrees, so as far as the animation is concerned, the value hasn't changed.

Now try using `M_PI` again, but set it as the `byValue` property instead of `toValue`, to indicate that the rotation should be *relative* to the current value. You might expect that to have the same effect as setting `toValue`, because $0 + 90 \text{ degrees} == 90 \text{ degrees}$, but in fact the ship image *expands* instead of rotating because transform matrices cannot be added together like angular values can.

What if we want to animate the translation or scale of the ship independently of its angle? Because both of those require us to modify the `transform` property, we would need to recalculate the combined effect of each of those animations at each point in time and create a complex keyframe animation from the combined transform values, even though all we really want to do is animate a few conceptually discrete attributes of our layer independently.

Fortunately, there is a solution: To rotate the layer, we can apply our animation to the `transform.rotation` key path instead of animating the `transform` property itself (see Listing 8.9).

Listing 8.9 Animating the Virtual `transform.rotation` Property

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add the ship
    CALayer *shipLayer = [CALayer layer];
    shipLayer.frame = CGRectMake(0, 0, 128, 128);
    shipLayer.position = CGPointMake(150, 150);
    shipLayer.contents = (__bridge id)[UIImage imageNamed:@"Ship.png"].CGImage;
    [self.containerView.layer addSublayer:shipLayer];

    //animate the ship rotation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform.rotation";
    animation.duration = 2.0;
    animation.byValue = @(M_PI * 2);
    [shipLayer addAnimation:animation forKey:nil];
}

@end
```

This approach works *great*. The benefits of animating `transform.rotation` instead of the `transform` are as follows:

- It allows us to rotate more than 180 degrees in a single step, without using keyframes.
- It allows us to perform a relative rather than absolute rotation (by setting the `byValue` instead of `toValue`).
- It allows us to specify the angle as a simple numeric value instead of constructing a `CATransform3D`.
- It won't conflict with `transform.position` or `transform.scale` (which are also individually animatable using key paths).

The odd thing about the `transform.rotation` property is that it *doesn't really exist*. It *can't* exist because `CATransform3D` isn't an object; it's a struct and so cannot have KVC (Key-Value Coding) compliant properties. `transform.rotation` is actually a *virtual* property that `CALayer` provides to simplify the process of animating transforms.

You cannot set properties like `transform.rotation` or `transform.scale` directly; they can only be used for animation. When you animate these properties, Core Animation automatically updates the `transform` property with the actual value that your changes necessitate by using a class called `CAValueFunction`.

`CAValueFunction` is used to convert the simple floating-point value that we assign to the virtual `transform.rotation` property into the actual `CATransform3D` matrix value that is needed to position the layer. You can change the value function used by a given `CAPropertyAnimation` by setting its `valueFunction` property. The function you specify will override the default.

`CAValueFunction` *seems* like it could be a useful mechanism for animating properties that cannot naturally be summed together or interpolated (such as transform matrices), but because the implementation details of `CAValueFunction` are private, it's not currently possible to subclass it to create new value functions. You can only use the functions that Apple already make available as constants (which currently all relate to the transform matrix's virtual properties and are therefore somewhat redundant since the default actions for those properties already use the appropriate value functions).

Animation Groups

Although `CABasicAnimation` and `CAKeyframeAnimation` only target individual properties, multiple such animations can be gathered together using a `CAAnimationGroup`.

`CAAnimationGroup` is another concrete subclass of `CAAnimation` that adds an `animations` array property, to be used for grouping other animations. Let's test this out by grouping the keyframe animation in Listing 8.6 together with another basic animation that adjusts the layer background color (see Listing 8.10). Figure 8.3 shows the result.

Adding an animation group to a layer is not fundamentally different from adding the animations individually, so it's not immediately clear when or why you would use this class. It provides *some* convenience in terms of being able to collectively set animation durations, or add and remove multiple animations from a layer with a single command, but it's usefulness only really becomes apparent when it comes to *hierarchical timing*, which is explained in Chapter 9.

Listing 8.10 Grouping a Keyframe and Basic Animation Together

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
//create a path
UIBezierPath *bezierPath = [[UIBezierPath alloc] init];
[bezierPath moveToPoint:CGPointMake(0, 150)];
[bezierPath addCurveToPoint:CGPointMake(300, 150)
    controlPoint1:CGPointMake(75, 0)
    controlPoint2:CGPointMake(225, 300)];

//draw the path using a CAShapeLayer
CAShapeLayer *pathLayer = [CAShapeLayer layer];
pathLayer.path = bezierPath.CGPath;
pathLayer.fillColor = [UIColor clearColor].CGColor;
pathLayer.strokeColor = [UIColor redColor].CGColor;
pathLayer.lineWidth = 3.0f;
[self.containerView.layer addSublayer:pathLayer];

//add a colored layer
CALayer *colorLayer = [CALayer layer];
colorLayer.frame = CGRectMake(0, 0, 64, 64);
colorLayer.position = CGPointMake(0, 150);
colorLayer.backgroundColor = [UIColor greenColor].CGColor;
[self.containerView.layer addSublayer:colorLayer];

//create the position animation
CAKeyframeAnimation *animation1 = [CAKeyframeAnimation animation];
animation1.keyPath = @"position";
animation1.path = bezierPath.CGPath;
animation1.rotationMode = kCAAnimationRotateAuto;

//create the color animation
CABasicAnimation *animation2 = [CABasicAnimation animation];
animation2.keyPath = @"backgroundColor";
animation2.toValue = (_bridge id)[UIColor redColor].CGColor;

//create group animation
CAAnimationGroup *groupAnimation = [CAAnimationGroup animation];
groupAnimation.animations = @*[animation1, animation2];
groupAnimation.duration = 4.0;

//add the animation to the color layer
[colorLayer addAnimation:groupAnimation forKey:nil];
}
```

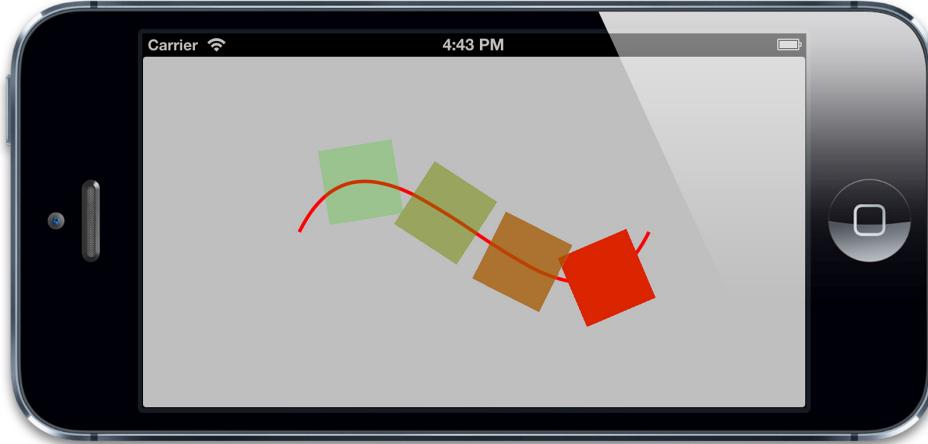


Figure 8.3 A grouped keyframe path and basic color property animation

Transitions

Sometimes with iOS applications it is necessary to make layout changes that are very difficult to animate using property animations. You might need to swap some text or an image, for example, or replace a whole grid or table of views at once. Property animations only work on animatable properties of a layer, so if you need to change a nonanimatable property (such as an image) or actually add and remove layers from the hierarchy, property animations won't work.

This is where transitions come in. A transition animation does not try to smoothly interpolate between two values like a property animation; instead it is designed as a sort of *distraction tactic*—to *cover up* content changes with an animation. Transitions affect an entire layer instead of just a specific property. The transition takes a snapshot of the old layer appearance and then animates in the new appearance in a single sweep.

To create a transition, we use `CATransition`, another subclass of `CAAnimation`. In addition to all the timing functions and so on that it inherits from `CAAnimation`, `CATransition` has a `type` and a `subtype` that are used to specify the transition effect. The `type` property is an `NSString` and can be set to one of the following constant values:

```
kCATransitionFade  
kCATransitionMoveIn  
kCATransitionPush  
kCATransitionReveal
```

You are currently limited to these four basic `CATransition` types, but there are some ways that you can achieve additional transition effects, as covered later in the chapter.

The default transition type is `kCATransitionFade`, which creates a smooth crossfade between the previous layer appearance and the new appearance after you have modified its properties or contents.

We made use of the `kCATransitionPush` type in the custom action example in Chapter 7; this slides the new layer appearance in from the side, pushing the old one out of the opposite side.

`kCATransitionMoveIn` and `kCATransitionReveal` are similar to `kCATransitionPush`; they both implement a directional swipe animation, but with subtle differences; `kCATransitionMoveIn` moves the new layer appearance in over the top of the previous appearance, but doesn't push it out to the side like the push transition, and `kCATransitionReveal` slides the old appearance out to reveal the new one instead of sliding the new one in.

The latter three standard transition types are inherently directional in nature. By default, they slide from the left, but you can control their direction using the `subtype` property, which accepts one of the following constants:

```
kCATransitionFromRight  
kCATransitionFromLeft  
kCATransitionFromTop  
kCATransitionFromBottom
```

A simple example of using `CATransition` to animate a nonanimatable property is shown in Listing 8.11. Here we are changing the `image` property of a `UIImageView`, which cannot normally be animated using either implicit animation or a `CAPropertyAnimation` because Core Animation doesn't know how to interpolate between images. By using a crossfade transition applied to the layer, however, we make it possible to smoothly animate the change regardless of the content type (see Figure 8.4). Try changing the transition `type` constant to see the other possible effects.

Listing 8.11 Animating a `UIImageView` Using `CATransition`

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIImageView *imageView;  
@property (nonatomic, copy) NSArray *images;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{
```

```

[super viewDidLoad];

//set up images
self.images = @[[UIImage imageNamed:@"Anchor.png"],
                [UIImage imageNamed:@"Cone.png"],
                [UIImage imageNamed:@"Igloo.png"],
                [UIImage imageNamed:@"Spaceship.png"]];
}

- (IBAction)switchImage
{
    //set up crossfade transition
    CATransition *transition = [CATransition animation];
    transition.type = kCATransitionFade;

    //apply transition to imageview backing layer
    [self.imageView.layer addAnimation:transition forKey:nil];

    //cycle to next image
    UIImage *currentImage = self.imageView.image;
    NSUInteger index = [self.images indexOfObject:currentImage];
    index = (index + 1) % [self.images count];
    self.imageView.image = self.images[index];
}

@end

```

As you can see from the code, transitions are added to a layer in the same way as property or group animations, using the `-addAnimation:forKey:` method. Unlike property animations, though, only one `CATransition` can operate on a given layer at a time. For this reason, regardless of what value you specify for the key, the transition will actually be attached with a key of "transition", which is represented by the constant `kCATransition`.

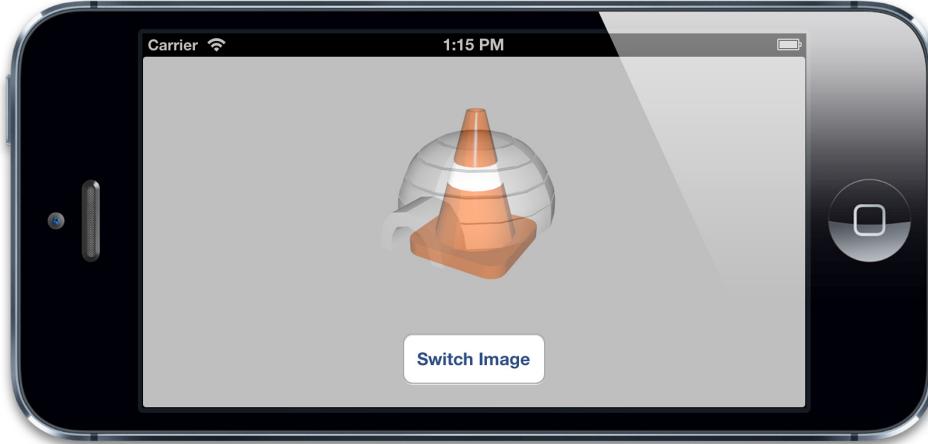


Figure 8.4 Smoothly crossfading between images using `CATransition`

Implicit Transitions

The fact that `CATransition` can smoothly cover any changes made to a layer makes it an ideal candidate for use as a layer action for properties that are otherwise hard to animate. Apple realizes this, of course, and `CATransition` is used as the default action when setting the `CALayer contents` property. This is disabled for view backing layers along with all other implicit animation actions, but for layers that you create yourself, this means that changes to the `layer contents` image are automatically animated with a crossfade.

We used a `CATransition` as a layer action in Chapter 7 to animate changes to our layer's background color. The `backgroundColor` property can be animated using a normal `CAPropertyAnimation` but that doesn't mean you can't use a `CATransition` instead.

Animating Layer Tree Changes

The fact that `CATransition` does not operate on specific layer properties means that you can use it to animate layer changes even when you do not know exactly what has changed. You can, for example, smoothly cover the reloading of a complex `UITableView` with a crossfade without needing to know which rows have been added and removed, or transition between two different `UIViewController` instances without needing to know anything about their internal view hierarchies.

Both of these cases are different from anything we've tried so far because they involve animating not only changes to a layer's properties but actual *layer tree* changes—we need to physically add and remove layers from the hierarchy during the course of the animation.

The trick in this case is to ensure that the layer the `CATransition` is attached to will not itself be removed from the tree during the transition, because then the `CATransition` will be removed along with it. Generally, you just need to attach the transition to the superlayer of the layers that are affected.

In Listing 8.12, we show how you can implement a crossfade transition between tabs in a `UITabBarController`. Here we have simply taken the default Tabbed Application project template and used the `-tabBarController:didSelectViewController:` method of the `UITabBarControllerDelegate` to apply the transition animation. We've attached the transition to the `UITabBarController` view's layer because that doesn't get replaced when the tabs themselves are swapped.

Listing 8.12 Animating a `UITabBarController`

```
#import "AppDelegate.h"
#import "FirstViewController.h"
#import "SecondViewController.h"
#import <QuartzCore/QuartzCore.h>

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
                  [[UIScreen mainScreen] bounds]];
    UIViewController *viewController1 = [[FirstViewController alloc] init];
    UIViewController *viewController2 = [[SecondViewController alloc] init];
    self.tabBarController = [[UITabBarController alloc] init];
    self.tabBarController.viewControllers = @[viewController1,
                                             viewController2];
    self.tabBarController.delegate = self;
    self.window.rootViewController = self.tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)tabBarController:(UITabBarController *)tabBarController
didSelectViewController:(UIViewController *)viewController
{
    //set up crossfade transition
    CATransition *transition = [CATransition animation];
    transition.type = kCATransitionFade;

    //apply transition to tab bar controller's view
    [self.tabBarController.view.layer addAnimation:transition forKey:nil];
}
```

```
}
```

```
@end
```

Custom Transitions

We've established that transitions are a powerful way to animate properties that would otherwise be difficult to change smoothly. But the list of animation types for `CATransition` seems a bit limited.

What's even stranger is that Apple exposes Core Animation's transition feature via the `UIView +transitionFromView:toView:duration:options:completion:` and `+transitionWithView:duration:options:animations:` methods, but the transition options available are *completely different* to the constants made available via the `CATransition type` property. The constants that can be specified for the `UIView` transition method `options` parameter are as follows:

```
UIViewControllerAnimatedTransitionFromLeft  
UIViewControllerAnimatedTransitionFromRight  
UIViewControllerAnimatedTransitionCurlUp  
UIViewControllerAnimatedTransitionCurlDown  
UIViewControllerAnimatedTransitionCrossDissolve  
UIViewControllerAnimatedTransitionFlipFromTop  
UIViewControllerAnimatedTransitionFlipFromBottom
```

With the exception of `UIViewControllerAnimatedTransitionCrossDissolve`, none of these transitions correspond to the `CATransition` types. You can test these alternative transitions by using a modified version of our earlier transition example (see Listing 8.13).

Listing 8.13 Alternative Transition Implementation Using UIKit Methods

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIImageView *imageView;  
@property (nonatomic, copy) NSArray *images;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //set up images
```

```

self.images = @[[UIImageView imageNamed:@"Anchor.png"],
               [UIImageView imageNamed:@"Cone.png"],
               [UIImageView imageNamed:@"Igloo.png"],
               [UIImageView imageNamed:@"Spaceship.png"]];
}

- (IBAction)switchImage
{
    [UIView transitionWithView:self.imageView
                         duration:1.0
                           options:UIViewAnimationOptionTransitionFlipFromLeft
                         animations:^{
        //cycle to next image
        UIImage *currentImage = self.imageView.image;
        NSUInteger index = [self.images indexOfObject:currentImage];
        index = (index + 1) % [self.images count];
        self.imageView.image = self.images[index];
    } completion:NULL];
}

@end

```

The documentation in some places seems to imply that since iOS 5 (which introduced the Core Image framework), it might be possible to use `CIFilter` in conjunction with the `filter` property of `CATransition` to create additional transition types. As of iOS 6, however, this still does not work. Attempting to use Core Image filters with `CATransition` has no effect. (This *is* supported on Mac OS, which probably accounts for the documentation discrepancies.)

For this reason, you are forced to choose between using a `CATransition` or the `UIView` transition method, depending on the effect you want. Hopefully, a future version of iOS will add support for Core Image transition filters and so make the full range of Core Image transition animations available via `CATransition` (and maybe even add the ability to create new ones).

That doesn't mean that it's impossible to achieve custom transition effects on iOS, though. It just means that you have to do a bit of extra work. As mentioned earlier, the basic principle of a transition animation is that you take a snapshot of the current state of the layer and then apply an animation to that snapshot while you change the layer behind the scenes. If we can figure out how to take a snapshot of our layer, we can perform the animation ourselves using ordinary property animations without using `CATransition` or UIKit's transition methods at all.

As it turns out, taking a snapshot of a layer is relatively easy. `CALayer` has a `-renderInContext:` method that can be used to capture an image of its current contents by drawing it into a Core Graphics context, which can then be displayed in another view. If we place this snapshot view in front of the original, it will mask any changes we make to the real view's contents, allowing us to re-create the effect of a simple transition.

Listing 8.14 demonstrates a basic implementation of this idea: We take a snapshot of the current view state and then spin and fade out the snapshot while we change the background color of the original view. Figure 8.5 shows our custom transition in progress.

To keep things simple, we've performed the animation using the `UIView -animateWithDuration:completion:` method. Although it would be possible to perform the exact same effect using `CABasicAnimation`, we would have to set up separate animations for the layer transform and opacity properties and implement the `CAAnimationDelegate` to remove `coverView` from the screen once the animation has completed.

Listing 8.14 Creating a Custom Transition Using `renderInContext:`

```
@implementation ViewController

- (IBAction)performTransition
{
    //preserve the current view snapshot
    UIGraphicsBeginImageContextWithOptions(self.view.bounds.size, YES, 0.0);
    [self.view.layer renderInContext:UIGraphicsGetCurrentContext()];
    UIImage *coverImage = UIGraphicsGetImageFromCurrentImageContext();

    //insert snapshot view in front of this one
    UIView *coverView = [[UIImageView alloc] initWithImage:coverImage];
    coverView.frame = self.view.bounds;
    [self.view addSubview:coverView];

    //update the view (we'll simply randomize the layer background color)
    CGFloat red = arc4random() / (CGFloat)INT_MAX;
    CGFloat green = arc4random() / (CGFloat)INT_MAX;
    CGFloat blue = arc4random() / (CGFloat)INT_MAX;
    self.view.backgroundColor = [UIColor colorWithRed:red
                                              green:green
                                              blue:blue
                                             alpha:1.0];

    //perform animation (anything you like)
    [UIView animateWithDuration:1.0 animations:^{
        //scale, rotate and fade the view
        CGAffineTransform transform = CGAffineTransformMakeScale(0.01, 0.01);
    }];
}
```

```

        transform = CGAffineTransformRotate(transform, M_PI_2);
        coverView.transform = transform;
        coverView.alpha = 0.0;

    } completion:^(BOOL finished) {

        //remove the cover view now we're finished with it
        [coverView removeFromSuperview];
    }];
}

@end

```

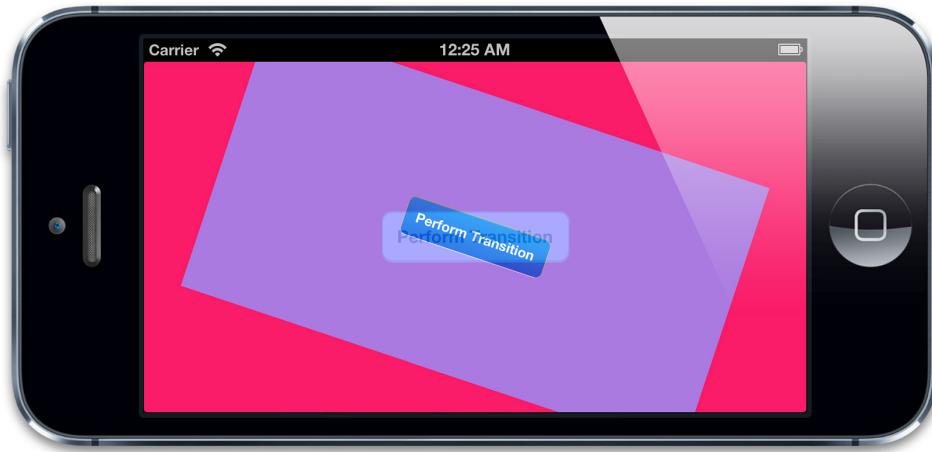


Figure 8.5 A custom transition implemented using `renderInContext:`

There is a caveat to this approach: The `-renderInContext:` method captures the layer's backing image and sublayers, but does not correctly handle transforms applied to those sublayers, and doesn't work with video or OpenGL content. `CATransition` doesn't seem to be affected by this limitation, so is presumably using a private method to capture the snapshot.

Cancelling an Animation in Progress

As mentioned earlier in this chapter, you can use the `key` parameter of the `-addAnimation:forKey:` method to retrieve an animation after it has been added to a layer by using the following method:

```
- (CAAnimation *)animationForKey:(NSString *)key;
```

Modifying animations once they are in progress is not supported, so the primary uses of this are to check what the animation properties are or to detect whether a particular animation is attached to the layer.

To terminate a specific animation, you can remove it from the layer using the following method:

```
- (void)removeAnimationForKey:(NSString *)key;
```

Or you can just remove all animations using this method:

```
- (void)removeAllAnimations;
```

As soon as an animation is removed, the layer appearance updates to match its current model value. Animations are removed automatically when they finish unless you specify that they shouldn't by setting the `removedOnCompletion` property of the animation to `NO`. If you set the animation to *not* be removed automatically, it is important to remove it yourself when it is no longer needed; otherwise, it will stay in memory until the layer itself is eventually destroyed.

Let's extend our rotating ship example once again with buttons to stop and start the animation. This time, we provide a non-`nil` value for our animation key so that we can remove it later. The `flag` argument in the `-animationDidStop:finished:` method indicates whether the animation finished naturally or was interrupted, which we log in the console. If you terminate the animation using the stop button, it will log `NO`, but if you allow it to complete, it will log `YES`.

See Listing 8.15 for the updated example code. Figure 8.6 shows it in action.

Listing 8.15 Starting and Stopping an Animation

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic, strong) CALayer *shipLayer;  
  
@end
```

```

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add the ship
    self.shipLayer = [CALayer layer];
    self.shipLayer.frame = CGRectMake(0, 0, 128, 128);
    self.shipLayer.position = CGPointMake(150, 150);
    self.shipLayer.contents = (_bridge id)[UIImage imageNamed:
                                         @"Ship.png"].CGImage;
    [self.containerView.layer addSublayer:self.shipLayer];
}

- (IBAction)start
{
    //animate the ship rotation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform.rotation";
    animation.duration = 2.0;
    animation.byValue = @(M_PI * 2);
    animation.delegate = self;
    [self.shipLayer addAnimation:animation forKey:@"rotateAnimation"];
}

- (IBAction)stop
{
    [self.shipLayer removeAnimationForKey:@"rotateAnimation"];
}

- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
    //log that the animation stopped
    NSLog(@"The animation stopped (finished: %@", flag? @"YES": @"NO");
}

@end

```

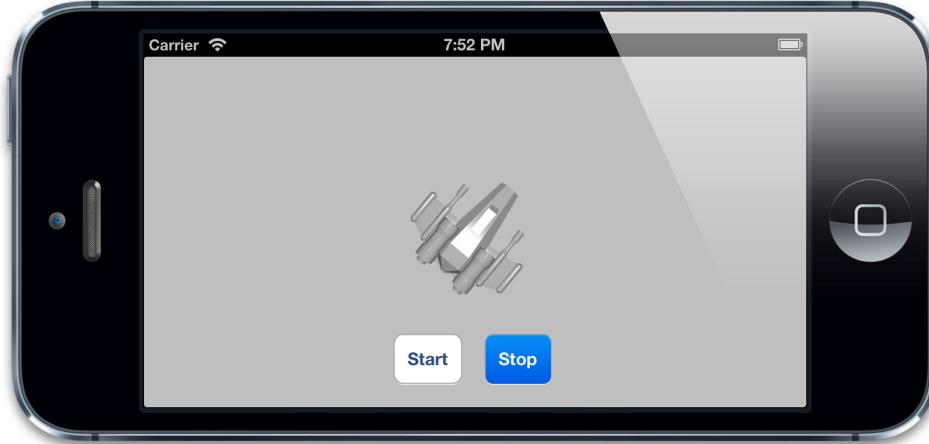


Figure 8.6 A rotation animation controlled by start and stop buttons

Summary

In this chapter, we covered property animations (which allow you to exert very specific control over the animation of individual layer properties), animation groups (which allow you to combine multiple property animations into a single unit), and transitions (which affect an entire layer and can be used to animate any sort of change to the layer's contents, including the addition and removal of sublayers).

In Chapter 9, we study the `CAMediaTiming` protocol and discover how Core Animation deals with the passage of time.

Layer Time

The biggest difference between time and space is that you can't reuse time.

Merrick Furst

In the previous two chapters, we explored the various types of layer animation that can be implemented using `CAAnimation` and its subclasses. Animation is a change that happens over time, so *timing* is crucial to the whole concept. In this chapter, we will look at the `CAMediaTiming` protocol, which is how Core Animation keeps track of time.

The `CAMediaTiming` Protocol

The `CAMediaTiming` protocol defines a collection of properties that are used to control the passage of time during an animation. Both `CALayer` and `CAAnimation` conform to this protocol, so time can be controlled on both a per-layer and per-animation basis.

Duration and Repetition

We briefly mentioned `duration` (one of the `CAMediaTiming` properties) in Chapter 8, “Explicit Animations.” The `duration` property is of type `CFTimeInterval` (which is a double-precision floating-point value that represents seconds, just like `NSTimeInterval`), and it is used to specify the duration for which a single iteration of an animation will run.

What do we mean by a *single iteration*? Well, another property of `CAMediaTiming` is `repeatCount`, which determines the number of iterations that an animation will be repeated for. The value of `repeatCount` represents the total number of times the animation will be played. If the `duration` is 2 seconds, and `repeatCount` is set to 3.5 (three-and-a-half iterations), the total time spent animating will be 7 seconds.

The duration and repeatCount properties both default to zero. This doesn't mean that the animation has a duration of zero seconds, or repeats zero times; a value of zero in this case, is just used to mean "use the defaults," which are 0.25 seconds and one iteration, respectively. You can try out various values for these properties using the simple test program in Listing 9.1. Figure 9.1 shows the program interface.

Listing 9.1 Testing the duration and repeatCount Properties

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic, weak) IBOutlet UITextField *durationField;  
@property (nonatomic, weak) IBOutlet UITextField *repeatField;  
@property (nonatomic, weak) IBOutlet UIButton *startButton;  
@property (nonatomic, strong) CALayer *shipLayer;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //add the ship  
    self.shipLayer = [CALayer layer];  
    self.shipLayer.frame = CGRectMake(0, 0, 128, 128);  
    self.shipLayer.position = CGPointMake(150, 150);  
    self.shipLayer.contents = (__bridge id)[UIImage imageNamed:  
        @"Ship.png"].CGImage;  
    [self.containerView.layer addSublayer:self.shipLayer];  
}  
  
- (void)setControlsEnabled:(BOOL)enabled  
{  
    for (UIControl *control in @[self.durationField,  
        self.repeatField, self.startButton])  
    {  
        control.enabled = enabled;  
        control.alpha = enabled? 1.0f: 0.25f;  
    }  
}  
  
- (IBAction)hideKeyboard  
{  
    [self.durationField resignFirstResponder];
```

```
[self.repeatField resignFirstResponder];
}

- (IBAction)start
{
    CFTimeInterval duration = [self.durationField.text doubleValue];
    float repeatCount = [self.repeatField.text floatValue];

    //animate the ship rotation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform.rotation";
    animation.duration = duration;
    animation.repeatCount = repeatCount;
    animation.byValue = @(M_PI * 2);
    animation.delegate = self;
    [self.shipLayer addAnimation:animation forKey:@"rotateAnimation"];

    //disable controls
    [self setControlsEnabled:NO];
}

- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
    //reenable controls
    [self setControlsEnabled:YES];
}

@end
```

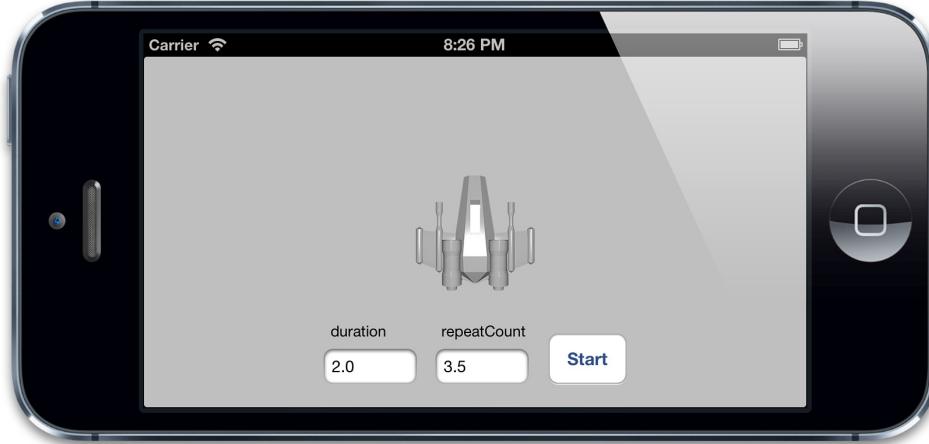


Figure 9.1 A test program to demonstrate the `duration` and `repeatCount` properties

An alternative way to create a repeating animation is to use the `repeatDuration` property, which tells the animation to repeat for a fixed time period instead of for a fixed number of iterations. You can even set a property called `autoreverses` (of type `BOOL`) to make the animation play backward during each alternate cycle. This is great for playing a nonlooping animation continuously, such as a door swinging open and then shut (see Figure 9.2).

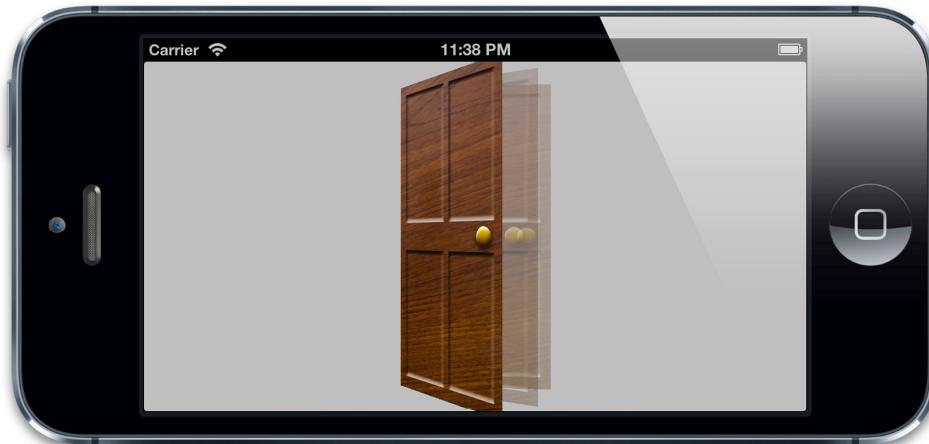


Figure 9.2 A swinging door animation

The code for the swinging door is shown in Listing 9.2. We've simply animated the outward swing and used the `autoreverses` property to make the door swing back automatically. In this case, we've set `repeatDuration` to `INFINITY` so that the animation plays indefinitely, although setting `repeatCount` to `INFINITY` would have had the same effect. Note that the `repeatCount` and `repeatDuration` properties could potentially contradict each other, so you should only specify a nonzero value for either one or the other. The behavior if *both* properties are nonzero is undefined.

Listing 9.2 Swinging Door Implemented Using the `autoreverses` Property

```
@interface ViewController : UIViewController

@property (nonatomic, weak) UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add the door
    CALayer *doorLayer = [CALayer layer];
    doorLayer.frame = CGRectMake(0, 0, 128, 256);
    doorLayer.position = CGPointMake(150 - 64, 150);
    doorLayer.anchorPoint = CGPointMake(0, 0.5);
    doorLayer.contents = (__bridge id)[UIImage imageNamed:@"Door.png"].CGImage;
    [self.containerView.layer addSublayer:doorLayer];

    //apply perspective transform
    CATransform3D perspective = CATransform3DIdentity;
    perspective.m34 = -1.0 / 500.0;
    self.containerView.layer.sublayerTransform = perspective;

    //apply swinging animation
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform.rotation.y";
    animation.toValue = @(-M_PI_2);
    animation.duration = 2.0;
    animation.repeatDuration = INFINITY;
    animation.autoreverses = YES;
    [doorLayer addAnimation:animation forKey:nil];
```

```
}
```

```
@end
```

Relative Time

As far as Core Animation is concerned, time is relative. Each animation has its own representation of time, which can be independently sped up, delayed, or offset.

The `beginTime` property specifies the time delay before the animation begins. This delay is measured from the point at which the animation is added to a visible layer. This defaults to zero (that is, the animation will begin immediately).

The `speed` property is a time *multiplier*. It defaults to 1.0, but decreasing it will slow down time for the layer/animation and increasing it will speed up time. With a speed of 2.0, an animation with a nominal duration of 1 second will actually complete in 0.5 seconds.

The `timeOffset` property is similar to `beginTime` in that it time-shifts the animation. But while increasing the `beginTime` increases the delay before an animation begins, increasing the `timeOffset` fast-forwards to a specific point in the animation. For example, for an animation that lasts 1 second, setting a `timeOffset` of 0.5 seconds would mean that the animation starts halfway through.

Unlike `beginTime`, `timeOffset` is unaffected by `speed`. So, if you were to increase the `speed` to 2.0 as well as setting the `timeOffset` to 0.5, you would have effectively skipped to the end of the animation because a 1-second animation sped up by a factor of two lasts for only 0.5 seconds. However, even if you skip to the end of the animation using the `timeOffset`, it will still play for the same total duration; the animation simply loops around and plays again up until the point where it originally started.

You can try this out with the simple test app in Listing 9.3. Just set the `speed` and `timeOffset` sliders to the desired value, and then press Play to see the effect they have (see Figure 9.3).

Listing 9.3 Testing the `timeOffset` and `speed` Properties

```
@interface ViewController : UIViewController  
{  
    @property (nonatomic, weak) IBOutlet UIView *containerView;  
    @property (nonatomic, weak) IBOutlet UILabel *speedLabel;  
    @property (nonatomic, weak) IBOutlet UILabel *timeOffsetLabel;  
    @property (nonatomic, weak) IBOutlet UISlider *speedSlider;  
    @property (nonatomic, weak) IBOutlet UISlider *timeOffsetSlider;  
    @property (nonatomic, strong) UIBezierPath *bezierPath;  
    @property (nonatomic, strong) CALayer *shipLayer;
```

```

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a path
    self.bezierPath = [[UIBezierPath alloc] init];
    [self.bezierPath moveToPoint:CGPointMake(0, 150)];
    [self.bezierPath addCurveToPoint:CGPointMake(300, 150)
                                controlPoint1:CGPointMake(75, 0)
                                controlPoint2:CGPointMake(225, 300)];

    //draw the path using a CAShapeLayer
    CAShapeLayer *pathLayer = [CAShapeLayer layer];
    pathLayer.path = self.bezierPath.CGPath;
    pathLayer.fillColor = [UIColor clearColor].CGColor;
    pathLayer.strokeColor = [UIColor redColor].CGColor;
    pathLayer.lineWidth = 3.0f;
    [self.containerView.layer addSublayer:pathLayer];

    //add the ship
    self.shipLayer = [CALayer layer];
    self.shipLayer.frame = CGRectMake(0, 0, 64, 64);
    self.shipLayer.position = CGPointMake(0, 150);
    self.shipLayer.contents = (__bridge id)[UIImage imageNamed:
                                             @"Ship.png"].CGImage;
    [self.containerView.layer addSublayer:self.shipLayer];

    //set initial values
    [self updateSliders];
}

- (IBAction)updateSliders
{
    CFTimeInterval timeOffset = self.timeOffsetSlider.value;
    self.timeOffsetLabel.text = [NSString stringWithFormat:@"%@", timeOffset];

    float speed = self.speedSlider.value;
    self.speedLabel.text = [NSString stringWithFormat:@"%@", speed];
}

- (IBAction)play

```

```

{
    //create the keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"position";
    animation.timeOffset = self.timeOffsetSlider.value;
    animation.speed = self.speedSlider.value;
    animation.duration = 1.0;
    animation.path = self.bezierPath.CGPath;
    animation.rotationMode = kCAAnimationRotateAuto;
    animation.removedOnCompletion = NO;
    [self.shipLayer addAnimation:animation forKey:@"slide"];
}

@end

```



Figure 9.3 A simple app to test the effect of time offset and speed

fillMode

An animation that has a `beginTime` greater than zero can be in a state where it is attached to a layer but has not yet started animating. Similarly, an animation whose `removeOnCompletion` property is set to `NO` will remain attached to a layer after it has finished. This raises the question of what the value of the animated properties will be before the animation has started and after it has ended.

One possibility is that the property values could be the same as if the animation was not attached at all, that is, the value would be whatever was defined in the model layer. (See Chapter 7, “Implicit Animations,” for an explanation of layer model versus presentation.)

Another option is that the properties could take on the value of the first frame of the animation prior to it beginning and retain the last frame after it has ended. This is known as *filling*, because the animation’s start and end values are used to fill the time before and after the animation’s duration.

This behavior is left up to the developer; it can be controlled using the `fillMode` property of `CAMediaTiming`. `fillMode` is an `NSString` and accepts one of the following constant values:

```
kCAFillModeForwards  
kCAFillModeBackwards  
kCAFillModeBoth  
kCAFillModeRemoved
```

The default is `kCAFillModeRemoved`, which sets the property values to whatever the layer model specifies when the animation isn’t currently playing. The other three modes fill the animation forward, backward, or both so that the animatable properties take on the start value specified in the animation before it begins and/or the end value once it has finished.

This provides an alternative solution to the problem of having to manually update a layer property to match the end value of an animation to avoid snap-back when the animation finishes (mentioned in Chapter 8). Bear in mind, however, that if you intend to use it for this purpose, you will need to set `removeOnCompletion` to `NO` for your animation. You will also need to use a non-nil key when adding the animation, so that you can manually remove it from the layer when you no longer need it.

Hierarchical Time

In Chapter 3, “Layer Geometry,” you learned how each layer has a spatial coordinate system that is defined relative to its parent in the layer tree. Animation timing works in a similar way. Each animation and layer has its own hierarchical concept of time, measured relative to its parent. Adjusting the timing for a layer will affect its own animations and those of its sublayers, but not its superlayer. The same goes for animations that are grouped hierarchically (using nested `CAAnimationGroup` instances).

Adjusting the `duration` and `repeatCount/repeatDuration` properties for a `CALayer` or `CAGroupAnimation` will not affect its children’s animations. The `beginTime`, `timeOffset`, and `speed` properties *will* impact child animations, however. In hierarchical terms, `beginTime` specifies a time offset between when the parent layer (or parent animation in the case of a grouped animation) starts animating and when the object in question should begin its own animation. Similarly, adjusting the `speed` property of a `CALayer` or `CAGroupAnimation` will apply a scaling factor to the animation speed for all of the children, as well.

Global Versus Local Time

Core Animation has a concept of *global time*, also known as *mach time* (“mach” being the name for the system kernel on iOS and Mac OS). Mach time is global only in the sense that it is the same across all processes on the device—it isn’t universal across different devices—but that is sufficient to make it useful as a reference point for animations. To access mach time, you can use the `CACurrentMediaTime` function, as follows:

```
CFTimeInterval time = CACurrentMediaTime();
```

The *absolute* value returned by this function is largely irrelevant (it reflects the number of seconds that the device has been awake since it was last rebooted, which is unlikely to be something you care about), but its purpose is to act as a *relative* value against which you can make timing measurements. Note that mach time pauses when the device is asleep, which means that all `CAAnimations` (which depend on mach time) will also pause.

For this reason, mach time is not useful for making long-term time measurements. It would be unwise to use `CACurrentMediaTime` to update a real-time clock, for example. (You can poll the current value of `[NSDate date]` for that purpose instead, as we did in our clock example in Chapter 3.)

Each `CALayer` and `CAAnimation` instance has its own *local* concept of time, which may differ from global time depending on the `beginTime`, `timeOffset`, and `speed` properties of the parent objects in the layer/animation hierarchy. Just as there are methods to convert between different layers’ local spatial coordinate systems, `CALayer` also has methods to convert between different layers’ local *time frames*. These are as follows:

- `(CFTimeInterval)convertTime:(CFTimeInterval)t fromLayer:(CALayer *)l;`
- `(CFTimeInterval)convertTime:(CFTimeInterval)t toLayer:(CALayer *)l;`

These methods can be useful if you are trying to synchronize animations between multiple layers that do not share the same `speed`, or have nonzero `timeOffset` or `beginTime` values.

Pause, Rewind, and Fast-Forward

Setting an animation’s `speed` property to zero will pause it, but it’s not actually possible to modify an animation after it has been added to a layer, so you can’t use this property to pause an animation that’s in progress. Adding a `CAAnimation` to a layer makes an immutable copy of the animation object; so changing the properties of the original animation has no effect on the one actually attached to the layer. Conversely, retrieving the in-progress animation directly from the layer by using `-animationForKey:` will give you the correct animation object, but attempting to modify its properties will throw an exception.

If you remove an in-progress animation from its layer, the layer will snap back to its pre-animated state. But if you copy the property values from the presentation layer to the model layer before removing the animation, the animation will appear to have paused. The disadvantage of this is that you cannot easily resume the animation again later.

A simpler and less-destructive approach is to make use of the hierarchical nature of `CAMediaTiming` and pause the *layer* itself. If you set the layer speed to zero, it will pause any animations that are attached to that layer. Similarly, setting speed to a value greater than 1.0 will “fast-forward,” and setting a negative speed will “rewind” the animation.

By increasing the layer `speed` for the main window in your app, you can actually speed up animations across the entire application. This can prove useful for something like UI automation, where the tests can be made to run faster if you speed up all the view transitions. (Note that views that are displayed outside of the main window—such as `UIAlertView`—will not be affected.) Try adding the following line to your app delegate to see this in action:

```
self.window.layer.speed = 100;
```

You can also *slow down* animations across the app in this way, but this is less useful since it’s already possible to slow down animations in the iOS Simulator by using the Toggle Slow Animations option in the Debug menu.

Manual Animation

A really interesting feature of the `timeOffset` property is that it enables you to manually *scrub* through an animation. By setting `speed` to zero, you can disable the automatic playback of an animation and then use the `timeOffset` to move back and forth through the animation sequence. This can be a nifty way to allow the user to manipulate an animated user interface element using gestures.

We’ll try a simple example first: Starting with the swinging door animation from earlier in the chapter, let’s modify the code so that the animation is controlled using a finger gesture. We do this by attaching a `UIPanGestureRecognizer` to our view and then using it to set the `timeOffset` by swiping left and right.

Because we cannot modify the animation after it has been added to the layer, what we will do instead is pause and adjust the `timeOffset` value for the *layer*, which in this case has the same effect as if we were manipulating the animation directly (see Listing 9.4).

Listing 9.4 Manually Driving an Animation Using Touch Gestures

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) UIView *containerView;  
@property (nonatomic, strong) CALayer *doorLayer;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad
```

```

{
    [super viewDidLoad];

    //add the door
    self.doorLayer = [CALayer layer];
    self.doorLayer.frame = CGRectMake(0, 0, 128, 256);
    self.doorLayer.position = CGPointMake(150 - 64, 150);
    self.doorLayer.anchorPoint = CGPointMake(0, 0.5);
    self.doorLayer.contents =
        (__bridge id)[UIImage imageNamed:@"Door.png"].CGImage;
    [self.containerView.layer addSublayer:self.doorLayer];

    //apply perspective transform
    CATransform3D perspective = CATransform3DIdentity;
    perspective.m34 = -1.0 / 500.0;
    self.containerView.layer.sublayerTransform = perspective;

    //add pan gesture recognizer to handle swipes
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc] init];
    [pan addTarget:self action:@selector(pan:)];
    [self.view addGestureRecognizer:pan];

    //pause all layer animations
    self.doorLayer.speed = 0.0;

    //apply swinging animation (which won't play because layer is paused)
    CABasicAnimation *animation = [CABasicAnimation animation];
    animation.keyPath = @"transform.rotation.y";
    animation.toValue = @(-M_PI_2);
    animation.duration = 1.0;
    [self.doorLayer addAnimation:animation forKey:nil];
}

- (void)pan:(UIPanGestureRecognizer *)pan
{
    //get horizontal component of pan gesture
    CGFloat x = [pan translationInView:self.view].x;

    //convert from points to animation duration
    //using a reasonable scale factor
    x /= 200.0f;

    //update timeOffset and clamp result
    CFTimeInterval timeOffset = self.doorLayer.timeOffset;
    timeOffset = MIN(0.999, MAX(0.0, timeOffset - x));
    self.doorLayer.timeOffset = timeOffset;
}

```

```
//reset pan gesture
[pan setTranslation:CGPointZero inView:self.view];
}

@end
```

That's a neat trick, but you might be thinking that it would be easier to just set the door's `transform` directly using our pan gesture, rather than setting up an animation and then only displaying a single frame at a time.

That is true in this simple case, but for a more complex case such as a keyframe animation, or an animation group with several moving layers, this is actually a very simple way to scrub through the animation without having to manually calculate each property of every layer at a given point in time.

Summary

In this chapter, you learned about the `CAMediaTiming` protocol and the mechanisms that Core Animation uses to manipulate time for the purposes of controlling animations. In the next chapter, we cover *easing*, another time-manipulation technique used to make animations appear more natural.

This page intentionally left blank

10

Easing

In life, as in art, the beautiful moves in curves.

Edward G. Bulwer-Lytton

In Chapter 9, “Layer Time,” we discussed animation timing and the `CAMediaTiming` protocol. We now look at another time-related mechanism—a system known as *easing*. Core Animation uses easing to make animations move smoothly and naturally instead of seeming robotic and artificial, and in this chapter we explore how to control and customize the easing curves for your animations.

Animation Velocity

Animation is the change of a value over time, and that implies that the change must happen at a particular rate or *velocity*. The velocity of an animation is related to its duration by the following equation:

$$\text{velocity} = \text{change} / \text{time}$$

The *change* would be (for example) the distance that a moving object travels, and the *time* is the duration of the animation. This is easier to visualize for an animation that involves movement (such as animation of the `position` or `bounds` property), but it applies equally to any animatable property (such as `color` or `opacity`).

The equation above assumes that the velocity is constant throughout the animation (which was the case for the animations we created in Chapter 8, “Explicit Animations”). Using a constant velocity for an animation is known as *linear pacing*, and it’s the simplest way to implement animation, from a technical standpoint. It’s also *completely unrealistic*.

Consider a car driving a short distance. It would not start at 60 mph, drive to the destination and then immediately drop to 0 mph. For one thing, that would require infinite accelerating

capability (even the *best* sports car can't do 0 to 60 in 0 seconds), and for another, it would kill all the passengers. In reality, it would slowly accelerate up to full speed, then when it was nearing its destination, it would begin to slow down until it finally comes to a gentle stop.

What about a weight that is dropped onto a hard surface? It would start stationary, then continue to accelerate right up until it hits the surface, at which point it would stop suddenly (probably with a loud bang as its accumulated kinetic energy is turned into sound).

Every physical object in the real world accelerates and decelerates when it moves. So, how do we implement this kind of acceleration in our animations? One option is to use a *physics engine* to realistically model the friction and momentum of our animated objects, but this is overkill for most purposes. For an animated user interface, we just want some timing equations that make our layers move like plausible real-world objects, but which aren't too complicated to calculate. The name for these types of equations are *easing functions*, and fortunately Core Animation comes with a bunch of standard ones built in and ready to use.

CAMediaTimingFunction

To make use of easing functions, we need to set the `timingFunction` property of `CAAnimation`, which is an object of class `CAMediaTimingFunction`. We can also use the `+setAnimationTimingFunction:` method of `CATransaction` if we want to change the timing function for implicit animations.

There are a couple of ways to create a `CAMediaTimingFunction`. The simplest option is to call the `+timingFunctionWithName:` constructor method. This takes one of a number of possible name constants:

```
kCAMediaTimingFunctionLinear  
kCAMediaTimingFunctionEaseIn  
kCAMediaTimingFunctionEaseOut  
kCAMediaTimingFunctionEaseInEaseOut  
kCAMediaTimingFunctionDefault
```

The `kCAMediaTimingFunctionLinear` option creates a linear paced timing function, which is the same function that is used if you leave a `CAAnimation timingFunction` property with its default value of `nil`. Linear pacing makes sense when modeling something that accelerates almost instantaneously, and then doesn't slow down significantly until it arrives at its destination (for example, a bullet fired from the barrel a gun), but it's an odd choice for the default because it's rarely what you want for most animations.

The `kCAMediaTimingFunctionEaseIn` constant creates a function that starts slow and gradually accelerates up to full speed before stopping suddenly. This is a good fit for something like the dropped weight example we mentioned earlier, or a missile launched at a target.

The `kCAMediaTimingFunctionEaseOut` constant does the opposite; it starts off at full speed and then gradually slows to a stop. This has a sort of *damping* effect, and is good for

representing something like a door that starts to swing closed and then slows to a gradual stop instead of slamming shut.

The `kCAMediaTimingFunctionEaseInEaseOut` constant creates a gradual acceleration up to full speed and then a smooth deceleration back down to a stop. This is generally how most real-world objects move, and is the best choice for most animations. If you could only ever use one easing function, it would be this one. Given this fact, you might wonder why this isn't the default, and in fact when you use `UIView` animation methods, this *is* the default, but when creating a `CAAnimation`, you will need to specify it yourself.

Finally, we have `kCAMediaTimingFunctionDefault`, which is very similar to `kCAMediaTimingFunctionEaseInEaseOut`, but creates a slightly more rapid initial acceleration up to full speed, followed by a slightly more gradual deceleration. The difference between this and `kCAMediaTimingFunctionEaseInEaseOut` is almost imperceptible, but Apple presumably felt that this was a better choice as the default for implicit animations (although they subsequently changed their mind for `UIKit`, which uses `kCAMediaTimingFunctionEaseInEaseOut` as the default instead). Remember that despite the name, this is *not* the default value when creating an *explicit* `CAAnimation`, it is only used as the default for *implicit* animations. (In other words, the default layer action animations use `kCAMediaTimingFunctionDefault` as their timing function.)

You can try out these different easing functions using a simple test project (see Listing 10.1). Just adjust the timing function in the code before running the project, and then tap anywhere to see the layer move with the specified easing.

Listing 10.1 Simple Project for Testing Easing Functions

```
@interface ViewController : UIViewController

@property (nonatomic, strong) CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a red layer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(0, 0, 100, 100);
    self.colorLayer.position = CGPointMake(self.view.bounds.size.width/2.0,
                                          self.view.bounds.size.height/2.0);
    self.colorLayer.backgroundColor = [UIColor redColor].CGColor;
    [self.view.layer addSublayer:self.colorLayer];
}

}
```

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //configure the transaction
    [CATransaction begin];
    [CATransaction setAnimationDuration:1.0];
    [CATransaction setAnimationTimingFunction:
     [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseOut]];

    //set the position
    self.colorLayer.position = [[touches anyObject] locationInView:self.view];

    //commit transaction
    [CATransaction commit];
}

@end

```

UIView Animation Easing

UIKit's animation methods also support the use of easing functions, although the syntax and constants are different. To change the easing for the `UIView` animation methods, add one of the following constants to the `options` parameter for the animation:

```

UIViewAnimationOptionCurveEaseInOut
UIViewAnimationOptionCurveEaseIn
UIViewAnimationOptionCurveEaseOut
UIViewAnimationOptionCurveLinear

```

These correspond directly to their `CAMediaTimingFunction` counterparts. `UIViewAnimationOptionCurveEaseInOut` is the default value that will be used unless you specify otherwise. (There is no counterpart in UIKit for `kCAMediaTimingFunctionDefault`.)

See Listing 10.2 for how these functions are used. (Note that we have switched from using a hosted layer to a `UIView` in this example because UIKit animation doesn't work with hosted layers.)

Listing 10.2 Easing Test Project Converted to Use UIKit Animation

```

@interface ViewController : UIViewController

@property (nonatomic, strong) UIView *colorView;

@end

```

```

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create a red layer
    self.colorView = [[UIView alloc] init];
    self.colorView.bounds = CGRectMake(0, 0, 100, 100);
    self.colorView.center = CGPointMake(self.view.bounds.size.width / 2,
                                        self.view.bounds.size.height / 2);
    self.colorView.backgroundColor = [UIColor redColor];
    [self.view addSubview:self.colorView];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //perform the animation
    [UIView animateWithDuration:1.0
                          delay:0.0
                        options:UIViewAnimationOptionCurveEaseOut
                      animations:^{

        //set the position
        self.colorView.center =
        [[touches anyObject] locationInView:self.view];

    } completion:NULL];
}

@end

```

Easing and Keyframe Animations

You may recall that the color switching keyframe animation from Chapter 8 (refer to Listing 8.5) looked a bit odd due to the linear pacing between colors, which made the transition between them occur in an unnatural fashion. To correct that, we can apply a more appropriate easing function such as `kCAMediaTimingFunctionEaseIn`, which will add a slight *pulse* effect as the layer changes color—more like a colored light bulb would behave in real life.

We don't want to apply the function uniformly across the whole animation, though; we want to repeat the easing for each animation step so that each color transition pulses in turn.

`CAKeyframeAnimation` has a `timingFunctions` property, which is an `NSArray`. We can use this to specify a different timing function for each step in the animation. The number of functions specified must be equal to the number of items in the `keyframes` array *minus one*, because the function describes the animation velocity *between* each pair of keyframes.

In this case, we actually want to use the same easing function throughout, but we still need to provide an array of functions so that the animation knows that the function should be repeated for each step instead of applied once across the whole sequence. We simply use an array containing multiple copies of the same function (see Listing 10.3).

If you run the updated project, you will find that the animation now looks a bit more natural.

Listing 10.3 Using `CAMediaTimingFunction` with `CAKeyframeAnimation`

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) IBOutlet CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    self.colorLayer = [CALayer layer];
    self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;

    //add it to our view
    [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
    //create a keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"backgroundColor";
    animation.duration = 2.0;
    animation.values = @[
        (__bridge id)[UIColor blueColor].CGColor,
        (__bridge id)[UIColor redColor].CGColor,
        (__bridge id)[UIColor greenColor].CGColor,
    ];
}
```

```

        (_bridge id) [UIColor blueColor].CGColor
    ];

//add timing function
CAMediaTimingFunction *fn = [CAMediaTimingFunction functionWithName:
                             kCAMediaTimingFunctionEaseIn];
animation.timingFunctions = @[fn, fn, fn];

//apply animation to layer
[self.colorLayer addAnimation:animation forKey:nil];
}

@end

```

Custom Easing Functions

In Chapter 8, we updated our clock project to include animation. That looks good, but it would be even better with the right easing function. When a real-life analog clock’s hand moves, it usually starts slowly then suddenly snaps into position before easing to a stop at the last moment. None of the standard easing functions is *quite* right for the effect we want though. How can we create a new one?

In addition to the `+functionWithName:` constructor, `CAMediaTimingFunction` also has an alternative constructor method, `+functionWithControlPoints::::`, that has four floating-point arguments. (Note the bizarre method syntax, which doesn’t include separate names for each argument. This is perfectly legal in objective-C, but contravenes Apple’s own guidelines for method naming, and seems like a curious design choice.)

Using this method, we can construct a custom easing function that is ideally suited to our clock animation. To understand how to use this method, though, we must learn a bit more about how `CAMediaTimingFunction` works.

The Cubic Bézier Curve

The basic principle of `CAMediaTimingFunction` function is that it transforms an input time into a proportional change between a start and end value. We can represent these as a simple graph, with time (t) on the x axis and change (δ) on the y axis. The graph for linear easing is therefore a simple diagonal from the origin (see Figure 10.1).

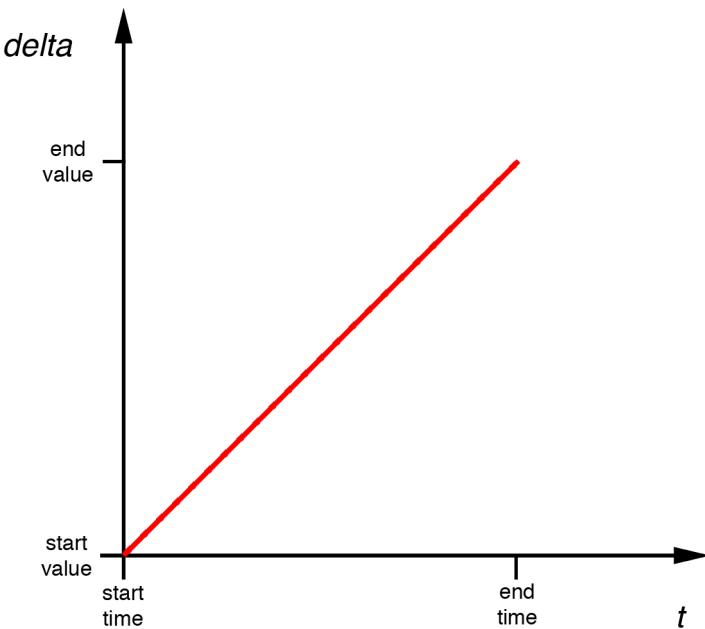


Figure 10.1 A graph of the linear easing function

The slope of this line represents the velocity of change. Changes in the slope represent acceleration or deceleration. In principle, any sort of acceleration curve can be represented on a graph like this, but `CAMediaTimingFunction` uses a specific function known as a *cubic Bézier curve*, which can only produce a specific subset of easing functions. (We previously encountered cubic Bézier curves in Chapter 8 when we used them to create a `CAKeyframeAnimation` path.)

As you may recall, a cubic Bézier curve is defined by four points: The first and last points indicate the starting and ending points for the curve, and the two middle points are called *control points*, because they control the shape of the curve. The control-points of a Bézier curve are *off-curve* points, meaning that the curve does not necessarily pass through them. You can think of these points as acting like magnets that *attract* the curve as it passes them.

Figure 10.2 shows an example of a cubic Bézier easing function.

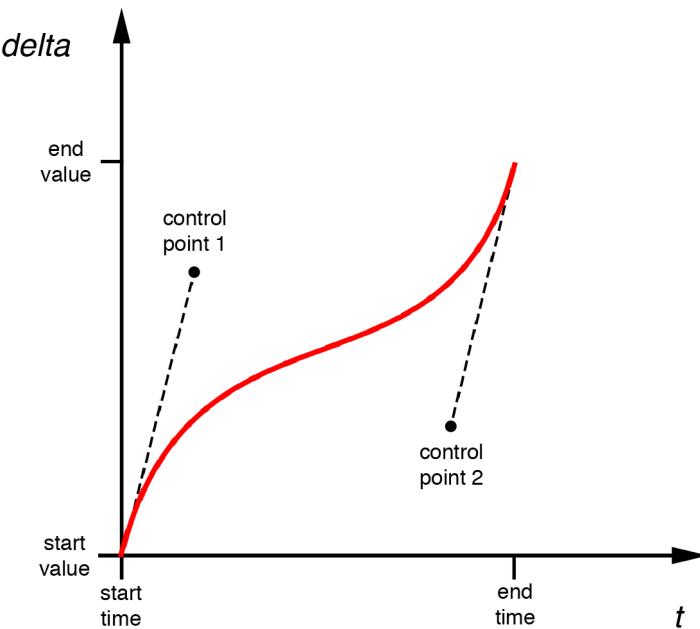


Figure 10.2 A cubic Bézier easing function

This would be a rather strange function in practice—it goes quickly at first, then slows down, then speeds up at the end. So how do the standard easing functions look when represented in graph form?

CAMediaTimingFunction has a method `-getControlPointAtIndex:values:` that can be used to retrieve the curve points. The design of this method is a bit quirky (why it doesn't simply return a `CGPoint` is a mystery that only Apple can answer) but using it we can find out the points for the standard easing functions and then plot them using `UIBezierPath` and `CAShapeLayer`.

The start and end points of the curve are always at $\{0, 0\}$ and $\{1, 1\}$, respectively, so we only need to retrieve the second and third points of the curve (the control points). The code to do this is shown in Listing 10.4. The graphs of all of the standard easing functions are shown in Figure 10.3.

Listing 10.4 Graphing a CAMediaTimingFunction Using UIBezierPath

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

```

```

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create timing function
    CAMediaTimingFunction *function =
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseOut];

    //get control points
    CGPoint controlPoint1, controlPoint2;
    [function getControlPointAtIndex:1 values:(float *)&controlPoint1];
    [function getControlPointAtIndex:2 values:(float *)&controlPoint2];

    //create curve
    UIBezierPath *path = [[UIBezierPath alloc] init];
    [path moveToPoint:CGPointZero];
    [path addCurveToPoint:CGPointMake(1, 1)
        controlPoint1:controlPoint1
        controlPoint2:controlPoint2];

    //scale the path up to a reasonable size for display
    [path applyTransform:CGAffineTransformMakeScale(200, 200)];

    //create shape layer
    CAShapeLayer *shapeLayer = [CAShapeLayer layer];
    shapeLayer.strokeColor = [UIColor redColor].CGColor;
    shapeLayer.fillColor = [UIColor clearColor].CGColor;
    shapeLayer.lineWidth = 4.0f;
    shapeLayer.path = path.CGPath;
    [self.layerView.layer addSublayer:shapeLayer];

    //flip geometry so that 0,0 is in the bottom-left
    self.layerView.layer.geometryFlipped = YES;
}

@end

```

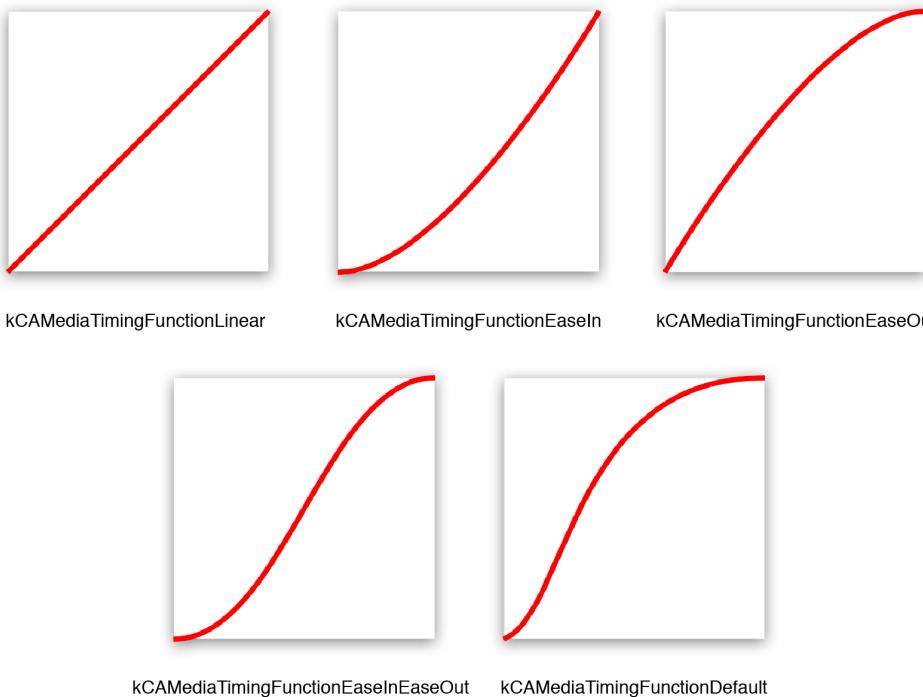


Figure 10.3 The standard `CAMediaTimingFunction` easing curves

For our custom clock hand easing function, we want a shallow curve initially, then a steep curve right up until the last second where it eases off. With a bit of experimentation, we end up with this:

```
[CAMediaTimingFunction functionWithControlPoints:1 :0 :0.75 :1];
```

If we drop this into our timing function graphing app, we get the curve shown in Figure 10.4. If we add this into the clock program, we have the nice *tick* effect we were looking for (see Listing 10.5).

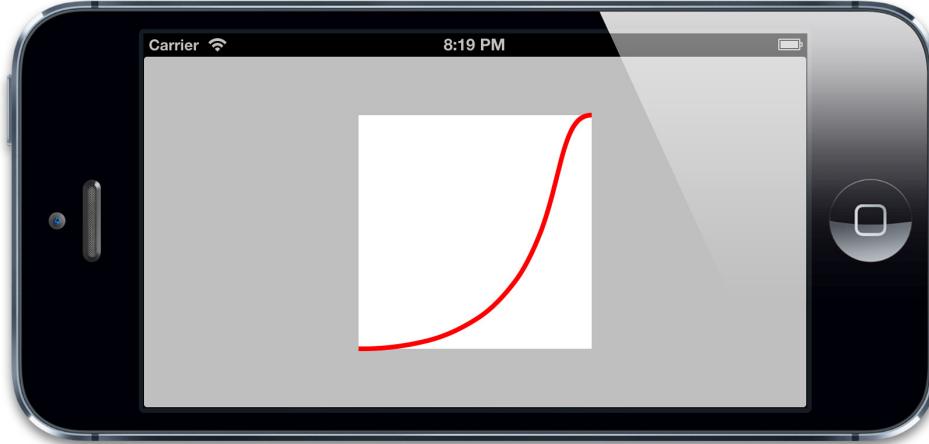


Figure 10.4 A custom easing function, suitable for a clock tick

Listing 10.5 The Clock Program with Custom Easing Function Added

```
- (void)setAngle:(CGFloat)angle
    forHand:(UIView *)handView
    animated:(BOOL)animated
{
    //generate transform
    CATransform3D transform = CATransform3DMakeRotation(angle, 0, 0, 1);

    if (animated)
    {
        //create transform animation
        CABasicAnimation *animation = [CABasicAnimation animation];
        animation.keyPath = @"transform";
        animation.fromValue =
            [handView.layer.presentationLayer valueForKey:@"transform"];
        animation.toValue = [NSValue valueWithCATransform3D:transform];
        animation.duration = 0.5;
        animation.delegate = self;
        animation.timingFunction =
            [CAMediaTimingFunction functionWithControlPoints:1 :0 :0.75 :1];

        //apply animation
        handView.layer.transform = transform;
        [handView.layer addAnimation:animation forKey:nil];
    }
}
```

```
        }
    else
    {
        //set transform directly
        handView.layer.transform = transform;
    }
}
```

More Complex Animation Curves

Consider a rubber ball dropped onto a hard surface: When dropped, it will accelerate until it hits the ground, bounce several times, and then eventually come to a stop. If we were to represent this with a graph, it would look something like Figure 10.5.

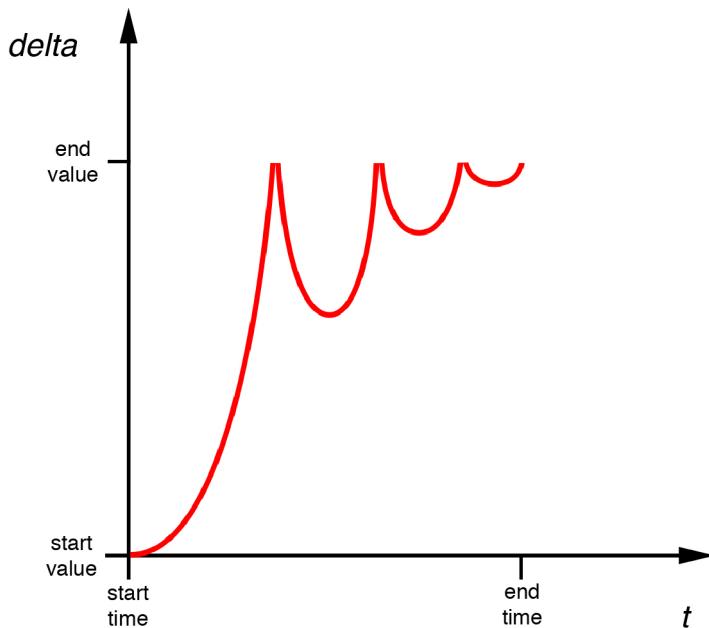


Figure 10.5 A bounce animation; not possible with a cubic Bézier curve

This sort of effect cannot be represented with a single cubic Bézier curve, and therefore cannot be achieved using `CAMediaTimingFunction`. If you want an effect like this, you have a couple of options:

- You can create the animation using a `CAKeyframeAnimation`, by splitting your animation into several steps, each with its own timing function (described in the next section).
- You can implement the animation yourself using a timer to update each frame (as explored in Chapter 11, “Timer-Based Animation”).

Keyframe-Based Easing

To implement our bounce using keyframes, we need to create a keyframe for each significant point in the easing curve (in this case, the peak and trough of each bounce) and then apply an easing function that matches up with that segment of the graph. We also need to specify the time offset for each keyframe using the `keyTimes` property, as the time between bounces will decrease each time, so the keyframes will not be evenly spaced.

Listing 10.6 shows the code for implementing a bouncing ball animation using this approach (see Figure 10.6).

Listing 10.6 Implementing a Bouncing Ball Animation Using Keyframes

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) UIImageView *ballView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add ball image view
    UIImage *ballImage = [UIImage imageNamed:@"Ball.png"];
    self.ballView = [[UIImageView alloc] initWithImage:ballImage];
    [self.containerView addSubview:self.ballView];

    //animate
    [self animate];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //replay animation on tap
    [self animate];
}
```

```
}

- (void)animate
{
    //reset ball to top of screen
    self.ballView.center = CGPointMake(150, 32);

    //create keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"position";
    animation.duration = 1.0;
    animation.delegate = self;
    animation.values = @[
        [NSValue valueWithCGPoint:CGPointMake(150, 32)],
        [NSValue valueWithCGPoint:CGPointMake(150, 268)],
        [NSValue valueWithCGPoint:CGPointMake(150, 140)],
        [NSValue valueWithCGPoint:CGPointMake(150, 268)],
        [NSValue valueWithCGPoint:CGPointMake(150, 220)],
        [NSValue valueWithCGPoint:CGPointMake(150, 268)],
        [NSValue valueWithCGPoint:CGPointMake(150, 250)],
        [NSValue valueWithCGPoint:CGPointMake(150, 268)]
    ];
    animation.timingFunctions = @[
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseIn],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseOut],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseIn],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseOut],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseIn],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseOut],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseIn],
        [CAMediaTimingFunction functionWithName:
            kCAMediaTimingFunctionEaseIn]
    ];
    animation.keyTimes = @[@0.0, @0.3, @0.5, @0.7, @0.8, @0.9, @0.95, @1.0];

    //apply animation
    self.ballView.layer.position = CGPointMake(150, 268);
    [self.ballView.layer addAnimation:animation forKey:nil];
}
```

```
}
```

```
@end
```

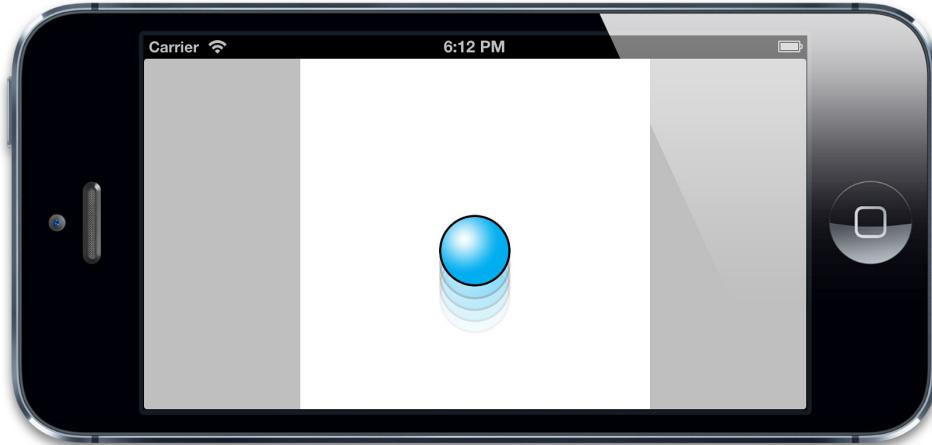


Figure 10.6 A bouncing ball animation implemented using keyframes

This approach works reasonably well, but it's cumbersome (calculating the keyframes and keytimes is basically a matter of trial and error) and highly animation specific (if you change any properties of the animation, you need to recalculate all the keyframes). What would be really useful would be if we could write a method to convert any simple property animation into a keyframe animation with an arbitrary easing function. Let's do that.

Automating the Process

In Listing 10.6, we split the animation into fairly large sections and used Core Animation's ease-in and ease-out functions to approximate the curve we wanted. But if we split the animation into smaller parts, then we can approximate any sort of curve just by using straight lines (a.k.a linear easing). To automate this, we need to figure out how to do two things:

- Automatically split an arbitrary property animation into multiple keyframes
- Represent our bounce animation as a mathematical function that we can use to offset our frames

To solve the first problem, we need to replicate Core Animation's interpolation mechanism. This is the algorithm for taking a start and end value and producing a new value at a specific point in time between those two. For simple floating-point start/end values, the formula for this (assuming that time is normalized to a value between 0 and 1) is as follows:

$$\text{value} = (\text{endValue} - \text{startValue}) \times \text{time} + \text{startValue};$$

To interpolate a more complex value type such as a `CGPoint`, `CGColorRef`, or `CATransform3D`, we can simply apply this function to each individual element (that is, the x and y values in a `CGPoint`, the red, green, blue, and alpha values in a `CGColorRef`, or the individual matrix coordinates in a `CATransform3D`). We also need some logic to unbox the values from an object type before interpolating, and rebox it afterward, which means inspecting the type at runtime.

Once we have the code to get an arbitrary intermediate value between the start and end values of our property animation, we can quite easily split our animation into as many individual keyframes as we like and produce a linear keyframe animation. Listing 10.7 shows the code to do this.

Note that we have used $60 \times$ the animation duration (in seconds) as the number of keyframes. This is because Core Animation renders updates to the screen at 60 frames per second, so if we generate 60 keyframes per second, we guarantee that the animation is smooth (although in practice we could probably get away with using fewer frames and still get a good result).

We have only included the code to interpolate `CGPoint` values in the example. However, it should be clear from the code how this could be extended to handle other types. As a fallback for types we don't recognize, we simply return the `fromValue` for the first half of the animation and the `toValue` for the second half.

Listing 10.7 Creating a Keyframe Animation by Interpolating Values

```
float interpolate(float from, float to, float time)
{
    return (to - from) * time + from;
}

- (id)interpolateFromValue:(id)fromValue
                      toValue:(id)toValue
                        time:(float)time
{
    if ([fromValue isKindOfClass:[NSValue class]])
    {
        //get type
        const char *type = [fromValue objCType];
        if (strcmp(type, @encode(CGPoint)) == 0)
        {
            CGPoint from = [fromValue CGPointValue];
```

```

        CGPoint to = [toValue CGPointValue];
        CGPoint result = CGPointMake(interpolate(from.x, to.x, time),
                                      interpolate(from.y, to.y, time));
        return [NSValue valueWithCGPoint:result];
    }
}

//provide safe default implementation
return (time < 0.5)? fromValue: toValue;
}

- (void)animate
{
    //reset ball to top of screen
    self.ballView.center = CGPointMake(150, 32);

    //set up animation parameters
    NSValue *fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
    NSValue *toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];
    CFTimeInterval duration = 1.0;

    //generate keyframes
    NSInteger numFrames = duration * 60;
    NSMutableArray *frames = [NSMutableArray array];
    for (int i = 0; i < numFrames; i++)
    {
        float time = 1/(float)numFrames * i;
        [frames addObject:[self interpolateFromValue:fromValue
                                              toValue:toValue
                                              time:time]];
    }

    //create keyframe animation
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.keyPath = @"position";
    animation.duration = 1.0;
    animation.delegate = self;
    animation.values = frames;

    //apply animation
    [self.ballView.layer addAnimation:animation forKey:nil];
}

```

That works, but it's not especially impressive. All we have really achieved so far is a very complicated way to replicate the behavior of a `CABasicAnimation` with linear easing. The advantage of this approach, however, is that we are now in a position to precisely control the easing, which means we can apply a completely bespoke easing function. So, how do we create one?

The math behind easing is nontrivial, but fortunately we don't need to implement it from first principles. Robert Penner has a web page devoted to easing functions (<http://www.robertpenner.com/easing>) that contains links to public domain code samples for all common (and some less-common) easing functions in multiple programming languages, including C. Here is an example of an ease-in-ease-out function. (There are actually several different ways to implement ease-in-ease-out.)

```
float quadraticEaseInOut(float t)
{
    return (t < 0.5)? (2 * t * t) : (-2 * t * t) + (4 * t) - 1;
}
```

For our bouncing ball animation, we will be using the `bounceEaseOut` function:

```
float bounceEaseOut(float t)
{
    if (t < 4/11.0)
    {
        return (121 * t * t)/16.0;
    }
    else if (t < 8/11.0)
    {
        return (363/40.0 * t * t) - (99/10.0 * t) + 17/5.0;
    }
    else if (t < 9/10.0)
    {
        return (4356/361.0 * t * t) - (35442/1805.0 * t) + 16061/1805.0;
    }
    return (54/5.0 * t * t) - (513/25.0 * t) + 268/25.0;
}
```

If we modify our code from Listing 10.7 to include the `bounceEaseOut` function, our task is complete: Just by swapping the easing function, we can now create an animation with any easing type we choose (see Listing 10.8).

Listing 10.8 Implementing a Custom Easing Function Using Keyframes

```
- (void)animate
{
    //reset ball to top of screen
    self.ballView.center = CGPointMake(150, 32);
```

```

//set up animation parameters
NSValue *fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
NSValue *toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];
CFTimeInterval duration = 1.0;

//generate keyframes
NSInteger numFrames = duration * 60;
NSMutableArray *frames = [NSMutableArray array];
for (int i = 0; i < numFrames; i++)
{
    float time = 1/(float)numFrames * i;

    //apply easing
    time = bounceEaseOut(time);

    //add keyframe
    [frames addObject:[self interpolateFromValue:fromValue
                                             toValue:toValue
                                             time:time]];
}

//create keyframe animation
CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
animation.keyPath = @"position";
animation.duration = 1.0;
animation.delegate = self;
animation.values = frames;

//apply animation
[self.ballView.layer addAnimation:animation forKey:nil];
}

```

Summary

In this chapter, you learned about easing, and the `CAMediaTimingFunction` class, which allows us to create custom easing functions to fine-tune our animations. You also learned how to use `CAKeyframeAnimation` to bypass the limitations of `CAMediaTimingFunction` and create your own completely bespoke easing functions.

In the next chapter, we look at timer-based animation—an alternative approach to animation that gives us more control and makes it possible to manipulate animations on-the-fly.

Timer-Based Animation

I can guide you, but you must do exactly as I say.

Morpheus, *The Matrix*

In Chapter 10, “Easing,” we looked at `CAMediaTimingFunction`, which enables us to control the easing of animations to add realism by simulating physical effects such as acceleration and deceleration. But what if we want to simulate even more realistic physical interactions or to modify our animation on-the-fly in response to user input? In this chapter, we explore timer-based animations that allow us to precisely control how our animation will behave on a frame-by-frame basis.

Frame Timing

Animation appears to show continuous movement, but in reality that’s impossible when the display pixels are at fixed locations. Ordinary displays cannot display continuous movement; all they can do is display a sequence of static images fast enough that you perceive it as motion.

We mentioned previously that iOS refreshes the screen 60 times per second. What `CAAnimation` does is calculate a new frame to display and then draws it in sync with each screen update. Most of the cleverness of `CAAnimation` is in doing the interpolation and easing calculations to work out what to display each time.

In Chapter 10, we worked out how to do the interpolation and easing ourselves, and then essentially told a `CAKeyframeAnimation` instance exactly what to draw by providing an array of frames to be displayed. All Core Animation was doing for us at that point was displaying those frames in sequence. Surely we can do that part ourselves as well?

NSTimer

Actually, we already did something like that in Chapter 3, “Layer Geometry,” with the clock example, when we used an `NSTimer` to animate the hand movement. In that example, the hand only updated once per second, but the principle is really no different if we speed up the timer to fire 60 times per second instead.

Let’s try modifying the bouncing ball animation from Chapter 10 to use an `NSTimer` instead of a `CAKeyframeAnimation`. Because we will now be calculating the animation frames continuously as the timer fires (instead of in advance), we need some additional properties in our class to store the animation’s `fromValue`, `toValue`, `duration`, and the current `timeOffset` (see Listing 11.1).

Listing 11.1 Implementing the Bouncing Ball Animation Using NSTimer

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) UIImageView *ballView;
@property (nonatomic, strong) NSTimer *timer;
@property (nonatomic, assign) NSTimeInterval duration;
@property (nonatomic, assign) NSTimeInterval timeOffset;
@property (nonatomic, strong) id fromValue;
@property (nonatomic, strong) id toValue;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //add ball image view
    UIImage *ballImage = [UIImage imageNamed:@"Ball.png"];
    self.ballView = [[UIImageView alloc] initWithImage:ballImage];
    [self.containerView addSubview:self.ballView];

    //animate
    [self animate];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //replay animation on tap
    [self animate];
}
```

```

}

float interpolate(float from, float to, float time)
{
    return (to - from) * time + from;
}

- (id) interpolateFromValue:(id)fromValue
    toValue:(id)toValue
    time:(float)time
{
    if ([fromValue isKindOfClass:[NSValue class]])
    {
        //get type
        const char *type = [(NSValue *)fromValue objCType];
        if (strcmp(type, @encode(CGPoint)) == 0)
        {
            CGPoint from = [fromValue CGPointValue];
            CGPoint to = [toValue CGPointValue];
            CGPoint result = CGPointMake(interpolate(from.x, to.x, time),
                                         interpolate(from.y, to.y, time));
            return [NSValue valueWithCGPoint:result];
        }
    }

    //provide safe default implementation
    return (time < 0.5)? fromValue: toValue;
}

float bounceEaseOut(float t)
{
    if (t < 4/11.0)
    {
        return (121 * t * t)/16.0;
    }
    else if (t < 8/11.0)
    {
        return (363/40.0 * t * t) - (99/10.0 * t) + 17/5.0;
    }
    else if (t < 9/10.0)
    {
        return (4356/361.0 * t * t) - (35442/1805.0 * t) + 16061/1805.0;
    }
    return (54/5.0 * t * t) - (513/25.0 * t) + 268/25.0;
}

```

```
- (void)animate
{
    //reset ball to top of screen
    self.ballView.center = CGPointMake(150, 32);

    //configure the animation
    self.duration = 1.0;
    self.timeOffset = 0.0;
    self.fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
    self.toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];

    //stop the timer if it's already running

    [self.timer invalidate];

    //start the timer
    self.timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0
                                                target:self
                                              selector:@selector(step:)
                                             userInfo:nil
                                              repeats:YES];
}

- (void)step:(NSTimer *)step
{
    //update time offset
    self.timeOffset = MIN(self.timeOffset + 1/60.0, self.duration);

    //get normalized time offset (in range 0 - 1)
    float time = self.timeOffset / self.duration;

    //apply easing
    time = bounceEaseOut(time);

    //interpolate position
    id position = [self interpolateFromValue:self.fromValue
                                         toValue:self.toValue
                                           time:time];

    //move ball view to new position
    self.ballView.center = [position CGPointValue];

    //stop the timer if we've reached the end of the animation
    if (self.timeOffset >= self.duration)
    {
        [self.timer invalidate];
```

```
        self.timer = nil;
    }
}

@end
```

That works pretty well, and is roughly the same amount of code as the keyframe-based version. But there are some problems with this approach that would become apparent if we tried to animate a lot of things onscreen at once.

An `NSTimer` is not the optimal way to draw stuff to the screen that needs to refresh every frame. To understand why, we need to look at exactly how `NSTimer` works. Every thread on iOS maintains an `NSRunLoop`, which in simple terms is a loop that endlessly works through a list of tasks it needs to perform. For the main thread, these tasks might include the following:

- Processing touch events
- Sending and receiving network packets
- Executing code scheduled using GCD (Grand Central Dispatch)
- Handling timer actions
- Redrawing the screen

When you set up an `NSTimer`, it gets inserted into this task list with an instruction that it must not be executed until *at least* the specified time has elapsed. There is no upper limit on how long a timer may wait before it fires; it will only happen after the previous task in the list has finished. This will usually be within a few milliseconds of the scheduled time, but it might take longer if the previous task is slow to complete.

The screen redrawing is scheduled to happen every sixtieth of a second, but just like a timer action, it may get delayed by an earlier task in the list that takes too long to execute. Because these delays are effectively random, the result is that it is impossible to guarantee that a timer scheduled to fire every sixtieth of a second will always fire before the screen is redrawn. Sometimes it will happen too late, resulting in a delay in the update that will make the animation appear choppy. Sometimes it may fire twice between screen updates, resulting in a skipped frame that will make the animation appear to jump forward.

We can do some things to improve this:

- We can use a special type of timer called `CADisplayLink` that is designed to fire in lockstep with the screen refresh.
- We can base our animations on the *actual* recorded frame duration instead of assuming that frames will fire on time.

- We can adjust the *run loop mode* of our animation timer so that it won't be delayed by other events.

CADisplayLink

`CADisplayLink` is an `NSTimer`-like class provided by Core Animation that always fires immediately prior to the screen being redrawn. Its interface is very similar to `NSTimer`, so it is essentially a drop-in replacement, but instead of a `timeInterval` specified in seconds, the `CADisplayLink` has an integer `frameInterval` property that specifies how many frames to skip each time it fires. By default, this has a value of 1, meaning that it should fire every frame. But if your animation code takes too long to reliably execute within a sixtieth of a second, you can specify a `frameInterval` of 2, meaning that your animation will update every other frame (30 frames per second) or 3, resulting in 20 frames per second, and so on.

Using a `CADisplayLink` instead of an `NSTimer` will produce a smoother animation by ensuring that the frame rate is as consistent as possible. But even `CADisplayLink` cannot guarantee that every frame will happen on schedule. A stray task or an event outside of your control (such as a resource-hungry background application) might still cause your animation to occasionally skip frames. When using an `NSTimer`, the timer will simply fire whenever it gets a chance, but `CADisplayLink` works differently: If it misses a scheduled frame, it will skip it altogether and update at the next scheduled frame time.

Measuring Frame Duration

Regardless of whether we use an `NSTimer` or `CADisplayLink`, we still need to handle the scenario where a frame takes longer to calculate than the expected time of one-sixtieth of a second. Because we cannot know the actual frame duration in advance, we have to measure it as it happens. We can do this by recording the time at the start of each frame using the `CACurrentMediaTime()` function, and then comparing it to the time recorded for the previous frame.

By comparing these times, we get an accurate frame duration measurement that we can use in place of the hard-coded one-sixtieth value for our timing calculations. Let's update our example with these improvements (see Listing 11.2).

Listing 11.2 Measuring Frame Duration for Smoother Animation

```
@interface ViewController : UIViewController  
  
@property (nonatomic, weak) IBOutlet UIView *containerView;  
@property (nonatomic, strong) UIImageView *ballView;  
@property (nonatomic, strong) CADisplayLink *timer;  
@property (nonatomic, assign) CFTimeInterval duration;  
@property (nonatomic, assign) CFTimeInterval timeOffset;
```

```

@property (nonatomic, assign) CFTimeInterval lastStep;
@property (nonatomic, strong) id fromValue;
@property (nonatomic, strong) id toValue;

@end

@implementation ViewController

...

- (void)animate
{
    //reset ball to top of screen
    self.ballView.center = CGPointMake(150, 32);

    //configure the animation
    self.duration = 1.0;
    self.timeOffset = 0.0;
    self.fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
    self.toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];

    //stop the timer if it's already running
    [self.timer invalidate];

    //start the timer
    self.lastStep = CACurrentMediaTime();
    self.timer = [CADisplayLink displayLinkWithTarget:self
                                                selector:@selector(step:)];
    [self.timer addToRunLoop:[NSRunLoop mainRunLoop]
                      forMode:NSTimerMode];
}

- (void)step:(CADisplayLink *)timer
{
    //calculate time delta
    CFTimeInterval thisStep = CACurrentMediaTime();
    CFTimeInterval stepDuration = thisStep - self.lastStep;
    self.lastStep = thisStep;

    //update time offset
    self.timeOffset = MIN(self.timeOffset + stepDuration, self.duration);

    //get normalized time offset (in range 0 - 1)
    float time = self.timeOffset / self.duration;
}

```

```

//apply easing
time = bounceEaseOut(time);

//interpolate position
id position = [self interpolateFromValue:self.fromValue
                                      toValue:self.toValue
                                         time:time];

//move ball view to new position
self.ballView.center = [position CGPointValue];

//stop the timer if we've reached the end of the animation
if (self.timeOffset >= self.duration)
{
    [self.timer invalidate];
    self.timer = nil;
}
}

@end

```

Run Loop Modes

Notice that when we create the `CADisplayLink`, we are required to specify a *run loop* and *run loop mode*. For the run loop, we've used the main run loop (the run loop hosted by the main thread) because any user interface updates should always be performed on the main thread. The choice of mode is less clear, though. Every task that is added to the run loop has a mode that determines its priority. To ensure that the user interface remains smooth at all times, iOS will give priority to user interface related tasks and may actually stop executing other tasks altogether for a brief time if there is too much UI activity.

A typical example of this is when you are scrolling using `UIScrollView`. During the scroll, redrawing the scrollview content takes priority over other tasks, so standard `NSTimer` and network events may not fire while this is happening. Some common choices for the run loop mode are as follows:

- `NSDefaultRunLoopMode`—The standard priority
- `NSRunLoopCommonModes`—High priority
- `UITrackingRunLoopMode`—Used for animating `UIScrollView` and other controls

In our example, we have used `NSDefaultRunLoopMode`, but to ensure that our animation runs smoothly, we could use `NSRunLoopCommonModes` instead. Just be cautious when using this mode because when your animation is running at a high frame rate, you may find that

other tasks such as timers or other iOS animations such as scrolling will stop updating until your animation has finished.

It's possible to use a `CADisplayLink` with multiple run loop modes at the same time, so we could add it to both `NSDefaultRunLoopMode` and `UITrackingRunLoopMode` to ensure that it is not disrupted by scrolling, without interfering with the performance of other UIKit control animations, like this:

```
self.timer = [CADisplayLink displayLinkWithTarget:self
                                         selector:@selector(step:)];
[self.timer addToRunLoop:[NSRunLoop mainRunLoop]
                     forMode:NSDefaultRunLoopMode];
[self.timer addToRunLoop:[NSRunLoop mainRunLoop]
                     forMode:UITrackingRunLoopMode];
```

Like `CADisplayLink`, `NSTimer` can also be configured to use different run loop modes by using this alternative setup code instead of the `+scheduledTimerWithTimeInterval:` constructor:

```
self.timer = [NSTimer timerWithTimeInterval:1/60.0
                                         target:self
                                         selector:@selector(step:)
                                         userInfo:nil
                                         repeats:YES];
[[NSRunLoop mainRunLoop] addTimer:self.timer
                           forMode:NSRunLoopCommonModes];
```

Physical Simulation

Although we've used our timer-based animation to replicate the behavior of the keyframe animation from Chapter 10, there is actually a fundamental difference in the way that it works: In the keyframe implementation, we had to calculate all the frames in advance, but in our new solution, we are calculating them on demand. The significance of this is that it means we can modify the animation logic *on-the-fly* in response to user input, or integrate with other real-time animation systems such as a *physics engine*.

Chipmunk

Instead of our current easing-based bounce animation, let's use physics to create a truly realistic gravity simulation. Simulating physics accurately—even in 2D—is incredibly complex, so we won't attempt to implement this from scratch. Instead, we'll use an open source physics library, or *engine*.

The physics engine we're going to use is called Chipmunk. Other 2D physics libraries are available (such as Box2D), but Chipmunk is written in pure C instead of C++, making it easier to integrate into an Objective-C project. Chipmunk comes in various flavors,

including an “indie” version with Objective-C bindings. The plain C version is free, though, so that’s what we will use for this example. Version 6.1.4 is the latest at the time of writing; you can download it from <http://chipmunk-physics.net>.

The Chipmunk physics engine as a whole is quite large and complex, but we will be using only the following classes:

- `cpSpace`—This is the container for all of the physics bodies. It has a size and (optionally) a gravity vector.
- `cpBody`—This is a solid, inelastic object. It has a position in space and other physical properties such as mass, moment, coefficient of friction, and so on.
- `cpShape`—This is an abstract geometric shape, used for detecting collisions. Multiple shapes may be attached to a body, and there are concrete subclasses of `cpShape` to represent different shape types.

In our example, we will model a wooden crate that falls under the influence of gravity. We will create a `Crate` class that encompasses both the visual representation of the crate onscreen (a `UIImageView`) and the physical model that represents it (a `cpBody` and `cpPolyShape`, which is a polygonal `cpShape` subclass that we will use to represent the rectangular crate).

Using the C version of Chipmunk introduces some challenges because it doesn’t support Objective-C’s reference counting model, so we need to explicitly create and free objects. To simplify this, we tie the lifespan of the `cpShape` and `cpBody` to our `Crate` class by creating them in the crate’s `-init` method and freeing them in `-dealloc`. The configuration of the crate’s physical properties is fairly complex, but it should make sense if you read the Chipmunk documentation.

The view controller will manage the `cpSpace`, along with the timer logic as before. At each step, we’ll update the `cpSpace` (which performs the physics calculations and repositions all the bodies in the world) and then iterate through the bodies and update the positions of our crate views to match the bodies that model them. (In this case, there is actually only one body, but we will add more later.)

Chipmunk uses an inverted coordinate system with respect to UIKit (the Y axis points upward). To make it easier to keep our physics model in sync with our view, we’ll invert our container view’s geometry using the `geometryFlipped` property (mentioned in Chapter 3) so that model and view are both using the same coordinate system.

The code for the crate example is shown in Listing 11.3. Note that we aren’t freeing the `cpSpace` object anywhere. In this simple example, the space will exist for the duration of the app lifespan anyway, so this isn’t really an issue but in a real-world scenario, we should really handle this in the same way we do with the crate body and shape, by wrapping it in standalone Cocoa object and using that to manage the Chipmunk object’s lifecycle. Figure 11.1 shows the falling crate.

Listing 11.3 Modeling a Falling Crate Using Realistic Physics

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>
#import "chipmunk.h"

@interface Crate : UIImageView

@property (nonatomic, assign) cpBody *body;
@property (nonatomic, assign) cpShape *shape;

@end

@implementation Crate

#define MASS 100

- (id)initWithFrame:(CGRect)frame
{
    if ((self = [super initWithFrame:frame]))
    {
        //set image
        self.image = [UIImage imageNamed:@"Crate.png"];
        self.contentMode = UIViewContentModeScaleAspectFill;

        //create the body
        self.body = cpBodyNew(MASS, cpMomentForBox(MASS, frame.size.width,
                                                    frame.size.height));

        //create the shape
        cpVect corners[] = {
            cpv(0, 0),
            cpv(0, frame.size.height),
            cpv(frame.size.width, frame.size.height),
            cpv(frame.size.width, 0),
        };
        self.shape = cpPolyShapeNew(self.body, 4, corners,
                                   cpv(-frame.size.width/2,
                                       -frame.size.height/2));

        //set shape friction & elasticity
        cpShapeSetFriction(self.shape, 0.5);
        cpShapeSetElasticity(self.shape, 0.8);

        //link the crate to the shape
        //so we can refer to crate from callback later on
    }
}
```

```

    self.shape->data = (_bridge void *)self;

    //set the body position to match view
    cpBodySetPos(self.body, cpv(frame.origin.x + frame.size.width/2,
                                300 - frame.origin.y - frame.size.height/2));

}

return self;
}

- (void)dealloc
{
    //release shape and body
    cpShapeFree(_shape);
    cpBodyFree(_body);
}

@end

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, assign) cpSpace *space;
@property (nonatomic, strong) CADisplayLink *timer;
@property (nonatomic, assign) CFTimeInterval lastStep;

@end

@implementation ViewController

#define GRAVITY 1000

- (void)viewDidLoad
{
    //invert view coordinate system to match physics
    self.containerView.layer.geometryFlipped = YES;

    //set up physics space
    self.space = cpSpaceNew();
    cpSpaceSetGravity(self.space, cpv(0, -GRAVITY));

    //add a crate
    Crate *crate = [[Crate alloc] initWithFrame:CGRectMake(100, 0, 100, 100)];
    [self.containerView addSubview:crate];
    cpSpaceAddBody(self.space, crate.body);
    cpSpaceAddShape(self.space, crate.shape);
}

```

```

//start the timer
self.lastStep = CACurrentMediaTime();
self.timer = [CADisplayLink displayLinkWithTarget:self
                                             selector:@selector(step:)];
[self.timer addToRunLoop:[NSRunLoop mainRunLoop]
                     forMode:NSTimerRunLoopMode];
}

void updateShape(cpShape *shape, void *unused)
{
    //get the crate object associated with the shape
Crate *crate = (__bridge Crate *)shape->data;

    //update crate view position and angle to match physics shape
cpBody *body = shape->body;
crate.center = cpBodyGetPos(body);
crate.transform = CGAffineTransformMakeRotation(cpBodyGetAngle(body));
}

- (void)step:(CADisplayLink *)timer
{
    //calculate step duration
CFTimeInterval thisStep = CACurrentMediaTime();
CFTimeInterval stepDuration = thisStep - self.lastStep;
self.lastStep = thisStep;

    //update physics
cpSpaceStep(self.space, stepDuration);

    //update all the shapes
cpSpaceEachShape(self.space, &updateShape, NULL);
}

@end

```

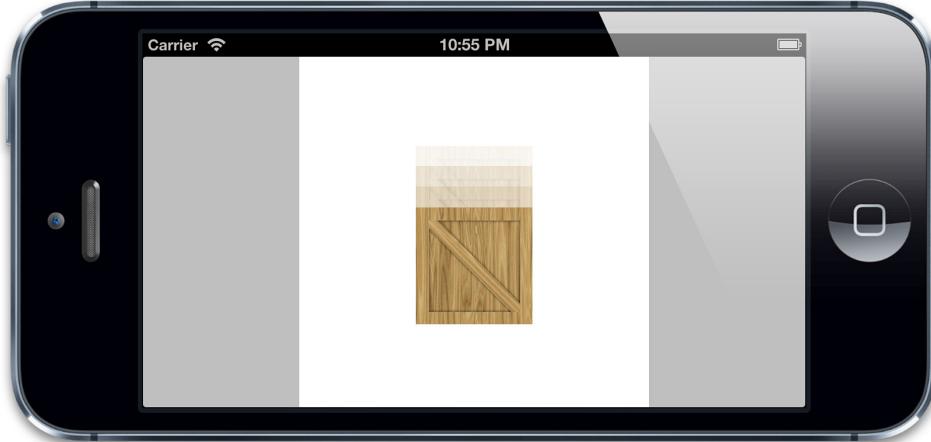


Figure 11.1 A crate image, falling due to simulated gravity

Adding User Interaction

The next step is to add an invisible wall around the edge of the view so that the crate doesn't fall off the bottom of the screen. You might think that we would implement this with another rectangular `cpPolyShape`, like we used for our crate, but we want to detect when our crate *leaves* the view, not when it is colliding with it, so we need a *hollow* rectangle, not a solid one.

We can implement this by adding four `cpSegmentShape` objects to our `cpSpace`. (`cpSegmentShape` represents a straight line segment, so four of them can be combined to form a rectangle.) We will attach these to the space's `staticBody` property (an immovable body that is not affected by gravity) instead of a new `cpBody` instance as we did with each crate, because we do not want the bounding rectangle to fall off the screen or be dislodged when hit by a falling crate.

We'll also add a few more crates so they can interact with one another. Finally, we'll add accelerometer support so that tilting the phone adjusts the gravity vector. (Note that to test this you will need to run on a real device because the simulator doesn't generate accelerometer events, even if you rotate the screen.) Listing 11.4 shows the updated code, and Figure 11.2 shows the result.

Because our example is locked to landscape mode, we've swapped the x and y values for the accelerometer vector. If you are running the example in portrait mode, you will need to swap them back to ensure the gravity direction matches up with what's displayed on the screen. You'll know if you get it wrong; the crates will fall upward or sideways!

Listing 11.4 Updated Physics Example with Walls and Multiple Crates

```
- (void)addCrateWithFrame:(CGRect)frame
{
    Crate *crate = [[Crate alloc] initWithFrame:frame];
    [self.containerView addSubview:crate];
    cpSpaceAddBody(self.space, crate.body);
    cpSpaceAddShape(self.space, crate.shape);
}

- (void)addWallShapeWithStart:(cpVect)start end:(cpVect)end
{
    cpShape *wall = cpSegmentShapeNew(self.space->staticBody, start, end, 1);
    cpShapeSetCollisionType(wall, 2);
    cpShapeSetFriction(wall, 0.5);
    cpShapeSetElasticity(wall, 0.8);
    cpSpaceAddStaticShape(self.space, wall);
}

- (void)viewDidLoad
{
    //invert view coordinate system to match physics
    self.containerView.layer.geometryFlipped = YES;

    //set up physics space
    self.space = cpSpaceNew();
    cpSpaceSetGravity(self.space, cpv(0, -GRAVITY));

    //add wall around edge of view
    [self addWallShapeWithStart:cpv(0, 0) end:cpv(300, 0)];
    [self addWallShapeWithStart:cpv(300, 0) end:cpv(300, 300)];
    [self addWallShapeWithStart:cpv(300, 300) end:cpv(0, 300)];
    [self addWallShapeWithStart:cpv(0, 300) end:cpv(0, 0)];

    //add a crates
    [self addCrateWithFrame:CGRectMake(0, 0, 32, 32)];
    [self addCrateWithFrame:CGRectMake(32, 0, 32, 32)];
    [self addCrateWithFrame:CGRectMake(64, 0, 64, 64)];
    [self addCrateWithFrame:CGRectMake(128, 0, 32, 32)];
    [self addCrateWithFrame:CGRectMake(0, 32, 64, 64)];

    //start the timer
    self.lastStep = CACurrentMediaTime();
    self.timer = [CADisplayLink displayLinkWithTarget:self
                                                selector:@selector(step:)];
    [self.timer addToRunLoop:[NSRunLoop mainRunLoop]
```

```

        forMode:NSUTFDefaultRunLoopMode];

//update gravity using accelerometer
[UIAccelerometer sharedAccelerometer].delegate = self;
[UIAccelerometer sharedAccelerometer].updateInterval = 1/60.0;
}

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
    //update gravity
    cpSpaceSetGravity(self.space, cpv(acceleration.y * GRAVITY,
                                    -acceleration.x * GRAVITY));
}

```



Figure 11.2 Multiple crates interacting with realistic physics

Simulation Time and Fixed Time Steps

Calculating the frame duration and using it to advance the animation was a good solution for the easing-based animation, but it's not ideal for advancing our physics simulation. Having a variable time step for a physics simulation is a bad thing for two reasons:

- If the time step is not set to a fixed, precise value, the physics simulation will not be *deterministic*. This means that given the same exact inputs it may produce different results on different occasions. Sometimes this doesn't matter, but in a physics-based game, players might be confused when the exact same actions on their part lead to different outcomes. It also makes testing more difficult.
- A skipped frame due to a performance glitch or an interruption like a phone call might cause incorrect behavior. Consider a fast-moving object like a bullet: Each frame, the simulation will move the bullet forward and check for collisions. If the time between frames increases, the bullet will move further in a single simulation step and may pass right through a wall or other obstacle without ever intersecting it, causing the collision to be missed.

What we ideally want is to have short *fixed* time steps for our physics simulation, but still update our views in sync with the display redraw (which may be unpredictable due to circumstances outside our control).

Fortunately, because our models (in this case, the `cpBody` objects in our Chipmunk `cpSpace`) are separated from our views (the `UIView` objects representing our crates onscreen), this is quite easy. We just need to keep track of the simulation step time independently of our display frame time, and potentially perform multiple simulation steps for each display frame.

We can do this using a simple loop. Each time our `CADisplayLink` fires to let us know the frame needs redrawing, we make a note of the `CACurrentMediaTime()`. We then advance our physics simulation repeatedly in small increments (1/120th of a second in this case) until the physics time catches up to the display time. We can then update our views to match the current positions of the physics bodies in preparation for the screen refresh.

Listing 11.5 shows the code for the fixed-step version of our crate simulation.

Listing 11.5 Fixed Time Step Crate Simulation

```
#define SIMULATION_STEP (1/120.0)

- (void)step:(CADisplayLink *)timer
{
    //calculate frame step duration
    CFTimeInterval frameTime = CACurrentMediaTime();

    //update simulation
    while (_self.lastStep < frameTime)
    {
        cpSpaceStep(_self.space, SIMULATION_STEP);
        _self.lastStep += SIMULATION_STEP;
    }
}
```

```
//update all the shapes  
cpSpaceEachShape (self.space, &updateShape, NULL) ;  
}
```

Avoiding the Spiral of Death

One thing you must ensure when using fixed simulation time steps is that the real-world time taken to perform the physics calculations does not exceed the simulated time step. In our example, we've chosen the arbitrary time step of 1/120th of a second for our physics simulation. Chipmunk is fast and our example is simple, so our `cpSpaceStep()` should complete well within that time and is unlikely to delay our frame update.

But suppose we had a more complex scene with hundreds of objects all interacting: The physics calculations would become more complex and the `cpSpaceStep()` might actually take more than 1/120th of a second to complete. We aren't measuring the time for our physics step because we are assuming that it will be trivial compared to the frame update, but if our simulation steps take longer to execute than the time period they are simulating, they will delay the frame update.

It gets worse: If the frame takes longer to update, our simulation will need to perform more steps to keep the simulated time in sync with real time. Those additional steps will then delay the frame update further, and so on. This is known as the *spiral of death* because the result is that the frame rate becomes slower and slower until the application effectively freezes.

We could add code to measure the real-world time for our physics step on a given device and adjust the fixed time step automatically, but in practice this is probably overkill. Just ensure that you leave a generous margin for error and test on the slowest device you intend to support. If the physics calculations take more than ~50% of the simulated time, consider increasing your simulation time step (or simplifying your scene). If your simulation time step increases to the point that it takes more than one-sixtieth of a second (an entire screen frame update), you will need to reduce your animation frame rate to 30 frames per second or less by increasing the `CADisplayLink frameInterval` so that you don't start randomly skipping frames, which will make your animation look unsmooth.

Summary

In this chapter, you learned how to create animations on a frame-by-frame basis using a timer combined with a variety of animation techniques including easing, physics simulation, and user input (via the accelerometer).

In Part III, we look at how animation performance is impacted by the constraints of the hardware, and learn how to tune our code to get the best frame rate possible.



The Performance of a Lifetime

12 Tuning for Speed

13 Efficient Drawing

14 Image IO

15 Layer Performance

This page intentionally left blank

Tuning for Speed

Code should run as fast as necessary, but no faster.

Richard E. Pattis

In Parts I and II, we learned about the awesome drawing and animation features that Core Animation has to offer. Core Animation is powerful and fast, but it's also easy to use inefficiently if you're not clear what's going on behind the scenes. There is an art to getting the best performance out of it. In this chapter, we explore some of the reasons why your animations might run slowly and how you can diagnose and fix problems.

CPU Versus GPU

There are two types of processor involved in drawing and animation: The CPU (central processing unit) and GPU (graphics processing unit). On modern iOS devices, these are both programmable chips that can run (more or less) arbitrary software, but for historical reasons, we tend to say that the part of the work that is performed by the CPU is done “in software” and the part handled by the GPU is done “in hardware.”

Generally speaking, we can do anything in software (using the CPU), but for graphics processing, it's usually much faster to use hardware because the GPU is optimized for the sort of highly parallel floating point math used in graphics. For this reason, we ideally want to offload as much of our screen rendering to hardware as possible. The problem is that the GPU does not have unlimited processing power, and once it's already being used to full capacity, performance will start to degrade (even if the CPU is not being fully utilized).

Most animation performance optimization is about intelligently utilizing the GPU and CPU so that neither is overstretched. To do that, we first have to understand how Core Animation divides the work between these processors.

The Stages of an Animation

Core Animation lies at the heart of iOS: It is used not just within applications but *between* them. A single animation may actually display content from multiple apps simultaneously, such as when you use gestures on an iPad to switch between apps, causing views from both to briefly appear onscreen at once. It wouldn't make sense for this animation to be performed inside the code of any particular app because then it wouldn't be possible for iOS to implement these kinds of effects. (Apps are sandboxed and cannot access each others' views.)

Animating and compositing layers onscreen is actually handled by a separate process, outside of your application. This process is known as the *render server*. On iOS 5 and earlier, this is the *SpringBoard* process (which also runs the iOS home screen). On iOS 6 and later, this is handled by a new process called *BackBoard*.

When you perform an animation, the work breaks down into four discrete phases:

- **Layout**—This is the phase where you prepare your view/layer hierarchy and set up the properties of the layers (frame, background color, border, and so on).
- **Display**—This is where the backing images of layers are drawn. That drawing may involve calling routines that you have written in your `-drawRect:` or `-drawLayer:inContext:` methods.
- **Prepare**—This is the phase where Core Animation gets ready to send the animation data to the render server. This is also the point at which Core Animation will perform other duties such as decompressing images that will be displayed during the animation (more on this later).
- **Commit**—This is the final phase, where Core animation packages up the layers and animation properties and sends them over IPC (Inter-Process Communication) to the render server for display.

But those are just the phases that take place inside your application—there is still more work to be done before the animation appears onscreen. Once the packaged layers and animations arrive in the render server process, they are deserialized to form another layer tree called the *render tree* (mentioned in Chapter 1, “The Layer Tree”). Using this tree, the render server does the following for each frame of the animation:

- Calculates the intermediate values for all the layer properties and sets up the OpenGL geometry (textured triangles) to perform the rendering
- Renders the visible triangles to the screen

So that's six phases in total; the last two are repeated over and over for the duration of the animation. The first five of these phases are handled in software (by the CPU), only the last is handled by the GPU. Furthermore, you only really have direct control of the first two phases: layout and display. The Core Animation framework handles the rest internally and you have no control over it.

That's not really a problem, because in the layout and display phases, you get to decide what work will be done upfront on the CPU and what will get passed down to the GPU. So how do you make that decision?

GPU-Bound Operations

The GPU is optimized for a specific task: It takes images and geometry (triangles), performs transforms, applies texturing and blending, and then puts them on the screen. The programmable GPUs in modern iOS devices allow for a lot of flexibility in how those operations are performed, but Core Animation does not expose a direct interface to any of that. Unless you are prepared to bypass Core Animation and start writing your own OpenGL shaders, you are basically stuck with a fixed set of things that are hardware accelerated, and everything else must be done in software on the CPU.

Broadly speaking, most properties of `CALayer` are drawn using the GPU. If you set the layer background or border colors, for example, those can be drawn really efficiently using colored triangles. If you assign an image to the `contents` property—even if you scale and crop it—it gets drawn using textured triangles, without the need for any software drawing.

A few things can slow down this (predominantly GPU-based) layer drawing, however:

- **Too much geometry**—This is where too many triangles need to be transformed and rasterized (turned into pixels) for the processor to cope. The graphics chip on modern iOS devices can handle millions of triangles, so geometry is actually unlikely to be a *GPU* bottleneck when it comes to Core Animation. But because of the way that layers must be preprocessed and sent to the render server via IPC before display (layers are fairly heavy objects, composed of several subobjects), too many layers will cause a *CPU* bottleneck. This limits the practical number of layers that you can display at once (see the section “CPU-Bound Operations,” later in this chapter).
- **Too much overdraw**—This is predominantly caused by overlapping semitransparent layers. GPU’s have a finite *fill-rate* (the rate at which they can fill pixels with color), so *overdraw* (filling the same pixel multiple times per frame) is something to be avoided. That said, modern iOS device GPUs are fairly good at coping with overdraw; even an iPhone 3GS can handle an overdraw ratio of 2.5 over the whole screen without dropping below 60 FPS (that means that you can draw one-and-a-half whole screenfuls of redundant information without impacting performance), and newer devices can handle more.
- **Offscreen drawing**—This occurs when a particular effect cannot be achieved by drawing directly onto the screen, but must instead be first drawn into an offscreen image context. Offscreen drawing is a generic term that may apply to either CPU or GPU-based drawing, but either way it involves allocating additional memory for the offscreen image and switching between drawing contexts, both of which will slow down the GPU performance. The use of certain layer effects, such as rounded corners, layer masks, drop shadows, or layer rasterization will force Core Animation to pre-

render the layer offscreen. That doesn't mean that you need to avoid these effects altogether, just that you need to be aware that they can have a performance impact.

- **Too-large images**—If you attempt to draw an image that is larger than the maximum texture size supported by the GPU (usually 2048×2048 or 4096×4096, depending on the device), the CPU must be used to preprocess the image each time it is displayed, slowing the performance right down.

CPU-Bound Operations

Most of the CPU work in Core Animation happens upfront before the animation begins. This is good because it means that it generally doesn't impact the frame rate, but it's bad because it can delay the start of an animation, making your interface appear unresponsive.

The following CPU operations can all slow down the start of your animation:

- **Layout calculations**—If your view hierarchy is complex, it might take a while to calculate all the layer frames when a view is presented or modified. This is especially true if you are using iOS 6's new autolayout mechanism, which is more CPU-intensive than the old autoresizing logic.
- **Lazy view loading**—iOS only loads a view controller's view when it is first displayed onscreen. This is good for memory usage and application startup time, but can be bad for responsiveness if a button tap suddenly results in a lot of work having to be done before anything can appear onscreen. If the controller gets its data from a database, or the view is loaded from a nib file, or contains images, this can also lead to *IO work*, which can be orders of magnitude slower than ordinary CPU operations (see the “IO-Bound Operations” section, later in this chapter).
- **Core Graphics drawing**—If you implement the `-drawRect:` method of your view, or the `-drawLayer:inContext:` method of your `CALayerDelegate`, you introduce a significant performance overhead even before you've actually drawn anything. To support arbitrary drawing into a layer's contents, Core Animation must create a backing image in memory equal in size to the view dimensions. Then, once the drawing is finished, it must transmit this image data via IPC to the render server. On top of that overhead, Core Graphics drawing is very slow anyway, and not something that you want to be doing in a performance critical situation.
- **Image decompression**—Compressed image files such as PNGs or JPEGs are much smaller than the equivalent uncompressed bitmaps. But before an image can be drawn onscreen, it must be expanded to its full, uncompressed size (usually this equates to $\text{image width} \times \text{height} \times \text{four bytes}$). To conserve memory, iOS often defers decompression of an image until it is drawn (more on this in Chapter 14, “Image IO”). Depending on how you load an image, the first time you assign it to a layer's contents (either directly or indirectly by using a `UIImageView`) or try to draw it into a Core Graphics context, it may need to be decompressed, which can take a considerable time for a large image.

After the layers have been successfully packaged up and sent to the render server, the CPU still has work to do: To display the layers on screen, Core Animation must loop through every visible layer in the render tree and convert it into a pair of textured triangles for consumption by OpenGL. The CPU has to do this conversion work because the GPU knows nothing about the structure of Core Animation layers. The amount of CPU work involved here scales proportionally with the number of layers, so too many layers in your hierarchy will indirectly cause additional CPU slowdown *every frame*, even though this work takes place outside of your application.

IO-Bound Operations

Something we haven't mentioned yet is *IO-bound* work. IO (Input/Output) in this context refers to accessing hardware such as flash storage or the network interface. Certain animations may require data to be loaded from flash (or even from a remote URL). A typical example would be a transition between two view controllers, which may lazily load a nib file and its contents, or a carousel of images, which may be too large to store in memory and so need to be loaded dynamically as the carousel scrolls.

IO is much slower than normal memory access, so if your animation is IO-bound that can be a big problem. Generally, this must be addressed using clever-but-awkward-to-get-right techniques such as threading, caching, and *speculative loading* (loading things in advance that you don't need right now but predict you will need in the future). These techniques are discussed in depth in Chapter 14.

Measure, Don't Guess

So, now that you know where the slowdown points might be in your animation, how do you go about fixing them? Well, first of all, *you don't*. There are many tricks to optimize an animation, but if applied blindly, these tricks often have as much chance of causing performance problems as correcting them.

It's important to always *measure* rather than guess why your animation is running slowly. There is a big difference between using your knowledge of performance to write code in an efficient way, and engaging in *premature optimization*. The former is good practice, but the latter is a waste of time and may actually be counterproductive.

So, how can you measure what's slowing down your app? Well, the first step is to make sure you that are testing under real-world conditions.

Test Reality, Not a Simulation

When you start doing any sort of performance tuning, test on a real iOS device, not on the simulator. The simulator is a brilliant tool that speeds up the development process, but it does *not* provide an accurate reflection of a real device's performance.

The simulator is running on your Mac, and the CPU in your Mac is most likely much faster than the one in your iOS device. Conversely, the GPU in your Mac is so different from the one in your iOS device that the simulator has to emulate the device's GPU entirely in software (on the CPU), which means that GPU-bound operations usually run *slower* on the simulator than they do on an iOS device, especially if you are writing bespoke OpenGL code using `CAEAGLLayer`.

This means that testing on the simulator gives a highly distorted view of performance. If an animation runs smoothly on the simulator, it may be terrible on a device. If it runs badly on the simulator, it may be fine on a device. You just can't be sure.

Another important thing when performance testing is to use the *Release* configuration, not Debug mode. When building for release, the compiler includes a number of optimizations that improve performance, such as stripping debugging symbols or removing and reorganizing code. You may also have implemented optimizations of your own, such as disabling `NSLog` statements when in release mode. You only care about release performance, so that's what you should test.

Finally, it is good practice to test on the slowest device that you support: If your base target is iOS 6, that probably means an iPhone 3GS or iPad 2. If at all possible, test on multiple devices and iOS versions, though, because Apple makes changes to the internals of iOS and iOS devices in major releases and these may actually *reduce* performance in some cases. For example, the iPad 3 is noticeably slower than the iPad 2 for a lot of animation and rendering tasks because of the challenge of rendering four times as many pixels (to support the Retina display).

Maintaining a Consistent Frame Rate

For smooth animation, you really want to be running at 60 FPS (frames per second), in sync with the refresh rate of the screen. With `NSTimer` or `CADisplayLink`-based animations you can get away with reducing your frame rate to 30 FPS, and you will still get a reasonable result, but there is no way to set the frame rate for Core Animation itself. If you don't hit 60 FPS consistently, there will be randomly skipped frames, which will look and *feel* bad for the user.

You can tell immediately if the frame rate is skipping badly just by using the app, but it's difficult to see the extent of the problem just by looking at the screen, and hard to tell whether your changes are making it slightly better or slightly worse. What you really want is an accurate numeric display of the frame rate.

You can put a frame rate counter in your app by using `CADisplayLink` to measure the frame period (as we did in Chapter 11, "Timer-Based Animation") and then display the

equivalent FPS value onscreen, but an in-app FPS display cannot ever be completely accurate when measuring Core Animation performance because it only measures the frame rate *inside* the app. We know that a lot of the animation happens *outside* the app (in the render server process), so while an in-app FPS counter can be a useful tool for certain types of performance issues, once you've identified a problem area, you will need to get some more accurate and detailed metrics to narrow down the cause. Apple provides the powerful *Instruments* toolset to help us with this.

Instruments

Instruments is an underutilized feature of the Xcode suite. Many iOS developers have either never used Instruments, or have only made use of the Leaks tool to check for retain cycles. But there are many other Instruments tools, including ones that are specifically designed to help us to tune our animation performance.

You can invoke Instruments by selecting the Profile option in the Product menu. (Before you do this, remember to target an actual iOS device, not the simulator.) This will present the screen shown in Figure 12.1. (If you don't see all of these options, you might be testing on the simulator by mistake.)



Figure 12.1 The Instruments tool selection window

As mentioned earlier, you should only ever profile the Release configuration of your app. Fortunately, the profiling option is set up to use the Release configuration by default, so you don't need to adjust your build scheme when you are profiling.

The tools we are primarily interested in are as follows:

- **Time Profiler**—Used to measure CPU usage, broken down by method/function.
- **Core Animation**—Used to debug all kinds of Core Animation performance issues.
- **OpenGL ES Driver**—Used to debug GPU performance issues. This tool is more useful if you are writing your own OpenGL code, but occasionally still handy for Core Animation work.

A nice feature of Instruments is that it lets us create our own groupings of tools. Regardless of which tool you select initially, if you open the Library window in Instruments, you can drag additional tools into the left sidebar. We'll create a group of the three tools that we are interested in so that we can use them all in parallel (see Figure 12.2).

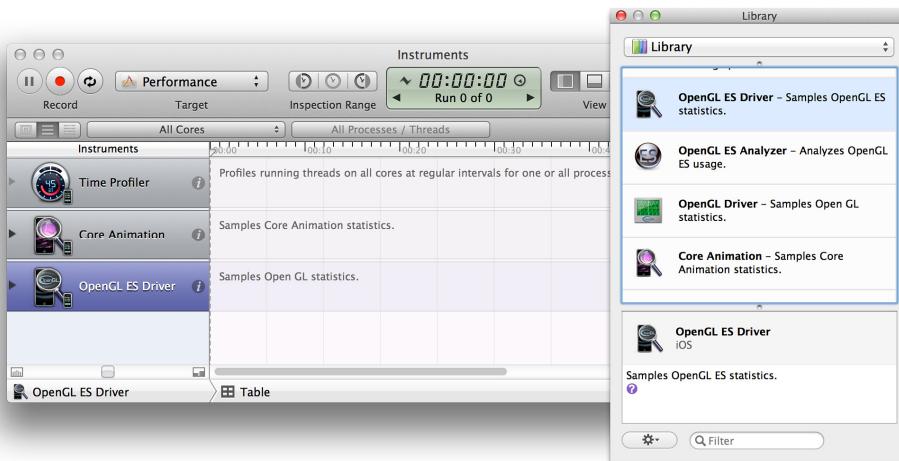


Figure 12.2 Adding additional tools to the Instruments sidebar

Time Profiler

The Time Profiler tool is used to monitor CPU usage. It gives us a breakdown of which methods in our application are consuming most CPU time. Using up a lot of CPU isn't necessarily a problem—you would expect your animation routines to be very CPU intensive because animation tends to be one of the most demanding tasks on an iOS device.

But if you are having performance problems, looking at the CPU time is a good way to determine if your performance is CPU-bound, and if so which methods need to be optimized (see Figure 12.3).

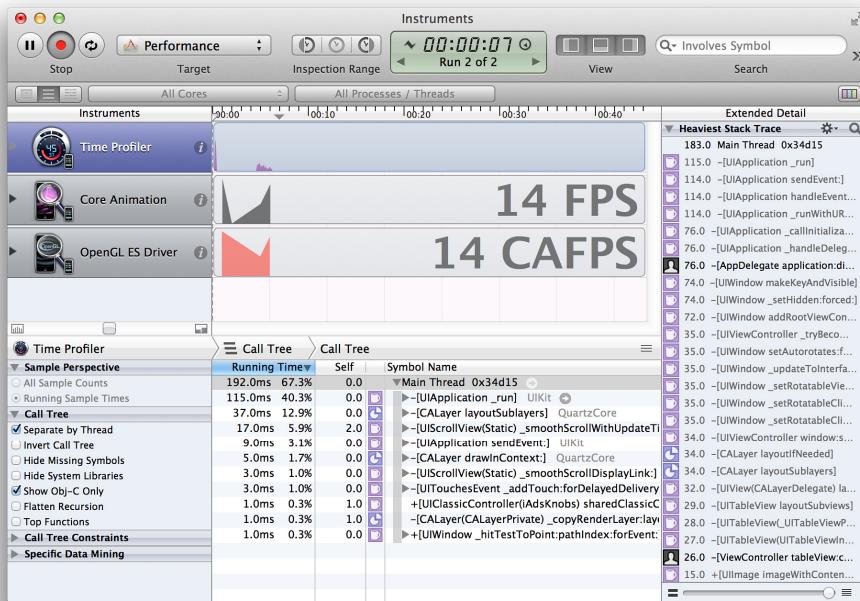


Figure 12.3 The Time Profiler tool

Time Profiler has some options to help narrow down the display to show only methods we care about. These can be toggled using checkboxes in the left sidebar. Of these, the most useful for our purposes are as follows:

- **Separate by Thread**—This groups methods by the thread in which they are executing. If our code is split between multiple threads, this will help to identify which ones are causing the problem.
- **Hide System Libraries**—This hides all methods and functions that are part of Apple’s frameworks. This helps us to identify which of our own methods contain bottlenecks. Since we can’t optimize framework methods, it is often useful to toggle this method to help narrow down the problem to something we can actually fix.
- **Show Obj-C Only**—This allows us to hide everything except Objective-C method calls. Most of the internal Core Animation code uses C or C++ functions, so this is a

good way to eliminate noise and help us focus on the methods that we are calling explicitly from our code.

Core Animation

The Core Animation tool is used to monitor Core Animation performance. It gives a breakdown of FPS sampled periodically, taking into account the parts of the animation that happen outside of our application (see Figure 12.4).

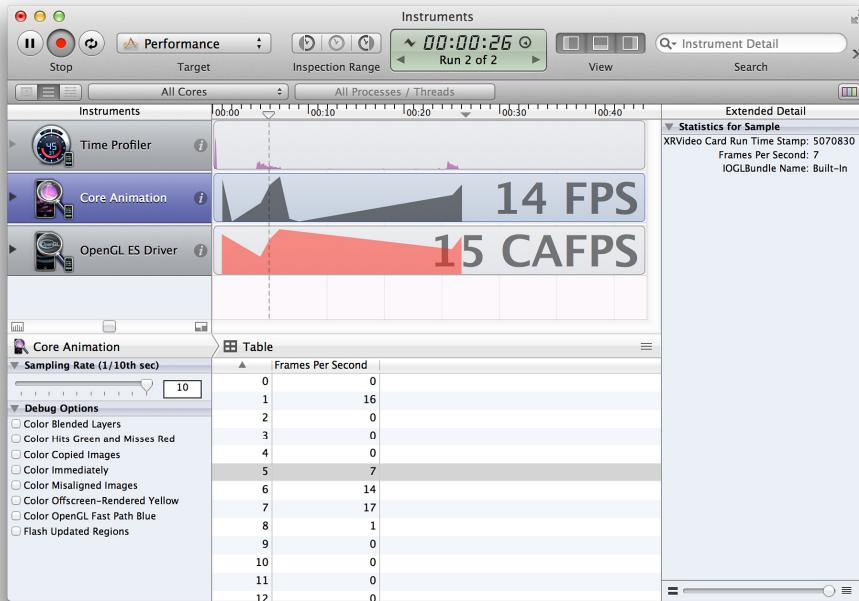


Figure 12.4 The Core Animation tool, with visual debug options

The Core Animation tool also provides a number of checkbox options to aid in debugging rendering bottlenecks:

- **Color Blended Layers**—This option highlights any areas of the screen where blending is happening (that is, where multiple semitransparent layers overlap one another), shaded from green to red based on severity. Blending can be bad for GPU performance because it leads to overdraw, and is a common cause of poor scrolling or animation frame rate.

- **Color Hits Green and Misses Red**—When using the `shouldRasterize` property, expensive layer drawing is cached and rendered as a single flattened image. This option highlights rasterized layers in red when the cache has to be regenerated. If the cache is regenerated frequently this is an indication that the rasterization may have a negative performance impact. (See Chapter 15, “Layer Performance,” for more detail about the implications of using the `shouldRasterize` property.)
- **Color Copied Images**—Sometimes the way that backing images are created means that Core Animation is forced to make a copy of the image and send it to the render server instead of just sending a pointer to the original. This option colors such images blue in the interface. Copying images is very expensive in terms of memory and CPU usage and should be avoided if possible.
- **Color Immediately**—Normally the Core Animation Instrument only updates the layer debug colors once every 10 milliseconds. For some effects, this may be too slow to detect an issue. This option sets it to update every frame (which may impact rendering performance and throw off the accuracy of frame rate measurements, so it should not be left on all the time).
- **Color Misaligned Images**—This highlights images that have been scaled or stretched for display, or which have not been correctly aligned to a pixel boundary (that is, they have nonintegral coordinates). Most of these will be false positives because it is common to deliberately scale images in an app, but if you are accidentally displaying a large image as a thumbnail, or making your graphics blurry by not aligning them correctly, this will help you to spot it.
- **Color Offscreen-Rendered Yellow**—This highlights any layer that requires offscreen rendering in yellow. Such layers may be candidates for using optimizations such as `shadowPath` or `shouldRasterize`.
- **Color OpenGL Fast Path Blue**—This highlights anywhere that you are drawing directly to the screen using OpenGL. If you are only using UIKit or Core Animation APIs, then this won’t have any effect. If you are using a `GLKView` or `CAEAGLLayer`, then if it doesn’t show in blue it may mean that you are doing more work than necessary by forcing the GPU to render to a texture instead of drawing directly to the screen.
- **Flash Updated Regions**—This will briefly highlight in yellow any content that is redrawn (that is, any layer that is doing software drawing using Core Graphics). Such drawing is slow. If it’s happening frequently, it may indicate a bug or an opportunity to improve performance by adding caching or using an alternative approach to generate your graphics.

Several of these layer coloring options are also available in the iOS Simulator Debug menu (see Figure 12.5). We stated earlier that it’s a bad idea to test performance in the simulator, but if you have identified that the cause of your performance issues can be highlighted using these debugging options, using the iOS Simulator to verify that you’ve resolved the problem may provide a quicker fix/test cycle than tethering to a device.

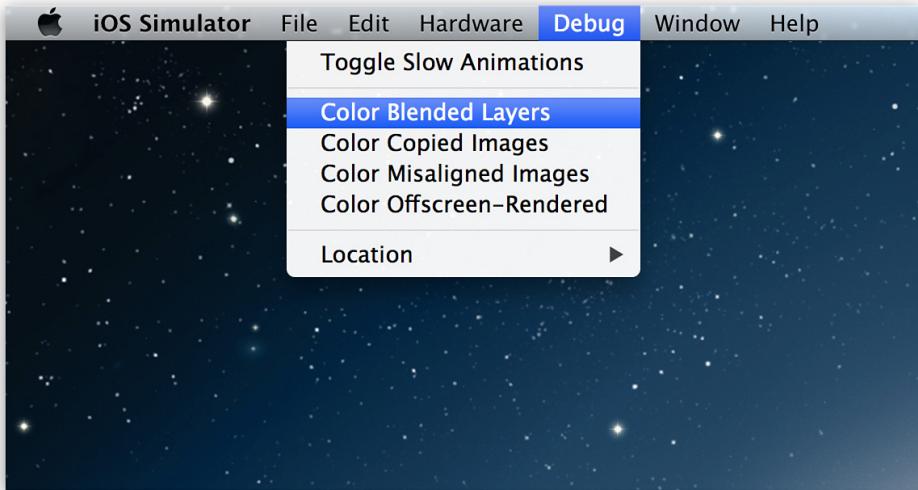


Figure 12.5 Core Animation visual debugging options in the iOS Simulator

OpenGL ES Driver

The OpenGL ES Driver tool can help you measure the GPU utilization, which is a good indicator as to whether your animation performance might be GPU-bound. It also provides the same FPS display as the Core Animation tool (see Figure 12.6).

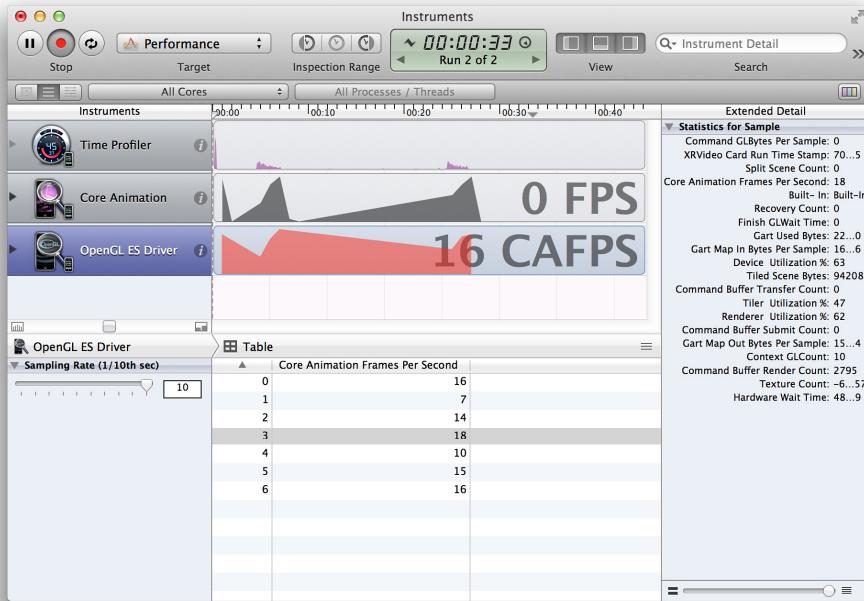


Figure 12.6 The OpenGL ES Driver tool

In the right sidebar are a number of useful metrics. Of these, the most relevant for Core Animation performance are as follows:

- **Renderer Utilization**—If this value is higher than ~50%, it suggests that your animation may be fill-rate limited, possibly due to offscreen rendering or overdraw caused by excessive blending.
- **Tiler Utilization**—If this value is higher than ~50%, it suggests that your animation may be geometry limited, meaning that there may be too many layers onscreen.

A Worked Example

Now that we are familiar with the animation performance tools in Instruments, let's use them to diagnose and solve a real-world performance problem.

We'll create a simple app that displays a mock contacts list in a table view using fake friend names and avatars. Note that even though the avatar images are stored in our app bundle, to make the app behave more realistically we're loading the avatar images individually in real

time rather than preloading them using the `-imageNamed:` method. We've also added some layer shadows to make our list a bit more visually appealing. Listing 12.1 shows the initial, naive implementation.

Listing 12.1 A Simple Contacts List Application with Mock Data

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

@interface ViewController () <UITableViewDataSource>

@property (nonatomic, strong) NSArray *items;
@property (nonatomic, weak) IBOutlet UITableView *tableView;

@end

@implementation ViewController

- (NSString *)randomName
{
    NSArray *first = @[@"Alice", @"Bob", @"Bill", @"Charles",
                      @"Dan", @"Dave", @"Ethan", @"Frank"];
    NSArray *last = @[@"Appleseed", @"Bandicoot", @"Caravan",
                      @"Dabble", @"Ernest", @"Fortune"];
    NSUInteger index1 = (rand()/(double)INT_MAX) * [first count];
    NSUInteger index2 = (rand()/(double)INT_MAX) * [last count];
    return [NSString stringWithFormat:@"%@ %@", first[index1], last[index2]];
}

- (NSString *)randomAvatar
{
    NSArray *images = @[@"Snowman", @"Igloo", @"Cone",
                        @"Spaceship", @"Anchor", @"Key"];
    NSUInteger index = (rand()/(double)INT_MAX) * [images count];
    return images[index];
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set up data
    NSMutableArray *array = [NSMutableArray array];
    for (int i = 0; i < 1000; i++)
    {
        //add name
```

```

        [array addObject:@{@"name": [self randomName],
                           @"image": [self randomAvatar]}];
    }
    self.items = array;

    //register cell class
    [self.tableView registerClass:[UITableViewCell class]
                     forCellReuseIdentifier:@"Cell"];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [self.items count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UITableViewCell *cell =
    [self.tableView dequeueReusableCellWithIdentifier:@"Cell"
                                         forIndexPath:indexPath];

    //load image
    NSDictionary *item = self.items[indexPath.row];
    NSString *filePath = [[NSBundle mainBundle] pathForResource:item[@"image"]
                                              ofType:@"png"];

    //set image and text
    cell.imageView.image = [UIImage imageWithContentsOfFile:filePath];
    cell.textLabel.text = item[@"name"];

    //set image shadow
    cell.imageView.layer.shadowOffset = CGSizeMake(0, 5);
    cell.imageView.layer.shadowOpacity = 0.75;
    cell.clipsToBounds = YES;

    //set text shadow
    cell.textLabel.backgroundColor = [UIColor clearColor];
    cell.textLabel.layer.shadowOffset = CGSizeMake(0, 2);
    cell.textLabel.layer.shadowOpacity = 0.5;
}

```

```

    return cell;
}

@end

```

When we scroll quickly, we get significant stuttering (see the FPS counter in Figure 12.7).



Figure 12.7 The frame rate falls to 15 FPS when scrolling in our app.

Instinctively, we might assume that the bottleneck is image loading. We're loading images from the flash drive in real time and not caching them, so that's probably why it's slow, right? We can fix that with some awesome code that will speculatively load our images asynchronously using GCD and then cache them using....

Stop. Wait.

Before we start writing code, let's test our hypothesis. We'll profile the app using our three Instruments tools to identify the problem. We suspect that the issue might be related to image loading, so let's start with the Time Profiler tool (see Figure 12.8).

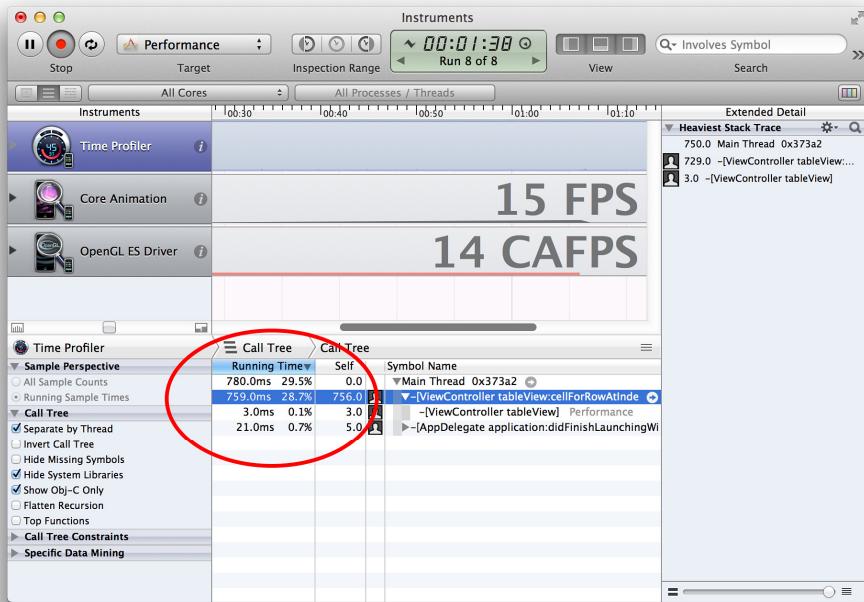


Figure 12.8 The timing profile for our contacts list app

The total percentage of CPU time spent in the `-tableView:cellForRowIndexPath:` method (which is where we load the avatar images) is only ~28%. That isn't really all that high. That would suggest that CPU/IO is not the limiting factor here.

Let's see if it's a GPU issue instead: We'll check the GPU utilization in the OpenGL ES Driver tool (see Figure 12.9).

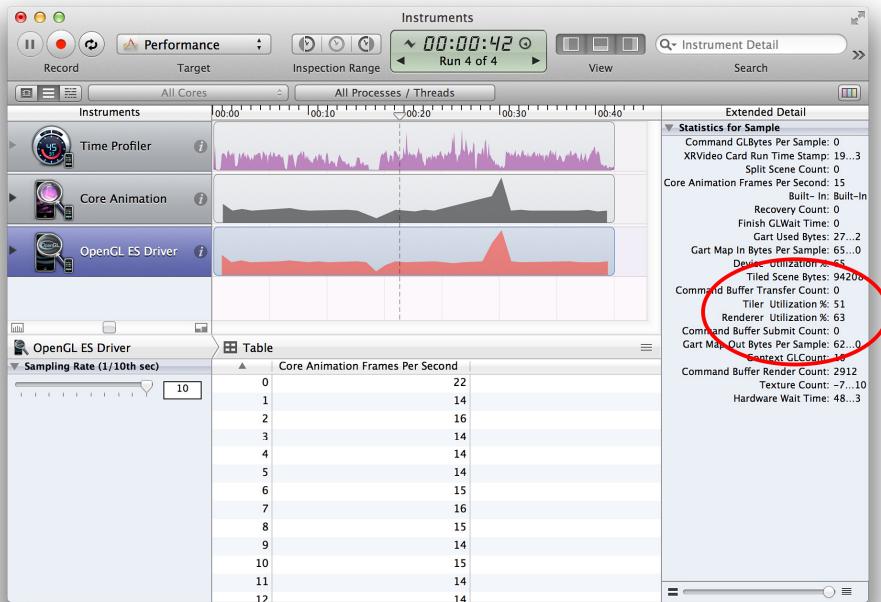


Figure 12.9 GPU utilization displayed in the OpenGL ES Driver tool

The tiler and renderer utilization values are at 51% and 63%, respectively. It looks like the GPU is having to work pretty hard to render our contacts list.

Why is our GPU usage so high? Let's inspect the screen using the Core Animation tool debugging options. Enable the Color Blended Layers option first (see Figure 12.10).

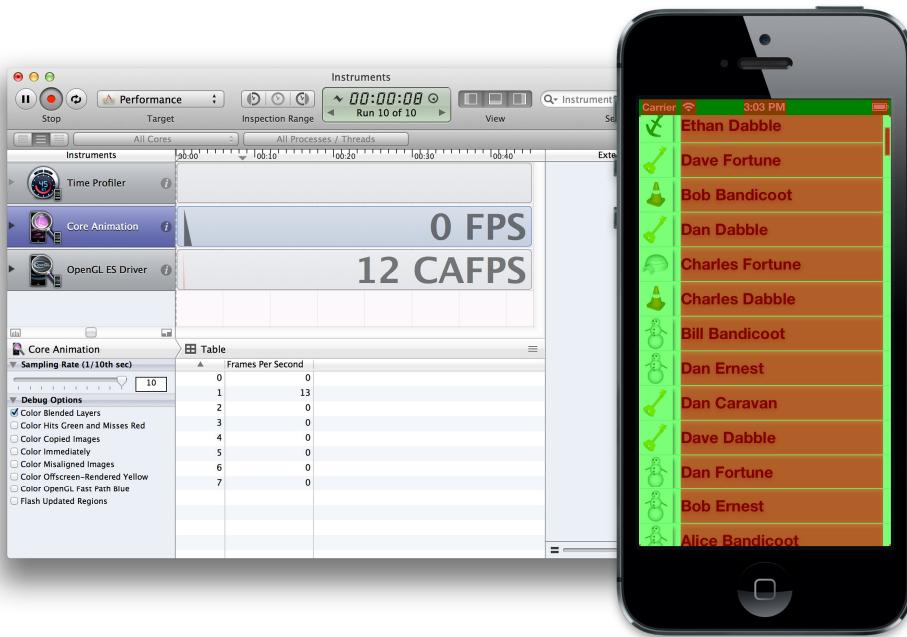


Figure 12.10 Debugging our app with the Color Blended Layers option

All that red on the screen indicates a high level of blending on the text labels, which is not surprising because we had to make the background transparent to apply our shadow effect. That explains why the renderer utilization is high.

What about offscreen drawing? Enable the Core Animation tool's Color Offscreen-Rendered Yellow option (see Figure 12.11).



Figure 12.11 The Color Offscreen–Rendered Yellow option

We see now that all our table cell content is being rendered offscreen. That must be because of the shadows we applied to the image and label views. Let's disable the shadows by commenting them out in our code and see whether that helps with our performance problem (see Figure 12.12).



Figure 12.12 Our app running at close-to 60 FPS with shadows disabled

Problem solved. Without the shadows, we get a smooth scroll. Our contacts list doesn't look as interesting as before, though. How can we keep our layer shadows and still have good performance?

Well, the text and avatar for each row don't need to change every frame, so it seems like the `UITableViewCell` layer is an excellent candidate for caching. We can cache our layer contents using the `shouldRasterize` property. This will render the layer once offscreen and then keep the result until it needs to be updated. Let's try that (see Listing 12.2).

Listing 12.2 Improving Performance with the `shouldRasterize` Option

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UITableViewCell *cell =
        [self.tableView dequeueReusableCellWithIdentifier:@"Cell"
            forIndexPath:indexPath];
    ...
}
```

```

//set text shadow
cell.textLabel.backgroundColor = [UIColor clearColor];
cell.textLabel.layer.shadowOffset = CGSizeMake(0, 2);
cell.textLabel.layer.shadowOpacity = 0.5;

//rasterize
cell.layer.shouldRasterize = YES;
cell.layer.rasterizationScale = [UIScreen mainScreen].scale;

return cell;
}

```

We are still drawing the layer content offscreen, but because we have explicitly enabled rasterization, Core Animation is now caching the result of that drawing, so it has less of a performance impact. We can verify that the caching is working correctly by using the Color Hits Green and Misses Red option in the Core Animation tool (see Figure 12.13).



Figure 12.13 Color Hits Green and Misses Red indicates caching works.

We see that—as expected—most rows are green, only flashing red briefly as they move onto the screen. Consequently, our frame rate is now much smoother.

So, our initial instinct was wrong. The loading of our images turns out *not* to have been a bottleneck after all, and any effort put into a complex, multithreaded loading and caching implementation would have been wasted. It’s a good thing we verified what the problem was before we tried to fix it!

Summary

In this chapter, you learned about how the Core Animation rendering pipeline works and where the bottlenecks are likely to occur. You also learned how to use Instruments to diagnose and fix performance problems.

In the next three chapters, we look at each of the common app performance pitfalls in detail and learn how to fix them.

This page intentionally left blank

Efficient Drawing

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

William Allan Wulf

In Chapter 12, “Tuning for Speed,” we looked at how to diagnose Core Animation performance problems using Instruments. There are many potential performance pitfalls when building iOS apps, but in this chapter, we focus on issues relating specifically to *drawing* performance.

Software Drawing

The term *drawing* is usually used in the context of Core Animation to mean *software* drawing (that is, drawing that is not GPU assisted). Software drawing in iOS is done primarily using the Core Graphics framework, and while sometimes necessary, it’s *really* slow compared to the hardware accelerated rendering and compositing performed by Core Animation and OpenGL.

In addition to being slow, software drawing requires a lot of memory. A `CALayer` requires relatively little memory by itself; it is only the backing image that takes up any significant space in RAM. Even if you assign an image to the `contents` property directly, it won’t use any additional memory beyond that required for storing a single (uncompressed) copy of the image; and if the same image is used as the `contents` for multiple layers, that memory will be shared between them, not duplicated.

But as soon as you implement the `CALayerDelegate -drawLayer:inContext:` method or the `UIView -drawRect:` method (the latter of which is just a wrapper around the former), an offscreen drawing context is created for the layer, and that context requires an amount of

memory equal to the width \times height of the layer (in pixels, not points) \times 4 bytes. For a full-screen layer on a Retina iPad, that's $2048 \times 1536 \times 4$ bytes, which amounts to a whole 12MB that must not only be stored in RAM, but must be wiped and repopulated every time the layer is redrawn.

Because software drawing is so expensive, you should avoid redrawing your view unless absolutely necessary. The secret to improving drawing performance is generally to try to *do as little drawing as possible*.

Vector Graphics

A common reason to use Core Graphics drawing is for vector graphics that cannot easily be created using images or layer effects. Vector drawing might involve the following:

- Arbitrary polygonal shapes (anything other than a rectangle)
- Diagonal or curved lines
- Text
- Gradients

Let's try an example. Listing 13.1 shows the code for a basic line-drawing app. The app converts user touches into points on a `UIBezierPath`, which is then drawn into the view. We've included all the drawing logic in a `UIView` subclass called `DrawingView` in this case (the view controller is empty), but you could implement the touch handling in the view controller instead if you prefer. Figure 13.1 shows a drawing created by using this app.

Listing 13.1 Implementing a Simple Drawing App Using Core Graphics

```
#import "DrawingView.h"

@interface DrawingView : UIView

@property (nonatomic, strong) UIBezierPath *path;

@end

@implementation DrawingView

- (void)awakeFromNib
{
    //create a mutable path
    self.path = [[UIBezierPath alloc] init];
    self.path.lineJoinStyle = kCGLineJoinRound;
    self.path.lineCapStyle = kCGLineCapRound;
}
```

```
    self.path.lineWidth = 5;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the starting point
    CGPoint point = [[touches anyObject] locationInView:self];

    //move the path drawing cursor to the starting point
    [self.path moveToPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the current point
    CGPoint point = [[touches anyObject] locationInView:self];

    //add a new line segment to our path
    [self.path addLineToPoint:point];

    //redraw the view
    [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
{
    //draw path
    [[UIColor clearColor] setFill];
    [[UIColor redColor] setStroke];
    [self.path stroke];
}

@end
```



Figure 13.1 A simple sketch produced using Core Graphics

The problem with this implementation is that the more we draw, the slower it gets. Because we are redrawing the entire `UIBezierPath` every time we move our finger, the amount of drawing work that needs to be done increases every frame as the path becomes more complex, causing the frame rate to plummet. We need a better approach.

Core Animation provides specialist classes for drawing these types of shape with hardware assistance (as covered in detail in Chapter 6, “Specialized Layers”). Polygons, lines, and curves can be drawn using `CAShapeLayer`. Text can be drawn using `CATextLayer`. Gradients can be drawn using `CAGradientLayer`. These will all be substantially faster than using Core Graphics, and they avoid the overhead of creating a backing image.

If we modify our drawing app to use `CAShapeLayer` instead of Core Graphics, the performance is substantially improved (see Listing 13.2). Performance will inevitably still degrade as the complexity of the path increases, but it would now take a very complex drawing to make an appreciable difference to the frame rate.

Listing 13.2 Reimplementing the Drawing App Using `CAShapeLayer`

```
#import "DrawingView.h"
#import <QuartzCore/QuartzCore.h>

@interface DrawingView : NSObject

@property (nonatomic, strong) UIBezierPath *path;

@end
```

```
@implementation DrawingView

+ (Class)layerClass
{
    //this makes our view create a CAShapeLayer
    //instead of a CALayer for its backing layer
    return [CAShapeLayer class];
}

- (void)awakeFromNib
{
    //create a mutable path
    self.path = [[UIBezierPath alloc] init];

    //configure the layer
    CAShapeLayer *shapeLayer = (CAShapeLayer *)self.layer;
    shapeLayer.strokeColor = [UIColor redColor].CGColor;
    shapeLayer.fillColor = [UIColor clearColor].CGColor;
    shapeLayer.lineJoin = kCALineJoinRound;
    shapeLayer.lineCap = kCALineCapRound;
    shapeLayer.lineWidth = 5;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the starting point
    CGPoint point = [[touches anyObject] locationInView:self];

    //move the path drawing cursor to the starting point
    [self.path moveToPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the current point
    CGPoint point = [[touches anyObject] locationInView:self];

    //add a new line segment to our path
    [self.path addLineToPoint:point];

    //update the layer with a copy of the path
    ((CAShapeLayer *)self.layer).path = self.path.CGPath;
}
```

```
}
```

```
@end
```

Dirty Rectangles

Sometimes it's not viable to replace Core Graphics drawing with a `CAShapeLayer` or one of the other vector graphics layers. Consider our drawing app: The current implementation uses hard, straight lines, ideally suited to vector drawing. But suppose we enhance the app so that it resembles a chalkboard, with a chalk-like texture applied to our lines. A simple way to simulate chalk is to use a small "brush stroke" image and paste it onto the screen wherever the user's finger touches, but that approach isn't achievable using a `CAShapeLayer`.

We could create an individual layer for each brush stroke, but this would perform very badly. The practical upper limit for the number of layers onscreen simultaneously is at most a few hundred, and we would quickly exceed that. This is the sort of situation where we have little recourse but to use Core Graphics drawing (unless we want to do something *far* more complex using OpenGL).

Our initial chalkboard implementation is shown in Listing 13.3. We've modified the `DrawingView` from our earlier example to use an array of stroke positions instead of a `UIBezierPath`. Figure 13.2 shows the result.

Listing 13.3 A Simple Chalkboard-Style Drawing App

```
#import "DrawingView.h"
#import <QuartzCore/QuartzCore.h>

#define BRUSH_SIZE 32

@interface DrawingView : NSObject

@property (nonatomic, strong) NSMutableArray *strokes;

@end

@implementation DrawingView

- (void)awakeFromNib
{
    //create array
    self.strokes = [NSMutableArray array];
}
```

```
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the starting point
    CGPoint point = [[touches anyObject] locationInView:self];

    //add brush stroke
    [self addBrushStrokeAtPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    //get the touch point
    CGPoint point = [[touches anyObject] locationInView:self];

    //add brush stroke
    [self addBrushStrokeAtPoint:point];
}

- (void)addBrushStrokeAtPoint:(CGPoint)point
{
    //add brush stroke to array
    [self.strokes addObject:[NSValue valueWithCGPoint:point]];

    //needs redraw
    [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
{
    //redraw strokes
    for (NSValue *value in self.strokes)
    {
        //get point
        CGPoint point = [value CGPointValue];

        //get brush rect
        CGRect brushRect = CGRectMake(point.x - BRUSH_SIZE/2,
                                      point.y - BRUSH_SIZE/2,
                                      BRUSH_SIZE, BRUSH_SIZE);

        //draw brush stroke
        [[UIImage imageNamed:@"Chalk.png"] drawInRect:brushRect];
    }
}
```

```
    }  
}  
  
@end
```

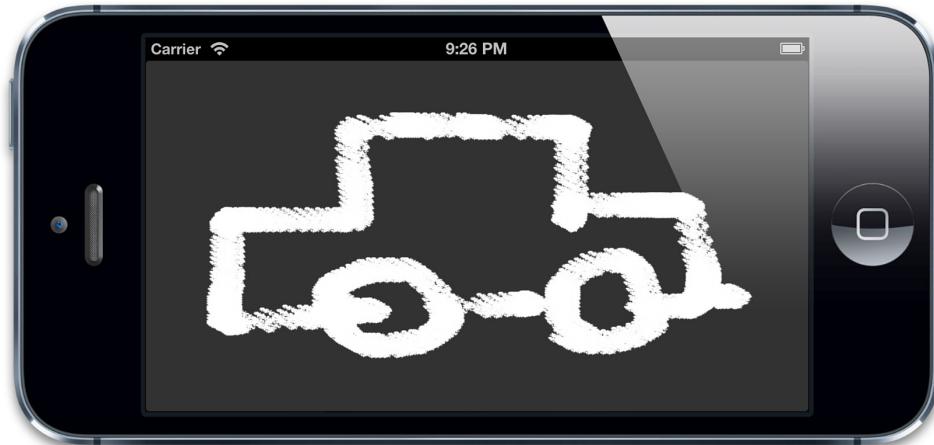


Figure 13.2 A sketch produced using the chalkboard drawing app

This performs reasonably well on the simulator, but is not great on a real device. The problem is that we are redrawing every previous brush stroke each time our finger moves, even though the vast majority of the scene hasn't changed. The more we draw, the slower it gets, so the gap between each new stroke increases over time as the frame rate drops (see Figure 13.3). How can we improve this performance?

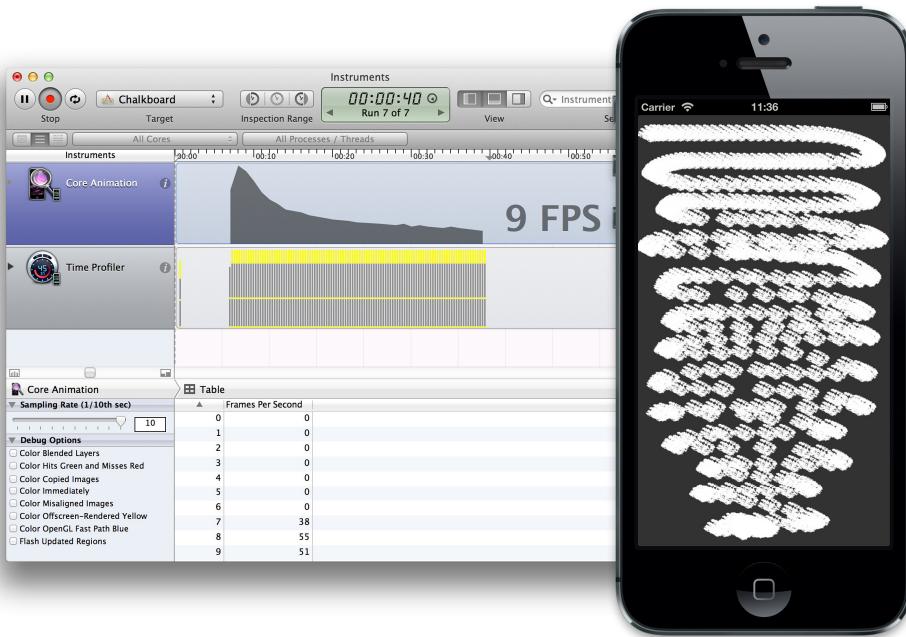


Figure 13.3 The frame rate and line quality degrades as we draw.

To cut down on unnecessary drawing, both Mac OS and iOS divide the screen into regions that need redrawing and ones that don’t. The part that needs to be redrawn is known as the “dirty” region. Because it’s not practical to redraw nonrectangular regions due to the complexity of clipping and blending the edges, the entire containing rectangle around the part of the dirty region that overlaps a given view will be redrawn—this is the *dirty rectangle*.

When a view is modified, it may need to be redrawn. But often, only part of the view has changed, so redrawing the entire backing image would be wasteful. Because Core Animation doesn’t usually know anything about your custom drawing code, it cannot calculate the dirty rectangle by itself. You can provide this information, however.

When you detect that a specific part of your view/layer needs to be redrawn, you mark it as dirty by calling `-setNeedsDisplayInRect:` passing the affected rectangle as a parameter. This will cause the view’s `-drawRect:` method (or the layer delegate’s `-drawLayer:inContext:` method) to be called automatically prior to the next display update.

The `CGContext` that is passed to `-drawLayer:inContext:` will automatically be clipped to match the dirty rectangle. To find out the dimensions of that rectangle, you can use the

`CGContextGetClipBoundingBox()` function to get this from the context itself. This is simpler when using `-drawRect:` because the `CGRect` is passed directly as a parameter.

You should try to limit your drawing to only the parts of the image that overlap this rectangle. Anything you draw outside of the dirty `CGRect` will be clipped automatically, but the CPU time spent on evaluating and discarding those redundant drawing commands will be wasted.

By clipping your own drawing rather than relying on Core Graphics to do it for you, you may be able to avoid unnecessary processing. That said, if your clipping logic is complex, you might be better off letting Core Graphics handle it; only clip your own drawing if you can do so efficiently.

Listing 13.4 shows an updated version of our `-addBrushStrokeToPoint:` method that only redraws the rectangle around the current brush stroke. In addition to only refreshing the part of the view around the last brush stroke, we also use `CGRectIntersectsRect()` to avoid redrawing any old brush strokes that don't overlap the updated region. This significantly improves the drawing performance (see Figure 13.4).

Listing 13.4 Using `setNeedsDisplayInRect:` to Reduce Unnecessary Drawing

```
- (void)addBrushStrokeToPoint:(CGPoint)point
{
    //add brush stroke to array
    [self.strokes addObject:[NSValue valueWithCGPoint:point]];

    //set dirty rect
    [self setNeedsDisplayInRect:[self brushRectForPoint:point]];
}

- (CGRect)brushRectForPoint:(CGPoint)point
{
    return CGRectMake(point.x - BRUSH_SIZE/2,
                      point.y - BRUSH_SIZE/2,
                      BRUSH_SIZE, BRUSH_SIZE);
}

- (void)drawRect:(CGRect)rect
{
    //redraw strokes
    for (NSValue *value in self.strokes)
    {
        //get point
        CGPoint point = [value CGPointValue];

        //get brush rect
        CGRect brushRect = [self brushRectForPoint:point];
    }
}
```

```

//only draw brush stroke if it intersects dirty rect
if (CGRectIntersectsRect(rect, brushRect))
{
    //draw brush stroke
    [UIImage imageNamed:@"Chalk.png"] drawInRect:brushRect];
}
}
}
}

```

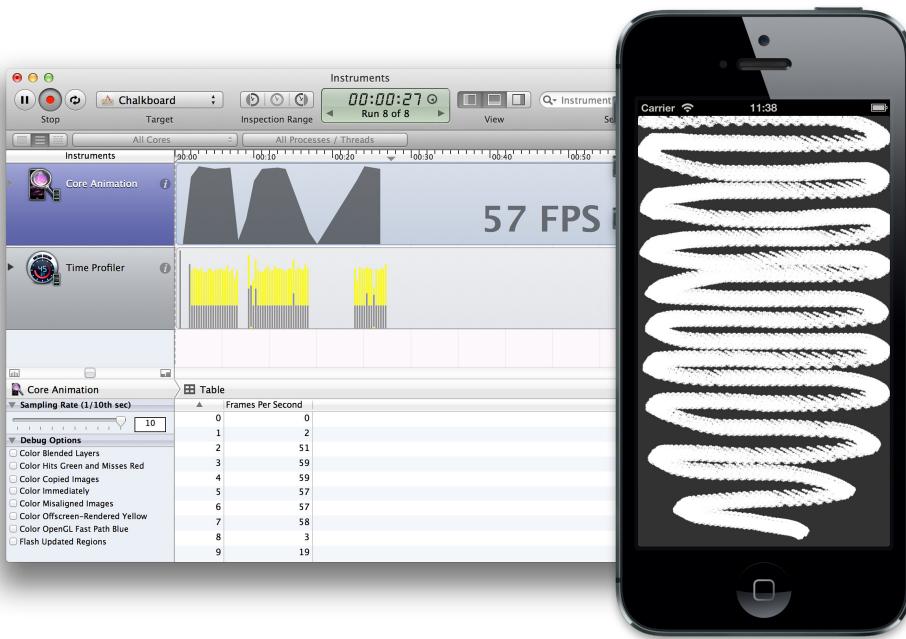


Figure 13.4 Better frame rate and smoother line thanks to smarter drawing

Asynchronous Drawing

The single-threaded nature of UIKit means that backing images usually have to be updated on the main thread, which means that the drawing interrupts user interaction and can make

the entire app feel unresponsive. We can't help that drawing is slow, but it would be good if we could avoid making the user wait for it to complete.

There are a few ways around this: In some cases, you can speculatively draw view contents in advance on a separate thread and then set the resultant image directly as the layer contents. This can be awkward to set up, however, and is only applicable in certain cases. Core Animation provides a couple of alternatives: `CATiledLayer` and the `drawsAsynchronously` property.

CATiledLayer

We explored `CATiledLayer` briefly in Chapter 6. In addition to subdividing a layer into individual tiles that are updated independently (a kind of automation of the dirty rectangles concept), `CATiledLayer` also has the interesting feature of calling the `-drawLayer:inContext:` method for each tile *concurrently on multiple threads*. This avoids blocking the user interface and also enables it to take advantage of multiple processor cores for faster tile drawing. A `CATiledLayer` with just a single tile is a cheap way to implement an asynchronously updating image view.

drawsAsynchronously

In iOS 6, Apple added this intriguing new property to `CALayer`. The `drawsAsynchronously` property modifies the `CGContext` that is passed to the `-drawLayer:inContext:` method, allowing it to defer execution of drawing commands so that they don't block user interaction.

This isn't the same kind of asynchronous drawing as `CATiledLayer` uses. The `-drawLayer:inContext:` method itself is only ever called on the main thread, but the `CGContext` doesn't wait for each drawing command to complete before proceeding. Instead, it will queue up the commands and perform the actual drawing on a background thread after the method has returned.

According to Apple, this feature works best for views that are redrawn frequently (such as our drawing app, or something like a `UITableViewCell`) and will not provide any benefit for layer contents that are only drawn once, or infrequently.

Summary

In this chapter, we discussed the performance challenges around software drawing using Core Graphics and explored some ways that we can either improve drawing performance or cut down the amount of drawing we need to do.

In Chapter 14, "Image IO," we look at image loading performance.

Image IO

The idea of latency is worth thinking about.

Kevin Patterson

In Chapter 13, “Efficient Drawing,” we looked at performance issues relating to Core Graphics drawing and how to fix them. Closely related to *drawing* performance is *image* performance. In this chapter, we investigate how to optimize the loading and display of images from the flash drive or over a network connection.

Loading and Latency

The time taken to actually *draw* an image is not usually the limiting factor when it comes to performance. Images consume a lot of memory, and it may be impractical to keep all the images that your app needs to display loaded in memory at once, meaning that you will need to periodically load and unload images while the app is running.

The speed at which an image file can be loaded is limited not only by the CPU but also by IO (Input/Output) latency. The flash storage in an iOS device is faster than a traditional hard disk, but still around 200 times slower than RAM, making it essential that you manage loading carefully to avoid noticeable delays.

Whenever possible, try to load images at inconspicuous times in your application’s lifecycle, such as at launch, or between screen transitions. The maximum comfortable delay between pressing a button and seeing a reaction onscreen is around 200ms—much more than the ~16ms between animation frames. You can get away with taking even more time to load your images when the app first starts up, but the iOS watchdog timer will terminate your app if it doesn’t start within 20 seconds (and users won’t be impressed if it takes more than 2 or 3).

In some cases, it simply isn't practical to load everything in advance. Take something like an image carousel that may potentially contain thousands of images: The user will expect to be able to flick through the images quickly without any slowdown, but it would be impossible to preload all of the images; it would take too long and consume too much memory.

Images may also sometimes need to be retrieved from a remote network connection, which can take considerably more time than loading from the flash drive and may even fail altogether due to connection problems (after several seconds of trying). You cannot do network loading on the main thread and expect the user to wait for it with a frozen screen. You need to use a *background thread*.

Threaded Loading

In our contacts list example in Chapter 12, “Tuning for Speed,” the images were small enough to load in real time on the main thread as we scrolled. But for large images, this doesn’t work well because the loading takes too long and causes scrolling to stutter. Scrolling animations are updated on the main run loop, and are therefore more vulnerable to CPU-related performance issues than CAAnimation, which is run in the render server process.

Listing 14.1 shows the code for a basic image carousel, implemented using `UICollectionView`. The image loading is performed synchronously on the main thread in the `-collectionView:cellForItemAtIndexPath:` method (see Figure 14.1 for an example).

Listing 14.1 Implementing an Image Carousel Using UICollectionView

```
//register cell class
[self.collectionView registerClass:[UICollectionViewCell class]
    forCellWithReuseIdentifier:@"Cell"];
}

- (NSInteger)collectionView:(UICollectionView *)collectionView
    numberOfItemsInSection:(NSInteger)section
{
    return [self.imagePaths count];
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UICollectionViewCell *cell =
    [collectionView dequeueReusableCellWithReuseIdentifier:@"Cell"
        forIndexPath:indexPath];

    //add image view
    const NSInteger imageTag = 99;
    UIImageView *imageView = (UIImageView *)[cell viewWithTag:imageTag];
    if (!imageView)
    {
        imageView = [[UIImageView alloc] initWithFrame:
            cell.contentView.bounds];
        imageView.tag = imageTag;
        [cell.contentView addSubview:imageView];
    }

    //set image
    NSString *imagePath = self.imagePaths[indexPath.row];
    imageView.image = [UIImage imageWithContentsOfFile:imagePath];
    return cell;
}

@end
```

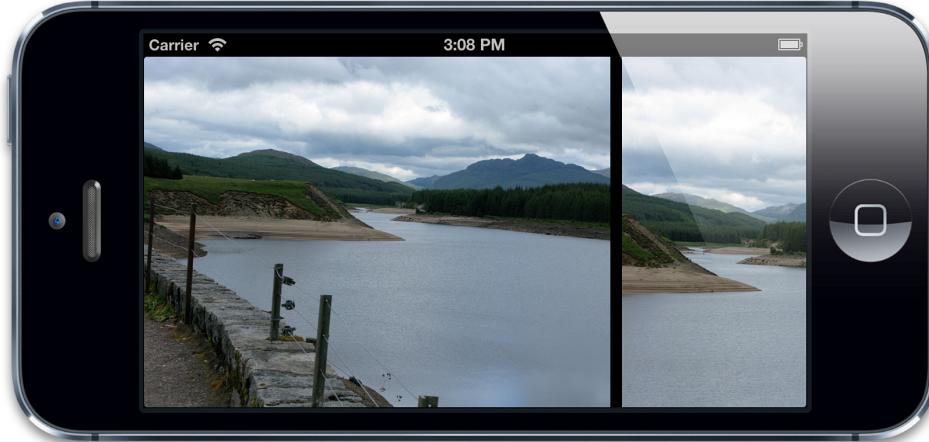


Figure 14.1 The image carousel in action

The images in the carousel are 800×600-pixel PNGs of around 700KB each—slightly too large for an iPhone 5 to load within one-sixtieth of a second. These images are loaded on-the-fly as the carousel scrolls, and (as expected) the scrolling stutters. The Time Profiler instrument (see Figure 14.2) reveals that a lot of time is being spent in the `UIImage +imageWithContentsOfFile:` method. Clearly, image loading is our bottleneck.

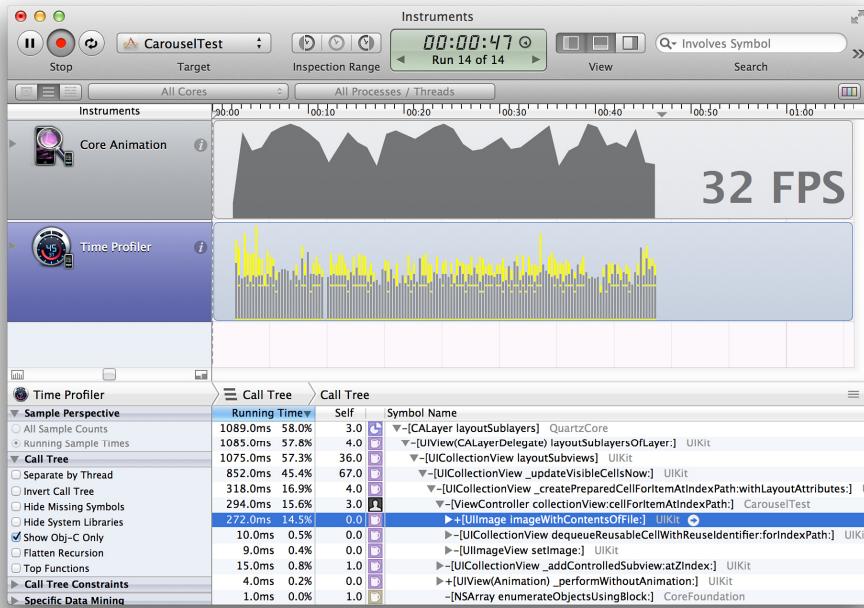


Figure 14.2 The Time Profiler instrument showing the CPU bottleneck

The only way to improve the performance here is to move the image loading onto another thread. This won't help to reduce the actual loading time (it might even make it slightly worse, because the system will potentially be devoting a smaller slice of CPU time to processing the loaded image data), but it means that the main thread can continue doing other things, like responding to user input and animating the scroll.

To load the images on a background thread, we can either create our own threaded loading solution using GCD or `NSOperationQueue`, or we can use `CATiledLayer`. To load images from a remote network, we could use an asynchronous `NSURLConnection`, but that's not a very efficient option for locally stored files.

GCD and `NSOperationQueue`

GCD (Grand Central Dispatch) and `NSOperationQueue` are similar in that they both allow us to queue blocks to be executed sequentially on a thread. `NSOperationQueue` has an Objective-C interface (as opposed to the global C functions used by GCD) and provides fine-grained control over operation prioritization and dependencies, but requires a bit more setup code.

Listing 14.2 shows an updated `-collectionView:cellForItemAtIndexPath:` method that uses GCD to load the images on a low priority background queue instead of on the main thread. We switch back to the main thread before we actually apply the newly loaded image to the cell because it's unsafe to access views on a background thread.

Because cells are recycled in a `UICollectionView`, we can't be sure that the cell hasn't been reused with a different index in the meantime while we were loading the image. To avoid images being loaded into the wrong cells, we tag the cell with the index before loading and check that the tag hasn't changed before we set the image.

Listing 14.2 Using GCD to Load the Carousel Images

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                               cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UICollectionViewCell *cell =
        [collectionView dequeueReusableCellWithReuseIdentifier:@"Cell"
                                              forIndexPath:indexPath];

    //add image view
    const NSInteger imageTag = 99;
    UIImageView *imageView = (UIImageView *)[cell viewWithTag:imageTag];
    if (!imageView)
    {
        imageView = [[UIImageView alloc] initWithFrame:
                     cell.contentView.bounds];
        imageView.tag = imageTag;
        [cell.contentView addSubview:imageView];
    }

    //tag cell with index and clear current image
    cell.tag = indexPath.row;
    imageView.image = nil;

    //switch to background thread
    dispatch_async(
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0), ^{
            //load image
            NSInteger index = indexPath.row;
            NSString *imagePath = self.imagePaths[index];
            UIImage *image = [UIImage imageWithContentsOfFile:imagePath];

            //set image on main thread, but only if index still matches up
            dispatch_async(dispatch_get_main_queue(), ^{

```

```

        if (index == cell.tag)
    {
        imageView.image = image;
    }
});
});

return cell;
}

```

When we run this updated version, the performance is better than the original nonthreaded version, but still not perfect (see Figure 14.3).

We can see that the `+imageWithContentsOfFile:` method no longer appears at the top of the CPU time trace, so we *have* fixed the loading delay. The problem is that we were assuming that the only performance bottleneck for our carousel was the actual loading of the image file, but that's not the case. Loading the image file data into memory is only the first part of the problem.

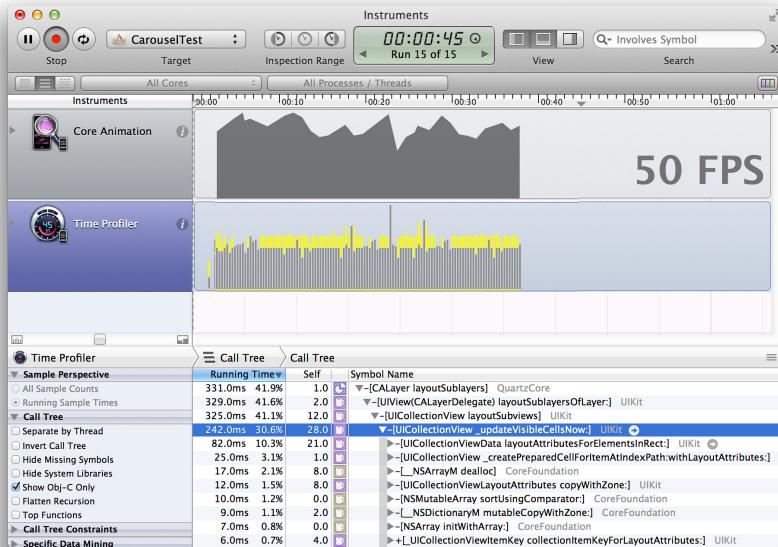


Figure 14.3 Improved performance when loading on a background thread

Deferred Decompression

Once an image file has been loaded, it must then be decompressed. This decompression can be a computationally complex task and take considerable time. The decompressed image will also use substantially more memory than the original.

The relative CPU time spent loading versus decompressing will depend on the image format. For PNG images, loading takes longer than for JPEGs because the file is proportionally larger, but decompression is relatively fast, especially since Xcode recompresses any PNGs included in the project using optimal settings for fast decoding. JPEG images are smaller and load quicker, but the decompression step is more expensive because the JPEG decompression algorithm is more complex than the zip-based algorithm used in PNG.

When you load an image, iOS usually defers decompression until later to conserve memory. This can cause a performance hiccup when you actually try to draw the image, as it has to be decompressed *at the point of drawing* (which is often the worst possible time).

The simplest way to avoid deferred decompression is to load images using the `UIImage +imageNamed:` method. Unlike `+imageWithContentsOfFile:` (and all the other `UIImage` loading methods), this method decompresses the image immediately after loading (as well as having other benefits that we discuss later in the chapter). The problem is that `+imageNamed:` works only for images loaded from within the application resources bundle, so it's not an option for user-generated content, or downloaded images.

Another way to decompress an image immediately is to assign it as the contents of a layer or as the `image` property of a `UIImageView`. Unfortunately, this has to be done on the main thread and so usually won't help with performance problems.

A third approach is to bypass UIKit altogether and load the image using the ImageIO framework instead, as follows:

```
NSInteger index = indexPath.row;
NSURL * imageURL = [NSURL fileURLWithPath:self.imagePaths[index]];
NSDictionary * options = @{@"__bridge id")kCGImageSourceShouldCache: @YES};
CGImageSourceRef source = CGImageSourceCreateWithURL(
    (__bridge CFURLRef) imageURL, NULL);
CGImageRef imageRef = CGImageSourceCreateImageAtIndex(source, 0,
    (__bridge CFDictionaryRef) options);
UIImage * image = [UIImage imageWithCGImage:imageRef];
CGImageRelease(imageRef);
CFRelease(source);
```

This allows you to make use of the `kCGImageSourceShouldCache` option when creating the image, which forces it to decompress immediately and retain the decompressed version for the lifetime of the image.

The final option is to load the image using UIKit as normal, but immediately draw it into a `CGContext`. An image must be decompressed before it is drawn, so this forces the decompression to happen immediately. The advantage of doing this is that the drawing can be done on a background thread (like the loading itself) so it need not block the UI.

There are two approaches you can take when pre-drawing an image for the purpose of forcing decompression:

- Draw a single pixel of the image into a single-pixel sized `CGContext`. This still decompresses the entire image, but the drawing itself takes essentially no time. The disadvantage is that the loaded image may not be optimized for drawing on the specific device and may take longer to draw in future as a result. It is also possible that iOS may discard the decompressed image again to conserve memory.
- Draw the entire image into a `CGContext`, discard the original, and replace it with a new image created from the context contents. This is more computationally expensive than drawing a single pixel, but the resultant image will be optimized for drawing on that specific iOS device, and since the original compressed image has been discarded, iOS can't suddenly decide to throw away the decompressed version again to save memory.

It's worth noting that Apple specifically recommends *against* using these kinds of tricks to bypass the standard image decompression logic (which is not surprising—they chose the default behavior for a reason), but if you are building apps that use a lot of large images, then you sometimes have to game the system if you want great performance.

Assuming that using `+imageNamed:` isn't an option, drawing the entire image into a `CGContext` seems to work best. Although you might think that the extra drawing step would make this perform unfavorably compared with the other decompression techniques, the newly created image (which is optimized specifically for the particular device on which it was created) seems to draw faster on every subsequent use than if you keep the original.

Also, if you intend to display the image at a smaller-than-actual size, redrawing it at the correct size for display on a background thread once and for all will perform better than re-applying the scaling every time it's displayed (although in this example, our loaded images are the correct size anyway, so that particular benefit doesn't apply here).

If you modify the `-collectionView:cellForItemAtIndexPath:` method to redraw the image prior to display (see Listing 14.3), you should find that the scrolling is now perfectly smooth.

Listing 14.3 Forcing Image Decompression Prior to Display

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                           cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UICollectionViewCell *cell =

```

```

[collectionView dequeueReusableCellWithReuseIdentifier:@"Cell"
                                         forIndexPath:indexPath] ;

...

//switch to background thread
dispatch_async(
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0), ^{
    //load image
    NSInteger index = indexPath.row;
    NSString *imagePath = self.imagePaths[index];
    UIImage *image = [UIImage imageWithContentsOfFile:imagePath];

    //redraw image using device context
    UIGraphicsBeginImageContextWithOptions(imageView.bounds.size, YES, 0);
    [image drawInRect:imageView.bounds];
    image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    //set image on main thread, but only if index still matches up
    dispatch_async(dispatch_get_main_queue(), ^{
        if (index == imageView.tag)
        {
            imageView.image = image;
        }
    });
});

return cell;
}

```

CATiledLayer

As demonstrated in Chapter 6, “Specialized Layers,” `CATiledLayer` can be used to load and display very large images asynchronously without blocking user interaction. But we can also use `CATiledLayer` to load our carousel images by creating a separate `CATiledLayer` instance for every cell in the `UICollectionView`, with just a single tile for each.

There are potentially a couple of disadvantages to using `CATiledLayer` in this way:

- The `CATiledLayer` algorithm for queuing and caching is not exposed, so we have to hope that it’s tuned appropriately for our purposes.

- CATiledLayer always requires us to redraw our image into a CGContext, even if it is already the same size as our tile and has already been decompressed (and could therefore be used directly as the layer contents, without redrawing).

Let's find out if those potential disadvantages make a difference in practice: Listing 14.4 shows a reimplementation of our image carousel using `CATiledLayer`.

Listing 14.4 The Image Carousel Updated to Use `CATiledLayer` for Loading

```

forIndexPath:indexPath] ;

//add the tiled layer
CATiledLayer *tileLayer = [cell.contentView.layer.sublayers lastObject];
if (!tileLayer)
{
    tileLayer = [CATiledLayer layer];
    tileLayer.frame = cell.bounds;
    tileLayer.contentsScale = [UIScreen mainScreen].scale;
    tileLayer.tileSize = CGSizeMake(
        cell.bounds.size.width * [UIScreen mainScreen].scale,
        cell.bounds.size.height * [UIScreen mainScreen].scale);
    tileLayer.delegate = self;
    [tileLayer setValue:@(indexPath.row) forKey:@"index"];
    [cell.contentView.layer addSublayer:tileLayer];
}

//tag the layer with the correct index and reload
tileLayer.contents = nil;
[tileLayer setValue:@(indexPath.row) forKey:@"index"];
[tileLayer setNeedsDisplay];
return cell;
}

- (void)drawLayer:(CATiledLayer *)layer inContext:(CGContextRef)ctx
{
    //get image index
    NSInteger index = [[layer valueForKey:@"index"] integerValue];

    //load tile image
    NSString *imagePath = self.imagePaths[index];
    UIImage *tileImage = [UIImage imageWithContentsOfFile:imagePath];

    //calculate image rect
    CGFloat aspectRatio = tileImage.size.height / tileImage.size.width;
    CGRect imageRect = CGRectMakeZero;
    imageRect.size.width = layer.bounds.size.width;
    imageRect.size.height = layer.bounds.size.height * aspectRatio;
    imageRect.origin.y = (layer.bounds.size.height - imageRect.size.height)/2;

    //draw tile
    UIGraphicsPushContext(ctx);
    [tileImage drawInRect:imageRect];
    UIGraphicsPopContext();
}

```

```
}
```

```
@end
```

We've used some tricks here that are worth explaining:

- The `tileSize` property of `CATiledLayer` is measured in pixels, not points, so to ensure that the tile exactly matches the size of the cell, we've multiplied the size by the screen scale.
- In the `-drawLayer:inContext:` method, we need to know which `indexPath` the layer relates to so that we can load the correct image. We've taken advantage of the feature of `CALayer` that allows us to store and retrieve arbitrary values using KVC, and tagged each layer with the correct image index.

Despite our concerns, it turns out that `CATiledLayer` works very well in this case; the performance problems are gone, and the amount of code needed is comparable to the GCD approach. The only slight issue is that there is a notable fade-in as each image appears onscreen after loading (see Figure 14.4).

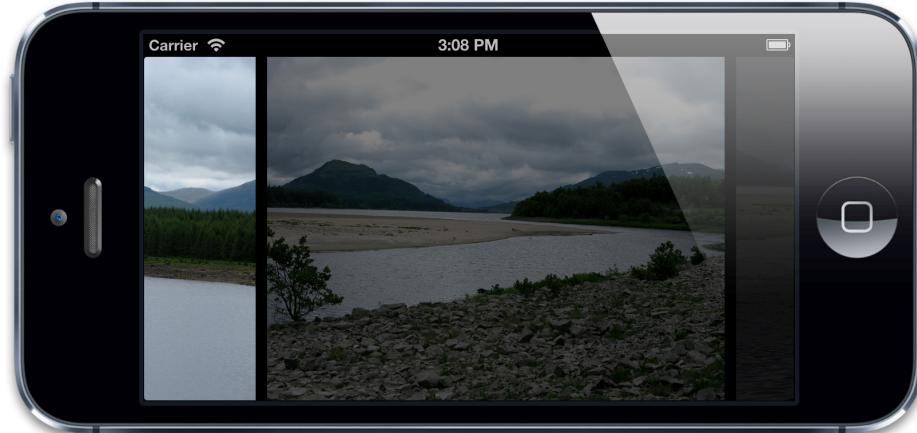


Figure 14.4 The images fading in as they are loaded

We can adjust the speed of the fade-in using the `CATiledLayer fadeDuration` property, or even remove the fade altogether, but it doesn't address the real issue: There will always be a delay between when the image begins loading and when it is ready to draw, and that will

result in *pop-in* of new images as we scroll. This problem isn't specific to `CATiledLayer`; it also affects our GCD-based version.

Even with all the image loading and caching techniques we've discussed, you will sometimes find that an image is simply too large to load and display in real time. As mentioned in Chapter 13, a full-screen Retina-quality image on an iPad has a resolution of 2048×1536 and consumes 12MB of RAM (uncompressed). The third-generation iPad's hardware simply is not capable of loading, decompressing, and displaying such an image within one-sixtieth of a second. Even if we load on a background thread to avoid animation stutter, we will still see gaps in our carousel.

We could display a placeholder image in the gap while the real image loads, but that's really just plastering over the problem. We can do better than that.

Resolution Swapping

Retina resolution (according to Apple's marketing) represents the smallest pixel size that the human eye is capable of distinguishing at a normal viewing distance. But that only applies to *static* pixels. When you observe a moving image, your eye is much less sensitive to detail, and a lower-resolution image is indistinguishable from Retina quality.

If we need to load and display very large moving images quickly, the simple solution is to *cheat*, and display smaller (or rather, *lower resolution*) images while the carousel is moving, and then swap in the full-res images when it comes to a stop. This means that we need to store two copies of each of our images at different resolutions, but fortunately that's common practice anyway as we still need to support both Retina and non-Retina devices.

If you are loading the image from a remote source, or the user's photo library, and don't already have a lower resolution version readily available, you can generate it dynamically by drawing the larger image into a smaller `CGContext` and then saving the resultant smaller image somewhere for later use.

To schedule the image swap, we can take advantage of a couple of delegate methods called by `UIScrollView` (as well as other scrollview-based controls such as `UITableView` and `UICollectionView`) as part of its `UIScrollViewDelegate` protocol:

```
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView  
    willDecelerate:(BOOL)decelerate;  
  
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView;
```

You can use these methods to detect when the carousel has come to rest, and defer loading the high-resolution versions of your images until that point. You'll find that the changeover is pretty much imperceptible as long as the low-res and high-res versions of the image match up perfectly in terms of color balance. (Be sure to generate them both on the same machine, using the same graphics application or script, so that they do.)

Caching

When you have a large number of images to display, it isn't practical to load them all in advance, but that doesn't mean that after you've gone to the trouble of loading them, you should just throw them away as soon as they move offscreen. By selectively *caching* the images after loading, you can avoid repeating the pop-in as users scroll back and forth across images that they've already seen.

Caching is simple *in principle*: You just store the result of an expensive computation (or a file that you've loaded from the flash drive or network) in memory so that when you need it again, it's quicker to access. The problem is that caching is essentially a tradeoff—you gain performance in return for consuming memory, but because memory is a limited resource, you can't just cache everything indefinitely.

Deciding when and what to cache (and for how long) is not always straightforward. Fortunately, most of the time, iOS takes care of image caching for us:

The `+imageNamed:` Method

We mentioned earlier that loading images using `[UIImage imageNamed:]` has the advantage that it decompresses the image immediately instead of deferring until it's drawn. But the `+imageNamed:` method has another significant benefit: It automatically caches the decompressed image in memory for future reuse, even if you don't retain any references to it yourself.

For the majority of images that you'll use in a typical iOS app (such as icons, buttons, and background images), loading the image using the `+imageNamed:` method is both the simplest and most performant approach. Images that you include in a nib file are loaded using the same mechanism, so often you'll use it implicitly without even realizing.

The `+imageNamed:` method isn't a magic bullet, though. It's optimized for user interface furniture and isn't appropriate for every type of image that an application might need to display. Here are a few reasons why it may be a good idea to implement your own image caching mechanism:

- The `+imageNamed:` method only works for images that are stored in the application bundle resources directory. In reality, most apps that display a lot of large images will need to load them from the Internet, or from the user's camera roll, so `+imageNamed:` won't work.
- The `+imageNamed:` cache is used to store all of your application interface images (buttons, backgrounds, and so on). If you fill the cache with large images like photographs, you increase the chances that iOS will remove those interface images to make room, which may lead to worse performance as you navigate around the app, as those images will then have to be reloaded. By using a separate cache for your carousel images, you can decouple their lifespan from the rest of your app images.

- The `+imageNamed:` caching mechanism is not public, so you have no fine-grained control. For example, you cannot test to see whether an image has already been cached before loading it, you cannot control the cache size, and you cannot remove objects from the cache when they are no longer needed.

Custom Caching

Building a bespoke caching system is *nontrivial*. Phil Karlton once said, “There are only two hard problems in computer science: cache invalidation and naming things.”

If we do decide to write our own image cache, how should we go about it? Let’s look at the challenges involved:

- **Choosing a suitable cache key**—The cache key is used to uniquely identify an image in the cache. If you are creating images at runtime, it’s not always obvious how to generate a string that will distinguish one cached image from another. In the case of our image carousel, though, it’s pretty straightforward because we can use either the image filename or the cell index.
- **Speculative caching**—If the effort of generating or loading data is high, you may want to load and cache it *before* it is needed the first time. Speculative preloading logic is inherently application specific, but in the case of our carousel, it is relatively simple to implement, because for a given position and scroll direction, we can determine exactly which images will be coming up next.
- **Cache invalidation**—If an image file changes, how do we know that our cached version needs to be updated? This is an extremely hard problem (as Phil Karlton quipped), but fortunately it’s not something we have to worry about when loading static images from our application resources. For user-supplied images (which may be modified or overwritten unexpectedly), a good solution is often to store a timestamp for when the image was cached and compare it with the modified date of the file.
- **Cache reclamation**—When you run out of cache space (memory), how do you decide what to throw away first? This may require you to write speculative algorithms to determine the relative likelihood of cached items to be reused. Thankfully, for the cache reclamation problem, Apple provides a handy general-purpose solution called `NSCache`.

NSCache

`NSCache` behaves a lot like an `NSDictionary`. You can insert and retrieve objects from the cache by key using the `-setObject:forKey:` and `-object:forKey:` methods, respectively. The difference is that unlike a dictionary, `NSCache` automatically discards stored objects when the system is low on memory.

The algorithm that `NSCache` uses to determine when to discard objects is not documented, but you can provide hints for how you would like it to behave using the `-setCountLimit:` method to set the total cache size and `-setObject:forKey:cost:` to specify a “cost” value for each stored object.

The *cost* is a numeric value that you can assign to an object to indicate the relative effort of recreating it. If you assign a large cost for large images, the cache knows that these are more expensive objects to store and that discarding a “costly” object will potentially have a greater performance impact than a “cheap” one. You can specify the total cache size in terms of cost instead of item count by using `-setTotalCostLimit:`.

`NSCache` is a general-purpose caching solution, and we could probably create a custom caching class that is better optimized for our specific carousel if we had to. (For example, we could determine which images to release first based on the difference between the cached image index and the currently centered index.) But `NSCache` should be sufficient for our current caching requirements; we don’t want to indulge in *premature optimization*.

Let’s extend our carousel example with an image cache and a basic speculative preloading implementation and see if that improves the pop-in effect for new images (see Listing 14.5).

Listing 14.5 Adding Caching and Speculative Loading

```
#import "ViewController.h"

@interface ViewController() <UICollectionViewDataSource>

@property (nonatomic, copy) NSArray *imagePaths;
@property (nonatomic, weak) IBOutlet UICollectionView *collectionView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    //set up data
    self.imagePaths =
        [[NSBundle mainBundle] pathsForResourcesOfType:@"png"
                                                inDirectory:@"Vacation Photos"];
}

//register cell class
[self.collectionView registerClass:[UICollectionViewCell class]
    forCellWithReuseIdentifier:@"Cell"];
}

- (NSInteger)collectionView:(UICollectionView *)collectionView
```

```

    numberOfRowsInSection:(NSInteger)section
{
    return [self.imagePaths count];
}

- (UIImage *)loadImageAtIndex:(NSUInteger)index
{
    //set up cache
    static NSCache *cache = nil;
    if (!cache)
    {
        cache = [[NSCache alloc] init];
    }

    //if already cached, return immediately
    UIImage *image = [cache objectForKey:@(index)];
    if (image)
    {
        return [image isKindOfClass:[NSNull class]]? nil: image;
    }

    //set placeholder to avoid reloading image multiple times
    [cache setObject:[NSNull null] forKey:@(index)];

    //switch to background thread
    dispatch_async(
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0), ^{
            //load image
            NSString *imagePath = self.imagePaths[index];
            UIImage *image = [UIImage imageWithContentsOfFile:imagePath];

            //redraw image using device context
            UIGraphicsBeginImageContextWithOptions(image.size, YES, 0);
            [image drawAtPoint:CGPointZero];
            image = UIGraphicsGetImageFromCurrentImageContext();
            UIGraphicsEndImageContext();

            //set image for correct image view
            dispatch_async(dispatch_get_main_queue(), ^{
                //cache the image
                [cache setObject:image forKey:@(index)];

                //display the image
                NSIndexPath *indexPath = [NSIndexPath indexPathForItem:

```

```

                index inSection:0];
        UICollectionViewCell *cell =
            [self.collectionView cellForItemAtIndexPath:indexPath];
        UIImageView *imageView = [cell.contentView.subviews lastObject];
        imageView.image = image;
    });
});

//not loaded yet
return nil;
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UICollectionViewCell *cell =
        [collectionView dequeueReusableCellWithReuseIdentifier:@"Cell"
                                                forIndexPath:indexPath];

    //add image view
    UIImageView *imageView = [cell.contentView.subviews lastObject];
    if (!imageView)
    {
        imageView = [[UIImageView alloc] initWithFrame:
                     cell.contentView.bounds];
        imageView.contentMode = UIViewContentModeScaleAspectFit;
        [cell.contentView addSubview:imageView];
    }

    //set or load image for this index
    imageView.image = [self loadImageAtIndex:indexPath.item];

    //preload image for previous and next index
    if (indexPath.item < [self.imagePaths count] - 1)
    {
        [self loadImageAtIndex:indexPath.item + 1];
    }
    if (indexPath.item > 0)
    {
        [self loadImageAtIndex:indexPath.item - 1];
    }
}

```

```
    return cell;
}

@end
```

Much better! There is still some pop-in if we scroll very quickly, but for normal scrolling it's now pretty rare, and the caching means that we're doing less loading anyway. Our preloading logic is very crude at the moment, and could be improved by taking into account the scroll speed and direction of the carousel, but it's already substantially better than our uncached version.

File Format

Image loading performance depends crucially on a tradeoff between the time taken to load a larger image file format and the time taken to decompress a smaller one. A lot of Apple documentation still states that PNG is the preferred format for all images on iOS, but this is outdated information and *grossly misleading*.

The lossless compression algorithm used by PNG images allows slightly faster decompression than the more complex lossy algorithm used for JPEG images, but this difference is usually dwarfed by the difference in loading time due to (relatively slow) flash storage access latency.

Listing 14.6 contains the code for a simple benchmarking app that loads images at various sizes and displays the time taken. To ensure a fair test, we measure the combined loading *and* drawing time of each image to ensure that the decompression performance of the resultant image is also taken into account. We also load and draw each image repeatedly for a duration of at least one second so that we can take the average loading time for a more accurate reading.

Listing 14.6 A Simple Image Loading Performance Benchmarking App

```
#import "ViewController.h"

static NSString *const ImageFolder = @"Coast Photos";

@interface ViewController () <UITableViewDataSource>

@property (nonatomic, copy) NSArray *items;
@property (nonatomic, weak) IBOutlet UITableView *tableView;

@end
```

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set up image names
    self.items = @[@"2048x1536", @"1024x768", @"512x384",
                  @"256x192", @"128x96", @"64x48", @"32x24"];
}

- (CFTimeInterval)loadImageForOneSec:(NSString *)path
{
    //create drawing context to use for decompression
    UIGraphicsBeginImageContext(CGSizeMake(1, 1));

    //start timing
    NSInteger imagesLoaded = 0;
    CFTimeInterval endTime = 0;
    CFTimeInterval startTime = CFAbsoluteTimeGetCurrent();
    while (endTime - startTime < 1)
    {
        //load image
        UIImage *image = [UIImage imageWithContentsOfFile:path];

        //decompress image by drawing it
        [image drawAtPoint:CGPointZero];

        //update totals
        imagesLoaded++;
        endTime = CFAbsoluteTimeGetCurrent();
    }

    //close context
    UIGraphicsEndImageContext();

    //calculate time per image
    return (endTime - startTime) / imagesLoaded;
}

- (void)loadImageAtIndex:(NSUInteger)index
{
    //load on background thread so as not to
    //prevent the UI from updating between runs
    dispatch_async(
```

```

dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
    //setup
    NSString *fileName = self.items[index];
    NSString *pngPath = [[NSBundle mainBundle] pathForResource:filename
                                                       ofType:@"png" inDirectory:ImageFolder];
    NSString *jpgPath = [[NSBundle mainBundle] pathForResource:filename
                                                       ofType:@"jpg" inDirectory:ImageFolder];

    //load
    NSInteger pngTime = [self loadImageForOneSec:pngPath] * 1000;
    NSInteger jpgTime = [self loadImageForOneSec:jpgPath] * 1000;

    //updated UI on main thread
    dispatch_async(dispatch_get_main_queue(), ^{
        //find table cell and update
        NSIndexPath *indexPath =
            [NSIndexPath indexPathForRow:index inSection:0];
        UITableViewCell *cell =
            [self.tableView cellForRowAtIndexPath:indexPath];
        cell.detailTextLabel.text =
            [NSString stringWithFormat:@"PNG: %03ims   JPG: %03ims",
             pngTime, jpgTime];
    });
});
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [self.items count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //dequeue cell
    UITableViewCell *cell =
        [self.tableView dequeueReusableCellWithIdentifier:@"Cell"];

    if (!cell)
    {
        cell = [[UITableViewCell alloc] initWithStyle:
            UITableViewCellStyleValue1 reuseIdentifier:@"Cell"];
    }
}

```

```

//set up cell
NSString *imageName = self.items[indexPath.row];
cell.textLabel.text = imageName;
cell.detailTextLabel.text = @"Loading...";

//load image
[self loadImageAtIndex:indexPath.row];

return cell;
}

@end

```

The PNG and JPEG compression algorithms are tuned for different image types: JPEG works well for noisy, imprecise images like photographs; and PNG is better suited to flat areas of color, sharp lines, or exact gradients. To make the benchmark fairer, we'll run it with a couple of different images: a photograph and a rainbow color gradient. The JPEG versions of each image were encoded using the default Photoshop "high-quality" setting of 60%. Figure 14.5 shows the results.

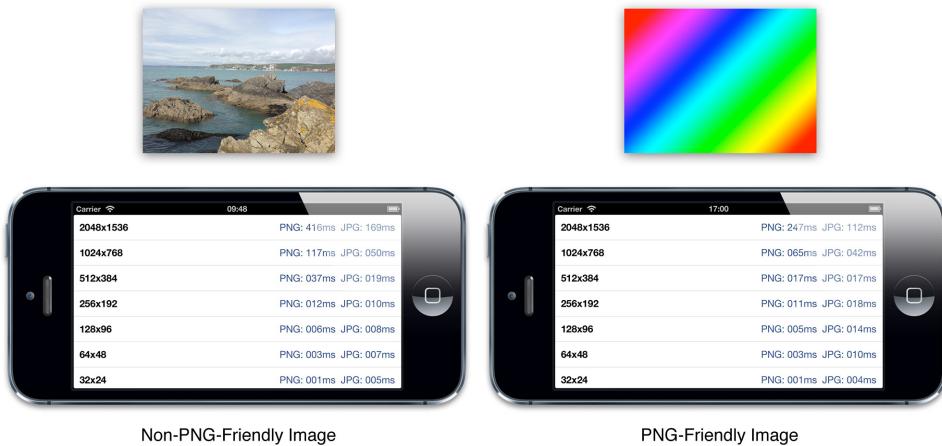


Figure 14.5 Relative loading performance for different types of images

As the benchmark demonstrates, for PNG-unfriendly images, JPEGs are consistently faster to load than PNGs of the same dimensions, unless the images are very small. For PNG-friendly images, they are still better for medium-to-large images.

In light of this, JPEG would have been a better choice for our image carousel app. If we had used JPEG rather than PNG, some of the threaded loading and caching tricks may not have been necessary at all.

Unfortunately, it's not always possible to use JPEG images. If the image requires transparency or has fine details that compress poorly using the JPEG algorithm, you have no choice but to use a different format. Apple has specifically optimized the PNG and JPEG loading code paths for iOS, so these are generally the preferred formats. That said, there are some other options available that can be useful in certain circumstances.

Hybrid Images

For images containing alpha transparency, it's possible to get the best of both worlds by using a PNG to compress the alpha channel and a JPEG to compress the RGB part of the image and then combine them after loading. This plays to the strengths of each format, and results in an image with close-to-PNG quality and close-to-JPEG file size and loading performance. Listing 14.7 shows the code to load a separate color and mask image and then combine them at runtime.

Listing 14.7 Creating a Hybrid Image from a PNG Mask and a JPEG

```
#import "ViewController.h"

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *imageView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //load color image
    UIImage *image = [UIImage imageNamed:@"Snowman.jpg"];

    //load mask image
    UIImage *mask = [UIImage imageNamed:@"SnowmanMask.png"];

    //convert mask to correct format
    CGColorSpaceRef graySpace = CGColorSpaceCreateDeviceGray();
    CGImageRef maskRef =
        CGImageCreateCopyWithColorSpace(mask.CGImage, graySpace);
    CGColorSpaceRelease(graySpace);
}
```

```

//combine images
CGImageRef resultRef = CGImageCreateWithMask(image.CGImage, maskRef);
UIImage *result = [UIImage imageWithCGImage:resultRef];
CGImageRelease(resultRef);
CGImageRelease(maskRef);

//display result
self.imageView.image = result;
}

@end

```

Using two separate files for each image in this way can be a bit cumbersome. The JPNG library (<https://github.com/nicklockwood/JPNG>) provides an open source, reusable implementation of this technique that reduces the burden by packing both images into a single file and adding support for loading hybrid images directly using the `+imageNamed:` and `+imageWithContentsOfFile:` methods.

JPEG 2000

iOS supports various image formats in addition to JPEG and PNG, such as TIFF and GIF, but for the most part, these are not worth bothering with as they have worse compression, quality, and performance tradeoffs than JPEG or PNG.

However, in iOS 5, Apple added support for the *JPEG 2000* image format. This was added with little fanfare and is not well known. It's not even properly supported by Xcode—JPEG 2000 images don't show up in Interface Builder.

JPEG 2000 images *do* work at runtime however (both on device and simulator), and they offer better image quality than JPEG for a given file size, as well as full support for alpha transparency. JPEG 2000 images are significantly slower to load and display than either PNG or JPEG, however, so they're only really a good option if reducing file size is a higher priority than runtime performance.

It's worth keeping an eye on JPEG 2000 in case it improves significantly in future iOS releases, but for now, hybrid images offer better performance for a similar file size and quality.

PVRTC

Every iOS device currently on the market uses an Imagination Technologies PowerVR graphics chip as its GPU. The PowerVR chip supports a proprietary image compression standard called PVRTC (PowerVR Texture Compression).

Unlike most image formats available on iOS, PVRTC images can be drawn directly to the screen without needing to be decompressed beforehand. This means that there is no decompression step after loading, and the size in memory is substantially smaller than any other image type (as little as one-sixteenth of the size, depending on the compression settings used).

There are several disadvantages to PVRTC, however:

- Although they consume less RAM when loaded, PVRTC files are larger than JPEG and may even be larger than PNG (depending on the contents) because the compression algorithm is optimized for performance rather than file size.
- PVRTC images must be exactly square and have power-of-two dimensions. If the source image doesn't meet these requirements, you'll have to stretch it or pad it with empty space before converting to PVRTC.
- The quality is not fantastic, especially for images with transparency. It generally looks like a heavily compressed JPEG file.
- PVRTC images cannot be drawn using Core Graphics, displayed in an ordinary `UIImageView` or used directly as a layer's contents. You have to load PVRTC images as an OpenGL texture and then map it onto a pair of triangles for display in a `CAEAGLLayer` or `GLKView`.
- Creating an OpenGL context for drawing a PVRTC image is quite expensive initially. Unless you plan to draw a lot of images into the same context, this may cancel out the performance benefit of using PVRTC.
- PVRTC images use an *asymmetric* compression algorithm. Although they *decompress* almost instantly, compressing them takes a phenomenally long time. On a modern, fast desktop Mac, it can take a minute or more to generate a single large PVRTC image. It is therefore not viable to generate them on-the-fly on an iOS device.

If you don't mind working with OpenGL, and have the luxury of generating your images in advance, PVRTC offers amazing loading performance compared to any other format available on iOS. For example, it's possible to load and display a 2048×2048 PVRTC image on the main thread in less than one-sixtieth of a second on a modern iOS device (that's more than big enough to fill a Retina iPad screen), avoiding a lot of the threading and caching complexity required when loading other formats.

Xcode includes a command line tool called *texturetool* for generating PVRTC images, but it's awkward to access (it's inside the Xcode application bundle) and rather limited in functionality. A better option is to use the Imagination Technologies *PVRTexTool*, which you can download for free as part of the PowerVR SDK from <http://www.imgtec.com/powervr/insider/sdkdownloads>.

After you've installed PVRTexTool, you can convert a suitably sized PNG image to a PVRTC file by using the following command in the Terminal:

```
/Applications/Imagination/PowerVR/GraphicsSDK/PVRTexTool/CL/OSX_x86/PVRTexToolCL -i  
{input_file_name}.png -o {output_file_name}.pvr -legacypvr -p -f PVRTC1_4 -q  
pvrtcbest
```

Listing 14.8 shows the code required to load and display a PVRTC image (adapted from the CAEAGLLayer example code in Chapter 6).

Listing 14.8 Loading and Displaying a PVRTC Image

```
#import "ViewController.h"  
#import <QuartzCore/QuartzCore.h>  
#import <GLKit/GLKit.h>  
  
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *glView;  
@property (nonatomic, strong) EAGLContext *glContext;  
@property (nonatomic, strong) CAEAGLLayer *glLayer;  
@property (nonatomic, assign) GLuint framebuffer;  
@property (nonatomic, assign) GLuint colorRenderbuffer;  
@property (nonatomic, assign) GLint framebufferWidth;  
@property (nonatomic, assign) GLint framebufferHeight;  
@property (nonatomic, strong) GLKBaseEffect *effect;  
@property (nonatomic, strong) GLKTextureInfo *textureInfo;  
  
@end  
  
@implementation ViewController  
  
- (void)setUpBuffers  
{  
    //set up frame buffer  
    glGenFramebuffers(1, &_framebuffer);  
    glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);  
  
    //set up color render buffer  
    glGenRenderbuffers(1, &_colorRenderbuffer);  
    glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);  
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,  
                             GL_RENDERBUFFER, _colorRenderbuffer);  
    [self.glContext renderbufferStorage:GL_RENDERBUFFER  
                           fromDrawable:self.glLayer];  
    glGetRenderbufferParameteriv(  
        GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH, &_framebufferWidth);  
    glGetRenderbufferParameteriv(  
        GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT, &_framebufferHeight);  
}
```

```

//check success
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
    NSLog(@"Failed to make complete framebuffer object: %i",
          glCheckFramebufferStatus(GL_FRAMEBUFFER));
}
}

- (void)tearDownBuffers
{
    if (_framebuffer)
    {
        //delete framebuffer
        glDeleteFramebuffers(1, &_framebuffer);
        _framebuffer = 0;
    }

    if (_colorRenderbuffer)
    {
        //delete color render buffer
        glDeleteRenderbuffers(1, &_colorRenderbuffer);
        _colorRenderbuffer = 0;
    }
}

- (void)drawFrame
{
    //bind framebuffer & set viewport
    glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);
    glViewport(0, 0, _framebufferWidth, _framebufferHeight);

    //bind shader program
    [self.effect prepareToDraw];

    //clear the screen
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(0.0, 0.0, 0.0, 0.0);

    //set up vertices
    GLfloat vertices[] =
    {
        -1.0f, -1.0f,
        -1.0f, 1.0f,
        1.0f, 1.0f,
        1.0f, -1.0f
    };
}

```



```

self.textureInfo = [GLKTextureLoader textureWithContentsOfFile:imageFile
    options:nil
    error:NULL];

//create texture
GLKEffectPropertyTexture *texture =
    [[GLKEffectPropertyTexture alloc] init];
texture.enabled = YES;
texture.envMode = GLKTextureEnvModeDecal;
texture.name = self.textureInfo.name;

//set up base effect
self.effect = [[GLKBaseEffect alloc] init];
self.effect.texture2d0.name = texture.name;

//set up buffers
[self setUpBuffers];

//draw frame
[self drawFrame];
}

- (void)viewDidUnload
{
    [self tearDownBuffers];
    [super viewDidUnload];
}

- (void)dealloc
{
    [self tearDownBuffers];
    [EAGLContext setCurrentContext:nil];
}

@end

```

As you can see, it's nontrivial to do this, but if you are interested in using PVRTC images in a regular app (as opposed to an OpenGL-based game), the GLView library (<https://github.com/nicklockwood/GLView>) provides a simple `GLImageView` class that replicates most of the functionality of `UIImageView`, but can display PVRTC images without requiring you to write any OpenGL code.

Summary

In this chapter, we investigated the performance problems relating to image loading and decompression and explored a range of different workarounds and solutions.

In Chapter 15, “Layer Performance,” we discuss performance issues relating to layer rendering and compositing.

This page intentionally left blank

Layer Performance

Doing more things faster is no substitute for doing the right things.

Stephen R. Covey

Chapter 14, “Image IO,” discussed how we can efficiently load and display images, with a view to avoiding performance glitches that might impact animation frame rate. In this final chapter, we study the performance of the layer tree itself and how to get the best out of it.

Inexplicit Drawing

The layer backing image can be drawn on-the-fly using Core Graphics, or set directly using the `contents` property by supplying an image loaded from a file, or drawn beforehand in an offscreen `CGContext`. In the previous two chapters, we talked about optimizing both of these scenarios. But in addition to *explicitly* creating a backing image, you can also create one *implicitly* through the use of certain layer properties, or by using particular view or layer subclasses.

It is important to understand exactly when and why this happens so that you can avoid accidentally introducing software drawing if it’s not needed.

Text

Both `CATextLayer` and `UILabel` draw their text directly into the backing image of the layer. These two classes actually use radically different approaches for rendering text: In iOS 6 and earlier, `UILabel` uses WebKit’s HTML rendering engine to draw its text, whereas `CATextLayer` uses Core Text. The latter is faster and should be used preferentially for any cases where you need to draw a lot of text, but they both require software drawing and are therefore inherently slow compared to hardware-accelerated compositing.

Wherever possible, try to avoid making changes to the frame of a view that contains text, because it will cause the text to be redrawn. For example, if you need to display a static block of text in the corner of a layer that frequently changes size, put the text in a sublayer instead.

Rasterization

We mentioned the `shouldRasterize` property of `CALayer` in Chapter 4, “Visual Effects,” as a way to solve blending glitches with overlapping translucent layers, and again in Chapter 12, “Tuning for Speed,” as a performance optimization technique when drawing complex layer subtrees.

Enabling the `shouldRasterize` property causes the layer to be drawn into an offscreen image. That image will then be cached and drawn in place of the actual layer’s contents and sublayers. If there are a lot of sublayers or they have complex effects applied, this is generally less expensive than redrawing everything every frame. But it takes time to generate that rasterized image initially, and it will consume additional memory.

Rasterizing can provide a big performance boost when used appropriately (as you saw in Chapter 12), but it’s very important to avoid rasterizing layers whose content changes every frame because it will negate any caching benefit, and actually make the performance worse.

To test whether you are using rasterization appropriately, use the Color Hits Green and Misses Red instrument to see if the rasterized image cache is being frequently flushed (which would indicate that the layer is either not a good candidate for rasterization, or that you are unwittingly making changes to it that are unnecessarily causing it to be redrawn).

Offscreen Rendering

Offscreen rendering is invoked whenever the combination of layer properties that have been specified mean that the layer cannot be drawn directly to the screen without pre-compositing. Offscreen rendering does not necessarily imply *software* drawing, but it means that the layer must first be rendered (either by the CPU or GPU) into an offscreen context before being displayed. The layer attributes that trigger offscreen rendering are as follows:

- Rounded corners (when combined with `masksToBounds`)
- Layer masks
- Drop shadows

Offscreen rendering is similar to what happens when we enable rasterization, except that the drawing is not normally as expensive as rasterizing a layer, the sublayers are not affected, and the result is not cached, so there is no long-term memory hit as a result. Too many layers being rendered offscreen will impact performance significantly, however.

It can sometimes be beneficial to enable rasterization as an optimization for layers that require offscreen rendering, but only if the layer/sublayers do not need to be redrawn frequently.

For layers that require offscreen rendering *and* need to animate (or which have animated sublayers), you may be able to use `CAShapeLayer`, `contentsCenter`, or `shadowPath` to achieve a similar appearance with less of a performance impact.

CAShapeLayer

Neither `cornerRadius` nor `masksToBounds` impose any significant overhead on their own, but when combined, they trigger offscreen rendering. You may sometimes find that you want to display rounded corners and clip sublayers to the layer bounds, but you don't necessarily need to clip to the rounded corners, in which case you can avoid this overhead by using `CAShapeLayer`.

You can get the effect of rounded corners and still clip to the (rectangular) bounds of the layer without incurring a performance overhead by drawing the rounded rectangle using the handy `+bezierPathWithRoundedRect:cornerRadius:` constructor for `UIBezierPath` (see Listing 15.1). This is no faster than using `cornerRadius` in itself, but means that the `masksToBounds` property no longer incurs a performance penalty.

Listing 15.1 Drawing a Rounded Rectangle Using CAShapeLayer

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create shape layer
    CAShapeLayer *blueLayer = [CAShapeLayer layer];
    blueLayer.frame = CGRectMake(50, 50, 100, 100);
    blueLayer.fillColor = [UIColor blueColor].CGColor;
    blueLayer.path = [UIBezierPath bezierPathWithRoundedRect:
                      CGRectMake(0, 0, 100, 100) cornerRadius:20].CGPath;
```

```
//add it to our view
    [self.layerView.layer addSublayer:blueLayer];
}

@end
```

Stretchable Images

Another way to create a rounded rectangle is by using a circular contents image combined with the `contentsCenter` property mentioned in Chapter 2, “The Backing Image,” to create a stretchable image (see Listing 15.2). In theory, this should be slightly faster to render than using a `CAShapeLayer` because drawing a stretchable image only requires 18 triangles (a stretchable image is rendered using nine rectangles arranged in a 3x3 grid), whereas many more are needed to render a smooth curve. In practice, the difference is unlikely to be significant.

Listing 15.2 Drawing a Rounded Rectangle Using a Stretchable Image

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create layer
    CALayer *blueLayer = [CALayer layer];
    blueLayer.frame = CGRectMake(50, 50, 100, 100);
    blueLayer.contentsCenter = CGRectMake(0.5, 0.5, 0.0, 0.0);
    blueLayer.contentsScale = [UIScreen mainScreen].scale;
    blueLayer.contents =
        (__bridge id)[UIImage imageNamed:@"Circle.png"].CGImage;

    //add it to our view
    [self.layerView.layer addSublayer:blueLayer];
}

@end
```

The advantage of using stretchable images over the other techniques is that they can be used to draw an arbitrary border effect at no extra cost to performance. So, for example, a stretchable image could also be used to efficiently create a rectangular drop shadow effect.

shadowPath

We mentioned the `shadowPath` property in Chapter 2. If your layer is a simple geometric shape like a rectangle or rounded rectangle (which it will be if it doesn't contain any transparent parts or sublayers), it is easy to create a shadow path that matches its shape and this will greatly simplify the calculations that Core Animation has to do to draw the shadow, avoiding the need to precompose the layer offscreen. This makes a huge difference to the performance.

If your layer has a more complex shape, it might be awkward to generate the correct shadow path, in which case you may want to consider pregenerating your shadow as a background image using a paint program.

Blending and Overdraw

As mentioned in Chapter 12, there is a limit to the number of pixels that the GPU can draw each frame (known as the *fill rate*), and while it can comfortably draw an entire screen full of pixels, it might begin to lag if it needs to keep repainting the same area multiple times due to overlapping layers (*overdraw*).

The GPU will discard pixels in layers that are fully obscured by another layer, but calculating whether a layer is obscured can be complicated and processor intensive. Merging together the colors from several overlapping translucent pixels (*blending*) is also expensive. You can help to speed up the process by ensuring that layers do not make use of transparency unless they need to. Whenever possible, you should do the following:

- Set the `backgroundColor` of your view to a fixed, opaque color.
- Set the `opaque` property of the view to YES.

This reduces blending (because the compositor knows that nothing behind the layer will contribute to the eventual pixel color) and speeds up the calculations for avoiding overdraw because Core Animation can discard any completely obscured layers in their entirety instead of having to test each overlapping pixel individually.

If you are using images, try to avoid alpha transparency unless it is strictly needed. If the image will appear in front of a fixed background color, or a static background image that doesn't need to move relative to the foreground, you can prefill the image background and avoid runtime blending.

If you are using text, a `UILabel` with a white background (or any other solid color) is more efficient to draw than one with a transparent background.

Finally, by judicious use of the `shouldRasterize` property, you can collapse a static layer hierarchy into a single image that doesn't need to be recomposed each frame, avoiding any performance penalties due to blending and overdraw between the sublayers.

Reducing Layer Count

Due to the overhead of allocating layers, preprocessing them, packaging them up to be sent over IPC to the render server, and then converting them to OpenGL geometry, there is a practical upper limit on the number of layers you can display onscreen at once.

The exact limit will depend on the iOS device, the type of layer, and the layer contents and properties, but in general once the number of layers runs into the hundreds or thousands, you are going to start to see performance problems even if the layers themselves are not doing anything particularly expensive.

Clipping

Before doing any other kind of optimization to your layers, the first thing to check is that you are not creating and attaching layers to the window if they will not be visible. Layers might be invisible for a variety of reasons, such as the following:

- They lie outside of the bounds of the screen, or the bounds of their parent layer.
- They are completely obscured by another opaque layer.
- They are fully transparent.

Core Animation does a fairly good job of culling layers that aren't going to contribute to the visible scene, but your code can usually determine if a layer isn't going to be needed earlier than Core Animation can. Ideally, you want to determine this before the layer object is ever created, to avoid the overhead of creating and configuring the layer unnecessarily.

Let's try an example. Listing 15.3 shows the code for creating a simple scrolling 3D matrix of layers. This looks pretty cool, especially when it's moving (see Figure 15.1), but it's not particularly expensive to draw because each layer is just a simple colored rectangle.

Listing 15.3 Drawing a 3D Matrix of Layers

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

#define WIDTH 10
#define HEIGHT 10
#define DEPTH 10

#define SIZE 100
#define SPACING 150

#define CAMERA_DISTANCE 500

@interface ViewController ()
```

```

@property (nonatomic, strong) IBOutlet UIScrollView *scrollView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set content size
    self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING,
                                             (HEIGHT - 1)*SPACING);

    //set up perspective transform
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / CAMERA_DISTANCE;
    self.scrollView.layer.sublayerTransform = transform;

    //create layers
    for (int z = DEPTH - 1; z >= 0; z--)
    {
        for (int y = 0; y < HEIGHT; y++)
        {
            for (int x = 0; x < WIDTH; x++)
            {
                //create layer
                CALayer *layer = [CALayer layer];
                layer.frame = CGRectMake(0, 0, SIZE, SIZE);
                layer.position = CGPointMake(x*SPACING, y*SPACING);
                layer.zPosition = -z*SPACING;

                //set background color
                layer.backgroundColor =
                    [UIColor colorWithRed:(1.0-z*(1.0/DEPTH)) green:1.0 blue:1.0 alpha:1].CGColor;

                //attach to scroll view
                [self.scrollView.layer addSublayer:layer];
            }
        }
    }
}

```

```
//log  
    NSLog(@"displayed: %i", DEPTH*HEIGHT*WIDTH);  
}  
  
@end
```

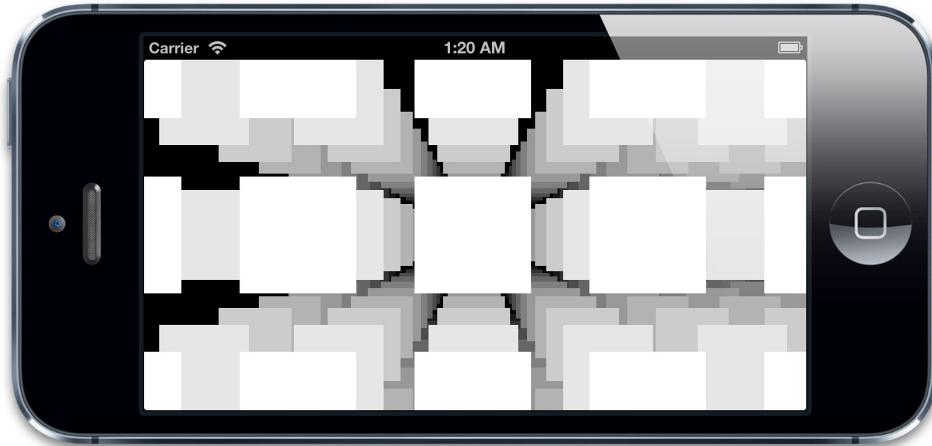


Figure 15.1 A scrolling 3D matrix of layers

The `WIDTH`, `HEIGHT`, and `DEPTH` constants control the number of layers being generated. In this case, we have $10 \times 10 \times 10$ layers, so 1000 in total, of which only a few hundred will be visible onscreen at a time.

If we increase the `WIDTH` and `HEIGHT` constants to 100, our app slows down to a crawl. We now have 100,000 layers being created, so it's not surprising that the performance has degraded.

But the number of layers that are actually visible onscreen has not increased at all, so nothing extra is being drawn. The reason the app is now slow is due to the sheer effort of managing all of those layers. Most of them don't contribute to the rendering effort, but by forcing Core Animation to calculate the position of each of those layers before discarding them, we've killed our frame rate.

Because we know that our layers are arranged in a uniform grid, we can determine mathematically which ones will be visible onscreen without needing to actually create them or calculate their positions individually. The calculation is nontrivial because it has to take perspective into account. If we do that work upfront, however, it will save Core Animation

from having to do something much more complex later on, and avoid us having to unnecessarily create and position layer objects that we don't need.

Let's refactor our app so that layers are instantiated dynamically as the view is scrolled instead of all being allocated in advance. That way, we can calculate whether they are needed before ever creating them. Next, we'll add some code to calculate the visible area so that we can eliminate layers that are outside the field of view. Listing 15.4 shows the updated code.

Listing 15.4 Eliminating Layers Outside of the Screen Bounds

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

#define WIDTH 100
#define HEIGHT 100
#define DEPTH 10

#define SIZE 100
#define SPACING 150

#define CAMERA_DISTANCE 500
#define PERSPECTIVE(z) (float)CAMERA_DISTANCE/(z + CAMERA_DISTANCE)

@interface ViewController () <UIScrollViewDelegate>

@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //set content size
    self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING,
                                             (HEIGHT - 1)*SPACING);

    //set up perspective transform
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / CAMERA_DISTANCE;
    self.scrollView.layer.sublayerTransform = transform;
}
```

```
- (void)viewDidLayoutSubviews
{
    [self updateLayers];
}

- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
    [self updateLayers];
}

- (void)updateLayers
{
    //calculate clipping bounds
    CGRect bounds = self.scrollView.bounds;
    bounds.origin = self.scrollView.contentOffset;
    bounds = CGRectInset(bounds, -SIZE/2, -SIZE/2);

    //create layers
    NSMutableArray *visibleLayers = [NSMutableArray array];
    for (int z = DEPTH - 1; z >= 0; z--)
    {
        //increase bounds size to compensate for perspective
        CGRect adjusted = bounds;
        adjusted.size.width /= PERSPECTIVE(z*SPACING);
        adjusted.size.height /= PERSPECTIVE(z*SPACING);
        adjusted.origin.x -= (adjusted.size.width - bounds.size.width) / 2;
        adjusted.origin.y -= (adjusted.size.height - bounds.size.height) / 2;

        for (int y = 0; y < HEIGHT; y++)
        {
            //check if vertically outside visible rect
            if (y*SPACING < adjusted.origin.y ||
                y*SPACING >= adjusted.origin.y + adjusted.size.height)
            {
                continue;
            }

            for (int x = 0; x < WIDTH; x++)
            {
                //check if horizontally outside visible rect
                if (x*SPACING < adjusted.origin.x ||
                    x*SPACING >= adjusted.origin.x + adjusted.size.width)
                {
                    continue;
                }
            }
        }
    }
}
```

```

    //create layer
    CALayer *layer = [CALayer layer];
    layer.frame = CGRectMake(0, 0, SIZE, SIZE);
    layer.position = CGPointMake(x*SPACING, y*SPACING);
    layer.zPosition = -z*SPACING;

    //set background color
    layer.backgroundColor =
        [UIColor colorWithRed:1.0-z*(1.0/DEPTH) alpha:1].CGColor;

    //attach to scroll view
    [visibleLayers addObject:layer];
}
}

//update layers
self.scrollView.layer.sublayers = visibleLayers;

//log
NSLog(@"displayed: %i/%i", [visibleLayers count], DEPTH*HEIGHT*WIDTH);
}

@end

```

The mathematics of the calculation used here are very specific to this particular problem, but the principle is applicable to other situations, as well. (When you use a UITableView or UICollectionView, it does something similar behind the scenes to work out which of the cells need to be displayed.) The result is that our app can now handle hundreds of thousands of “virtual” layers without any performance problems because it doesn’t ever need to instantiate more than a few hundred of them at a time.

Object Recycling

Another trick that we can use when managing a large number of similar views or layers is to *recycle* them. Object recycling is quite a common pattern in iOS; it’s used for UITableView and UICollectionView cells, and for the annotation pins in MKMapView, along with many other examples.

The basic principle of object recycling is that you create a *pool* of identical objects. When you finish with a particular instance of an object (a layer in this case), you add it to the object pool. Each time you need an instance, you take one out of the pool. Only if the pool is empty do you create a new one.

The advantage of this is that you avoid the overhead of constantly creating and releasing objects (which is expensive because it requires the allocation/deallocation of memory) and you avoid having to re-apply properties that don't vary between instances.

Let's update our matrix example to use an object pool (see Listing 15.5).

Listing 15.5 Reducing Unnecessary Object Allocation by Recycling

```
@interface ViewController () <UIScrollViewDelegate>

@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;
@property (nonatomic, strong) NSMutableSet *recyclePool;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    //create recycle pool
    self.recyclePool = [NSMutableSet set];

    //set content size
    self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING,
                                             (HEIGHT - 1)*SPACING);

    //set up perspective transform
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / CAMERA_DISTANCE;
    self.scrollView.layer.sublayerTransform = transform;
}

- (void)viewDidLayoutSubviews
{
    [self updateLayers];
}

- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
    [self updateLayers];
}

- (void)updateLayers
{
```

```

//calculate clipping bounds
CGRect bounds = self.scrollView.bounds;
bounds.origin = self.scrollView.contentOffset;
bounds = CGRectInset(bounds, -SIZE/2, -SIZE/2);

//add existing layers to pool
[self.recyclePool addObject:self.scrollView.layer.sublayers];

//disable animation
[CATransaction begin];
[CATransaction setDisableActions:YES];

//create layers
NSInteger recycled = 0;
NSMutableArray *visibleLayers = [NSMutableArray array];
for (int z = DEPTH - 1; z >= 0; z--)
{
    //increase bounds size to compensate for perspective
    CGRect adjusted = bounds;
    adjusted.size.width /= PERSPECTIVE(z*SPACING);
    adjusted.size.height /= PERSPECTIVE(z*SPACING);
    adjusted.origin.x -= (adjusted.size.width - bounds.size.width) / 2;
    adjusted.origin.y -= (adjusted.size.height - bounds.size.height) / 2;

    for (int y = 0; y < HEIGHT; y++)
    {
        //check if vertically outside visible rect
        if (y*SPACING < adjusted.origin.y ||
            y*SPACING >= adjusted.origin.y + adjusted.size.height)
        {
            continue;
        }

        for (int x = 0; x < WIDTH; x++)
        {
            //check if horizontally outside visible rect
            if (x*SPACING < adjusted.origin.x ||
                x*SPACING >= adjusted.origin.x + adjusted.size.width)
            {
                continue;
            }

            //recycle layer if available
            CALayer *layer = [self.recyclePool anyObject];
            if (layer)
            {

```

```

        recycled++;
        [self.recyclePool removeObject:layer];
    }
    else
    {
        //otherwise create a new one
        layer = [CALayer layer];
        layer.frame = CGRectMake(0, 0, SIZE, SIZE);
    }

    //set position
    layer.position = CGPointMake(x*SPACING, y*SPACING);
    layer.zPosition = -z*SPACING;

    //set background color
    layer.backgroundColor =
        [UIColor colorWithRed:(1.0-z*(1.0/DEPTH)) green:1.0 blue:1.0 alpha:1].CGColor;

    //attach to scroll view
    [visibleLayers addObject:layer];
}
}

//CATransaction commit];

//update layers
self.scrollView.layer.sublayers = visibleLayers;

//log
NSLog(@"displayed: %i/%i recycled: %i",
       [visibleLayers count], DEPTH*HEIGHT*WIDTH, recycled);
}

@end

```

In this case, we only have one type of layer object, but UIKit sometimes uses an identifier string to distinguish between multiple recyclable object types stored in separate pools.

You might have noticed that we're now using a `CATransaction` to suppress animation when setting the layer properties. This wasn't needed before because we were only ever setting properties on our layers once, prior to attaching them to the display. But now that the layers are being recycled, it has become necessary to disable the implicit animation that would normally occur when a visible layer's properties are modified.

Core Graphics Drawing

After you have eliminated views or layers that are not contributing to the display onscreen, there might still be ways that you can reduce the layer count further. For example, if you are using multiple `UILabel` or `UIImageView` instances to display static content, you can potentially replace it all with a single view that uses `-drawRect:` to replicate the appearance of a complex view hierarchy.

It might seem counterintuitive to do this because we know that software drawing is slower than GPU compositing and requires additional memory, but in a situation where the performance is limited by the number of layers, software drawing may actually *improve* the performance by avoiding excessive layer allocation and manipulation.

Doing the drawing yourself in this case involves a similar performance tradeoff to rasterizing, but means that you can remove sublayers from the layer tree altogether (as opposed to just obscuring them, as you do when using `shouldRasterize`).

The `-renderInContext:` Method

Using Core Graphics to draw a static layout may sometimes be faster than using a hierarchy of `UIView` instances, but using `UIView` instances is both more concise and more flexible than writing the equivalent drawing code by hand, especially if you use Interface Builder to do the layout. It would be a shame to have to sacrifice those benefits for the sake of performance tuning.

Fortunately, you don't have to. Having a large number of views or layers is only a problem if the layers are *actually attached to the screen*. Layers that are not connected to the layer tree don't get sent to the render server and won't impact performance (after they've been initially created and configured).

By using the `CALayer -renderInContext:` method, you can draw a snapshot of a layer and its sublayers into a Core Graphics context and capture the result as an image, which can then be displayed directly inside a `UIImageView`, or as the contents of another `CALayer`. Unlike using `shouldRasterize`—which still requires that the layers be attached to the layer tree—with this approach there is no ongoing performance cost.

Responsibility for refreshing this image when the layer content changes would be up to you (unlike using `shouldRasterize`, which handles caching and cache invalidation automatically), but once the image has initially been generated, you save significant per-frame performance overhead with this approach versus asking Core Animation to maintain a complex layer tree.

Summary

This chapter examined the common performance bottlenecks when using Core Animation layers and discussed how to avoid or mitigate them. You learned how to manage scenes containing thousands of virtual layers by only creating a few hundred real ones. You also

learned some useful tricks for redistributing work between the CPU and GPU by selectively rasterizing or drawing layer contents when appropriate.

And that's all there is to say about Core Animation (at least until Apple invents something new for us to play with).

Where Are the Companion Content Files?

Thank you for purchasing this digital version of
iOS Core Animation: Advanced Techniques



As an eBook reader, you have access to these files by following the steps below.

1. On your PC or MAC, open a web browser and go to this URL:
www.informit.com/title/9780133440751
Navigate to the Downloads tab and click on the link.
2. Download the Zip file (or files) from the web site to your hard drive.
3. Unzip the files and follow the directions for use in the README file included in the download.

Please note that many of our companion content files can be very large, especially image and video files.

If you are unable to locate the files for this title by following the steps at left, please visit www.informit.com/about/contact_us, select “Digital Products Help,” and enter in the Comments box the URL from step 1. Our customer service representatives will assist you.

The Professional and Personal Technology Brands of Pearson

Take Our Survey

Thank you for purchasing *iOSCore Animation: Advanced Techniques*. We would like to hear what you think about this eBook. Your feedback allows us to improve our products and your learning experience.

To complete the short survey, go to <https://www.surveymonkey.com/s/ZNR36TM>. After completing the survey, you will receive a 50% discount code that can be applied to a future purchase on InformIT.com.

This survey will close October 31, 2013. The discount code will expire December 31, 2013.