

CORE FRAMEWORKS SERIES



[SIMPLIFIED ANIMATION TECHNIQUES
FOR MAC AND IPHONE DEVELOPMENT]

CORE ANIMATION

marcus ZARRA
matt LONG

Core Animation: Simplified Animation Techniques for Mac® and iPhone® Development

Copyright © 2010 Pearson Education, Inc.

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.

Photographs of Marcus Zarra and Matt Long on the back cover: Copyright © 2009 LIZography. All Rights Reserved.

The authors 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@pearson.com

Visit us on the Web: www.informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Zarra, Marcus, 1970-

Core animation : simplified animation techniques for Mac and iPhone development / Marcus Zarra, Matt Long.
p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-61775-0 (pbk. : alk. paper) 1. Computer animation 2. Core animation (Application development environment) 3. Application program interfaces (Computer software) 4. Mac OS. 5. iPhone OS. I. Long, Matt, 1973- II. Title.

TR897.7.Z37 2010

006.6'96—dc22

2009038600

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-61775-0

ISBN-10: 0-321-61775-4

Text printed in the United States on recycled paper at Courier in Kendallville, Indiana.

First printing December 2009

Editor-in-Chief
Karen Gettman

Senior Acquisitions Editor
Chuck Toporek

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editors
Julie Anderson
Jovana San Nicolas-Shirley

Copy Editor
Apostrophe Editing
Services

Indexer
Erika Millen

Proofreader
Sheri Cain

Editorial Assistant
Romny French

Interior Designer
Gary Adair

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

CHAPTER 3

Basic Animations

Core Animation is a powerful and mature technology that enables you to create animations that are as simple as you like or as complex as you need. To perform simple animations on windows and views, Apple provides the animation proxy object that, when called, causes an implicit animation to play when some visual component such as the view frame, opacity, or location is changed. For basic layer animation, the `CABasicAnimation` class provides a way to animate between two values, a starting value and an ending value. In this chapter, we look at these most basic methods for implementing animation in your application.

The Simplest Animations

With Core Animation integrated into Cocoa, you can animate windows, views, and layers implicitly by simply setting the value of the parameter you are interested in animating to some new value. When using a layer (`CALayer`), all you need to do is set the value with a direct call. For example, if you want to change the bounds of a layer, you simply call `[layer setBounds:newFrame]` where `layer` is the `CALayer` object you've created and added to your layer tree and `newFrame` is a `CGRect` containing the values of the new bound's size and origin. When this code is run, the change to the bounds of the layer is animated using the default animation for the keypath "bounds."

Similarly, when using a window (`NSWindow`) or view (`NSView`), all you need to do is set the value of the window or view property using the animation proxy object. This means that instead of calling `[view setFrame:newFrame]` to

IN THIS CHAPTER

- ▶ The Simplest Animations
- ▶ The Animation Proxy Object
- ▶ The Differences Between Window, View, and Layer Animation
- ▶ Preparing a View to Perform Layer Animation
- ▶ Using `CABasicAnimation`
- ▶ Useful Animation Properties
- ▶ Animation Grouping

set the view frame, for example, you instead call `[[view animator] setFrame:newFrame]`. The difference is that we have instructed the view's animator proxy object to set the property for us—which implicitly animates the value from the current value to the value specified in `newFrame`.

The Animation Proxy Object

So what is the animator proxy object? The animator proxy object is available in both `NSView` and `NSWindow`. It implements the protocol `NSAnimatablePropertyContainer`. This container uses Key-Value Coding to set the actual value of whatever parameter was specified while doing the value interpolation and animation behind the scenes.

As the name implies, the animator proxy acts as an agent that takes the value you give it and handles animating the property from the starting or current value to the value specified. It then sets the property as if you had called `set` on the property explicitly.

The Differences Between Window, View, and Layer Animation

The idea behind animation in windows, views, and layers is the same; however, the implementation differs. In this section, we discuss one of the most common animations you will likely want to implement—frame resizing.

Window Resizing

Since the first version of Mac OS X, the ability to animate a window's frame has been available to developers in the method `-(void)setFrame:(NSRect>windowFrame display:(BOOL)displayViews animate:(BOOL)performAnimation`. The first parameter is the new frame you are animating to. The second parameter tells the window to call `-displayIfNeeded` on all of its subviews, and the third parameter tells the window to animate the transition from its current frame to the frame specified in the first parameter. If this last parameter is set to `NO`, the change to the new frame happens immediately rather than progressively with animation.

NOTE

This call is different than what you use for changing the frame in both `NSViews` and `CALayers`. They both have a method called `-setFrame`. We discuss that more in moment.

With this built-in window frame resizing capability, why would you need to use Core Animation for changing a window's frame? The answer is, simply, you don't. For many cases when resizing, you can use the built-in functionality and you probably should. There may be times, however, when you want more control over animating windows. Keep several things in mind when doing so. `NSWindow` has an animator proxy just like `NSView`. When you call the animator, it animates the parameter you specified, but the

parameter is the catch. If you want to move the window to a different position on the screen, for instance, you can either call `-(void)setFrame:(CGRect)windowFrame display:(BOOL)displayViews` (notice the missing third parameter) on the animator proxy object, or you can add an animation to the animations dictionary of the window itself.

First, let's look at how to use the animator proxy. Take a look at the following.

```
[[window animator] setFrame:newFrame display:YES];
```

This makes it simple to animate the frame.

By default, the animation plays back over the course of 0.25 seconds. If you want to change the duration, use an `NSAnimationContext` object, which is the `NSView/NSWindow` counterpart to the `CATransaction`. If we wrap the call to `-setFrame` in an `NSAnimationContext`, the animation runs at the duration we specify. Listing 3-1 demonstrates how to do this.

LISTING 3-1 Wrap Frame Change in an `NSAnimationContext`

```
[NSAnimationContext beginGrouping];
[[NSAnimationContext currentContext] setDuration:5.0f];
[[window animator] setFrame:newFrame display:YES];
[NSAnimationContext endGrouping];
```

This causes the frame to change over the course of 5 seconds rather than the default of 0.25 seconds. As you see in the next section, this grouping mechanism is also what you use when you want to change the duration of an animation for an `NSView`.

Basic animation using Core Animation can also be used on windows and views, but there is a slight difference in how the animation is set up. As an alternative to calling `-setFrame` on the window animator proxy, we can create a `CABasicAnimation` and animate the `frame` property. Take a look at Listing 3-2 to see how to create, add, and run a basic animation on a window.

LISTING 3-2 Adding an Animation to the Window Animations Dictionary

```
CABasicAnimation *animation =
    [CABasicAnimation animationWithKeyPath:@"frame"];
[animation setFromValue:[NSValue valueWithRect:oldFrame]];
[animation setToValue:[NSValue valueWithRect:newFrame]];
[animation setDuration:5.0f];
[window setAnimations:[NSDictionary animation forKey:@"frame"]];
[[window animator] setFrame:newFrame display:YES];
```

The visual effect is identical to what you see occur when running the code in Listing 3-1.

View Resizing

Views can be resized the same as windows can, but the keypath you use differs. You can call `-setFrame` on a view using the same code you used for a window, as shown in Listing 3-3.

LISTING 3-3 Animate View Frame Change in an NSAnimationContext

```
[NSAnimationContext beginGrouping];
[[NSAnimationContext currentContext] setDuration:5.0f];
[[view animator] setFrame:newFrame display:YES];
[NSAnimationContext endGrouping];
```

The only difference between this code and the code in Listing 3-1 is the object we're calling `-setFrame` on—a view in this case.

If you want to use explicit animation, instead of animating the frame, animate the `frameOrigin` and the `frameSize`. Listing 3-4 shows how to animate both of these properties.

LISTING 3-4 Explicitly Animating Frame Origin and Size

```
CABasicAnimation *originAnimation = [CABasicAnimation
    animationWithKeyPath:@"frameOrigin"];
[originAnimation setFromValue:[NSValue
    valueWithPoint:oldImageFrame.origin]];
[originAnimation setToValue:[NSValue valueWithPoint:newFrame.origin]];
[originAnimation setDuration:5.0];

CABasicAnimation *sizeAnimation = [CABasicAnimation
    animationWithKeyPath:@"frameSize"];
[sizeAnimation setFromValue:
[NSValue valueWithSize:oldImageFrame.size]];
[sizeAnimation setToValue:[NSValue valueWithSize:newFrame.size]];
[sizeAnimation setDuration:5.0];

[[view animator] setAnimations:[NSDictionary
    dictionaryWithObjectsAndKeys:originAnimation,
    @"frameOrigin",
    sizeAnimation,
    @"frameSize",
    nil]];

[[view animator] setFrame:newFrame];
```

Layer Resizing

Animating a layer's frame is a bit different from doing the same in windows and views. There is no animator proxy available in a `CALayer` object, but rather animation is always used when you make an explicit change to a property. In fact, if you don't want animation used, you have to explicitly turn animation off. Listing 3-5 demonstrates how to do this.

LISTING 3-5 Explicitly Disabling Layer Animation

```
[CATransaction begin]
[CATransaction setValue:[NSNumber numberWithInt:YES]
                      forKey:kCATransactionDisableActions]
[layer setBounds:bounds];
[CATransaction commit];
```

NOTE

Notes on Disabling Animations

Alternatively, you can disable animations in a layer based on a keypath by using the delegate method:

```
- (id<CAAction>)actionForLayer:(CALayer *)layer forKey:(NSString *)key
```

It returns an object that implements the `CAAction` protocol. It might also return `NSNull`, which in effect disables the animation for the key specified in the `key` parameter of the delegate method. When you implement this delegate method, simply check to see if the layer passed in is the one you are working with, and then check to see if the `key` field is the same as the keypath for which you want to disable animation. If it is, return `NSNull`.

The `CATransaction` class is the Core Animation analogue to AppKit's `NSAnimationContext` object we used in Listing 3-2 and 3-4 for windows and views. Just like `NSAnimationContext`, `CATransaction` enables us to set the animation duration. Listing 3-6 demonstrates how to do this.

LISTING 3-6 Setting Animation Duration in a Layer

```
[CATransaction begin]
[CATransaction setValue:[NSNumber numberWithFloat:5.0f]
                      forKey:kCATransactionAnimationDuration]
[layer setBounds:bounds];
[CATransaction commit];
```

As you might suspect, we can also animate properties of a layer explicitly. To achieve the exact same effect as we did with the code in Listing 3-6, we can instead use the code in Listing 3-7.

LISTING 3-7 Explicitly Animating the Layer Bounds Property

```
CABasicAnimation *boundsAnimation = [CABasicAnimation
    animationWithKeyPath:@"bounds"];
[boundsAnimation setFromValue:[NSValue valueWithRect:oldRect]];
[boundsAnimation setToValue:[NSValue valueWithRect:newRect]];
[boundsAnimation setDuration:5.0f];

[layer setBounds:NSRectToCGRect(newRect)];

[layer addAnimation:boundsAnimation forKey:@"bounds"];
```

NOTE

Notes on Frame Animation

You might have noticed in the layer example code, we are calling `-setBounds` rather than `-setFrame`. It is common to want to move a layer around its containing view, which causes many first-time Core Animation programmers to attempt to use `frame` as the keypath for layer resizing. As you will quickly learn, however, animating the frame itself won't work. The `frame` field of the layer is a derived value—calculated from the `position`, `bounds`, `anchorPoint`, and `transform` properties. This means that although you can set the frame explicitly, it will not animate. This is not a problem though. You just need to determine whether you want to *move* the frame or *resize* the frame. If you want to animate the size of the layer's rectangle, use `bounds` as your keypath. If you want to move the frame, use `position` as your keypath. If you want to move *and* resize the layer at the same time, create two animations, one to animate the bounds and one to animate the position.

In Listing 3-6, we used the `CABasicAnimation` class, the primary animation object for basic animation. We take a deeper look at it shortly, but first we are going to set up a simple Xcode project to demonstrate basic layer animation.

Preparing a View to Perform Layer Animation

The first thing you want to do when you create a Core Animation based project is to make sure the root layer of your view is layer backed. Let's walk through creating a Core Animation-based project and set up the root layer on OS X.

Create the Xcode Project

To create our application, follow these steps:

1. In Xcode, press Shift-⌘-N and select Cocoa Application in the Project Templates dialog.
2. Name the project **CA Basics** and click Save.
3. Expand the Frameworks group, Control-click the Linked Frameworks subgroup, and select **Add > Existing Frameworks**.
4. In the resulting dialog, navigate to `/System/Library/Frameworks` and select `QuartzCore.framework`. Click Add twice, as prompted.
5. Control-click the Classes group and select **Add > New File**.
6. In the New File template dialog, select **Objective-C class** under the Cocoa group and click Next.
7. Name the file `AppDelegate.m` and make sure Also Create “AppDelegate.h” is checked; click Finish.
8. Select `AppDelegate.h` to open the file in the code editor and add the following code:

```
@interface AppDelegate : NSObject {
    IBOutlet UIWindow *window;
}
```

9. Select `AppDelegate.m` to open the file in the code editor and add the following code:

```
@implementation AppDelegate
- (void)awakeFromNib;
{
    [[window contentView] setWantsLayer:YES];
}
@end
```

10. Under the Resources group in your project, double-click `MainMenu.xib` to open the XIB in Interface Builder.
11. From the Library palette in Interface Builder, drag an NSObject object into `MainMenu.xib` and rename it to AppDelegate.
12. Make sure the AppDelegate object is selected. In the object inspector, click the Identity tab and change the Class field to AppDelegate.
13. In the `MainMenu.xib`, Control-click on File’s Owner and drag the connection to the AppDelegate object. Select delegate in ensuing context menu.

14. In the *MainMenu.xib*, Control-click on AppDelegate and drag the connection to the Window object. Select window in the ensuing context menu.
15. Save the xib file and return to Xcode.

The project is now set up. In the preceding steps, we created an application delegate that we use to provide control to our layer, window, and view.

Add the Animation Layer to the Root Layer

To add a layer that we will be animating, do the following:

1. Open *AppDelegate.h* and add a CALayer instance variable:

```
@interface AppDelegate : NSObject
{
    IBOutlet NSWindow *window;
    CALayer *layer;
}
```

2. Open *AppDelegate.m* and add the layer initialization code in `-awakeFromNib`:

```
@implementation AppDelegate
- (void)awakeFromNib
{
    [[window contentView] setWantsLayer:YES];

    layer = [CALayer layer];
    [layer setBounds:CGRectMake(0.0, 0.0, 100.0, 100.0)];

    // Center the animation layer
    [layer setPosition:CGPointMake([[window contentView]
                                   frame].size.width/2,
                                   [[window contentView]
                                   frame].size.height/2)];

    CGColorRef color = CGColorCreateGenericRGB(0.4, 0.3, 0.2, 1);
    [layer setBackgroundColor:color];
    CFRelease(color);

    [layer setOpacity:0.75];
    [layer setBorderWidth:5.0f];

    [[[window contentView] layer] addSublayer:layer];
}
@end
```

Note About Centering the Layer

We could call `-setFrame` on the layer before adding it to the root layer of our window's `contentView` layer tree. However, we have decided instead to set the bounds of the layer first and then set the position. (Remember frame is a derived value based on position, bounds, `anchorPoint`, and `transform`.) Setting the bounds and position properties like this makes it simpler to center the layer in the containing view. We simply obtain the parent view's width and divide it in half, and then we take the parent view's height and divide it in half. We then call `-setPosition` on the layer, which perfectly centers our layer in the `contentView`. This works because the layer's `anchorPoint` defaults to `0.5,0.5`—the center of the containing view. If we were to change the `anchorPoint` to `0.0, 0.0` the bottom left of the layer would then display at the center of the `contentView`. Figure 3-1 shows the values for the different anchor points you can use on your layer.

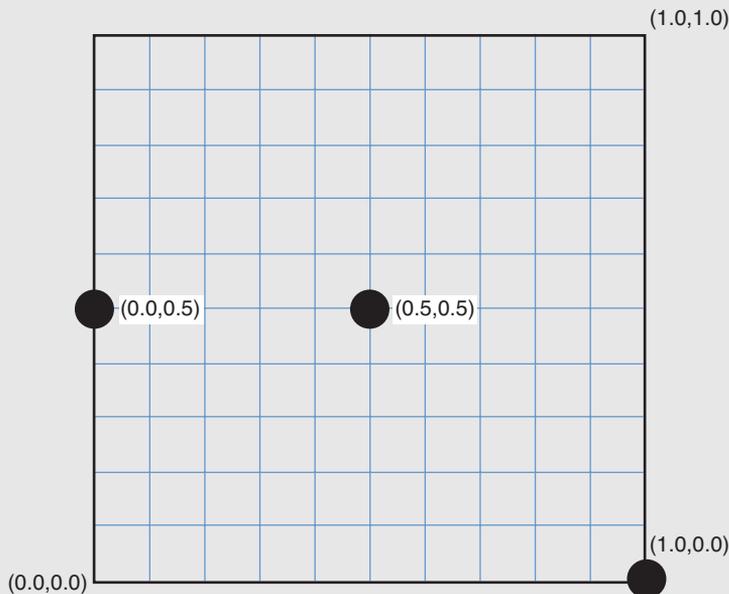


FIGURE 3-1 Layer Anchor Points

Layer Allocation Considerations

Another consideration of which you should be aware when you set up your layers is that even though you have an instance variable (ivar) of your `CALayer`, it is not retained unless you explicitly retain it. In the world of memory management in Objective-C, the rule of thumb is that you retain only that which you need to retain. You should not retain objects you don't need to hold onto, and you should retain objects that you do need.

It sounds simple, but in practice it tends to be more elusive. In our code in the previous steps, you see we allocate our layer by using the convenience initializer `layer = [CALayer layer];`. This allocates an auto-released `CALayer` object. When the layer object goes out of scope in the `-awakeFromNib`, it will be auto-released unless it is retained. In our case, we are adding it to the `contentView.layer.sublayers` array, which is retaining the layer for us. If, however, we wanted to wait until later to actually add the layer that we initialized in `-awakeFromNib` to the `sublayers` array, we need to allocate the layer by using `layer = [[CALayer alloc] init];`. Then we need to release the layer in the `dealloc` method with a call to `[layer release];`

The first time you go to use the `CALayer` method called `-removeFromSuperlayer`, you will find that if you try to add the layer to the `sublayer` array again, it will crash your application. This is because the layer will be released in the call to `-removeFromSuperlayer`. You must retain the layer yourself if you want to remove it from its superlayer but keep it around in memory.

Using CABasicAnimation

At this point, you have already seen the `CABasicAnimation` object in action. In this section, however, we consider in detail how to take advantage of this class and basic animations.

Basic animation as implemented using the `CABasicAnimation` class animates a layer property between two values, a starting and an ending value. To move a layer from one point to another in its containing window, for example, we can create a basic animation using the keypath `position`. We give the animation a start value and an ending value and add the animation to the layer. The animation begins immediately in the next run loop.

Listing 3-8 demonstrates how to animate the position of a layer.

LISTING 3-8 Animate the Layer Position

```
- (IBAction)animate:(id)sender;
{
    CABasicAnimation *animation =
        [CABasicAnimation animationWithKeyPath:@"position"];
    [animation setFromValue:[NSValue valueWithPoint:startPoint]];
    [animation setToValue:[NSValue valueWithPoint:endPoint]];
    [animation setDuration:5.0];

    [layer addAnimation:animation forKey:@"position"];
}
```

This code moves the position of a layer from `startPoint` to `endPoint`. These two values are `NSPoint` objects. The `position` property is the center point of the layer. It is set relative to its containing layer.

If you add this listing to your project we created in the previous section, you simply connect a button to the action in Interface Builder. To do so, follow these steps:

1. Open *AppDelegate.h* and add an action declaration, as follows:

```
@interface AppDelegate : NSObject
{
    IBOutlet UIWindow *window;
    CALayer *layer;
}
- (IBAction)animate:(id)sender;
```

2. Open *AppDelegate.m* and add the animate implementation code provided in Listing 3-8.
3. Open Interface Builder. From the Objects Library, drag a button onto the main window.
4. Control-click the button you just dragged on the main window and drag a connection to the AppDelegate object. Select the animate action.
5. Return to Xcode and Build and Go to see this animation run.

That's it. That is really all there is to animating a layer. You create the animation, set the to and from values, set a duration (which is optional as the default 0.25 seconds will be used if you don't specify a duration explicitly), and add the animation to the layer you want to animate.

That being said, you will not likely leave it at that because the details of implementation add nuance and complexity. For example, the first time you run the animation from Listing 3-8, you notice that while your layer animates to the correct position in the parent view using the duration you specified, when the animation completes, it jumps right back to its starting position. Is this a bug? How can we fix it? We get to that next.

Animating Versus Setting Layer Properties

When you create your CABasicAnimation, you need to specify a start and stop value for the animation using the calls to `-setFromValue` and `-setToValue` respectively. When you add your basic animation to a layer, it runs. However, when the property animation finishes, in the case of animating the position property, the layer snaps right back to its starting position.

Remember that when animating, you use at least two objects. These objects are the layer itself, a CALayer or CALayer-derived object, and the animation that you assign to it—the CABasicAnimation object in our previous examples. Just because you have set a final value (destination) for your animation object does not mean that the layer property being animated assumes this value when the animation has finished. You must explicitly set the

layer's property so that when the animation has finished, the property you animated will actually be set in the layer to the to-value you specified.

You can simply cause your animation to stop at the end point you specify, but this is only a visible stickiness, if you will. The internal value is still the same. To actually change the internal model value, you have to explicitly set the property in question. For example, to explicitly set the position property, you need to call `-setPosition` on the layer. This creates a little problem, though.

If you set the value of a property by calling `-set` on that property explicitly, the default animation will be used rather than one you might set for the property you are animating. Listing 3-9 demonstrates one way you might try to set the position. Notice that we have created a basic animation to use for the position property; however, the explicit call to `-setPosition` on the layer overrides the animation we set in the line that follows it, making the basic animation completely useless. If you try this code, you see that although our layer ends up in the right position, it uses the default duration of 0.25 seconds rather than the 5 seconds we have explicitly set in the animation.

LISTING 3-9 Animating and Updating the Position Property

```
CABasicAnimation *animation =
    [CABasicAnimation animationWithKeyPath:@"position"];
[animation setFromValue:[NSValue valueWithPoint:startPoint]];
[animation setToValue:[NSValue valueWithPoint:endPoint]];
[animation setDuration:5.0];

[layer setPosition:endpoint];

[layer addAnimation:animation forKey:nil];
```

So now the question becomes, how can you get the animation to use the specified duration? Take a look at the last line in Listing 3-9. Notice that the `forKey:` parameter of the call is set to `nil`. This is the reason why the animation is not overriding the default. If you change the last line to `[layer addAnimation:animation forKey:@"position"]`, the animation will work using the duration as expected. This tells the layer to use the new animation we have specified for this keypath whenever it needs to be animated.

Implicit Layer Animation and the Default Timing Function

We can use the `CATransaction` class to override the default duration as we previously did in this chapter, and it does make it simple to animate the layer using the duration we specify. If we use the code in Listing 3-10, the `position` property is set in the layer and the property is animated on its way there as you might expect.

Listing 3-10 Overriding the Default Duration for Implicit Animation

```
[CATransaction begin];
[CATransaction setValue:[NSNumber numberWithInt:5.0]
    forKey:kCATransactionAnimationDuration];

[layer setPosition:endPoint];
[CATransaction commit];
```

However, when you run this code, you see that although it animates the position over a five second duration, it also applies the default media timing function that is `kCAMediaTimingFunctionEaseInEaseOut`. This function causes the animation to start slowly and then speed up only to slow down again as it approaches its destination. This functionality is fine if that is the media timing function you want, but if you want it to be linear (`kCAMediaTimingFunctionLinear`), for example, you need to consider another way. And there is no apparent way to set the default media timing function for implicit animations.

This means that if you want to use any other timing function than the default, you have to use explicit animation, as shown in Listing 3-9.

Visual Stickiness

Another approach we might take is to set several properties in our animation object that cause the animation to be sticky when it finishes. In other words, the layer will appear to be at the destination value. The stickiness in this scenario, however, is visual only, which is to say that the underlying value of the layer property, `position` continues to be the value the position was when the animation started. This is a fine approach if you don't need the internal value to be updated. Listing 3-11 shows how to implement this method, making the layer stick at the end of its duration.

LISTING 3-11 Making the Layer Position Sticky

```
CABasicAnimation *animation = [CABasicAnimation
    animationWithKeyPath:@"position"];
[animation setValue:[NSValue valueWithPoint:endPoint]];
[animation setDuration:5.0];
[animation setFillMode:kCAFillModeForwards];
[animation setRemovedOnCompletion:NO];

[layer addAnimation:animation forKey:@"position"];
```

We need to set two animation properties for the layer to stay at the destination position. First is the fill mode. We tell it to anchor the animation value to the final value by calling `-setFillMode`, passing it the constant `kCAFillModeForwards`. Then we must tell the animation not to remove itself from the layer's array of animations when the animation completes by calling `-setRemovedOnCompletion` passing it `NO`.

NOTE

This code effectively causes the animation to finish at the destination, but remember it is only a visible effect. The internal position value of the layer is still at the start value. This becomes problematic if you need to obtain the current value after your animation has run.

Useful Animation Properties

You have already discovered all the properties that you can animate in a layer. However, there are numerous properties available in the animation (`CABasicAnimation`) object that can give you greater control and enhance your animations.

- ▶ **Autoreverses**
By setting `autoreverses` to `YES`, the animated property returns to its starting value after it has reached its destination value, but instead of snapping back to the start value, it animates there.
- ▶ **Duration**
Duration is a parameter you are quite familiar with at this point. It sets the amount of time to be taken between the `fromValue` and `toValue` of the animation. Duration is also affected by the `speed` property.
- ▶ **RemovedOnCompletion**
The default value for this property is `YES`, which means that when the animation has finished its specified duration, the animation is automatically removed from the layer. This might not be desirable. If you want to animate the property you've specified again, for example, you want to set this property to `NO`. That way, the next time you call `-set` on the property being animated in the animation, it will use your animation again rather than the default.
- ▶ **Speed**
The default value for this property is `1.0`. This means that the animation plays back at its default speed. If you change the value to `2.0`, the animation plays back at twice the default speed. This in effect splits the duration in half. If you specify a duration of 6 seconds and a speed of `2.0`, the animation actually plays back in three seconds—half the duration specified.

- ▶ **BeginTime**
This property is useful in an animation group. It specifies a time for the animation to begin playing in relation to the time of the parent group animation's duration. The default is `0.0`. Animation grouping is discussed in the next section, "Animation Grouping."
- ▶ **TimeOffset**
If a time offset is set, the animation won't actually become visible until this amount of time has elapsed in relation to the time of the parent group animation's duration.
- ▶ **RepeatCount**
The default is zero, which means that the animation will only play back once. To specify an infinite repeat count, use `1e100f`. This property should not be used with `repeatDuration`.
- ▶ **RepeatDuration**
This property specifies how long the animation should repeat. The animation repeats until this amount of time has elapsed. It should not be used with `repeatCount`.



Animation Grouping

In the previous section, "Useful Animation Properties," we defined two particular properties that are only pertinent to animation grouping: `beginTime` and `timeOffset`. Before discussing those, however, let's consider why you might want to use an animation group rather than just adding a list of animations to the layer.

In Listing 3-12, you can see that we build up a list of basic animations and simply add them to the layer. If you want all your animations to begin at the same time and each of them have the same duration, this method is perfectly adequate.

LISTING 3-12 Adding a List of Animations to the Layer

```
- (IBAction)animate:(id)sender;
{
    NSRect oldRect = NSMakeRect(0.0, 0.0, 100.0, 100.0);
    NSRect newRect = NSMakeRect(0.0, 0.0, 300.0, 300.0);

    CABasicAnimation *boundsAnimation =
    [CABasicAnimation animationWithKeyPath:@"bounds"];
    [boundsAnimation setValue:[NSValue valueWithRect:oldRect]];
    [boundsAnimation setValue:[NSValue valueWithRect:newRect]];
    [boundsAnimation setDuration:5.0f];

    CABasicAnimation *positionAnimation =
    [CABasicAnimation animationWithKeyPath:@"position"];
```

LISTING 3-12 Continued

```

[positionAnimation setFromValue:
 [NSValue valueWithPoint:
 NSPointFromCGPoint([layer position])]];
[positionAnimation setToValue:
 [NSValue valueWithPoint:NSMakePoint(0.0, 0.0)]];
[positionAnimation setDuration:5.0f];

CABasicAnimation *borderWidthAnimation =
 [CABasicAnimation animationWithKeyPath:@"borderWidth"];
[borderWidthAnimation setFromValue:[NSNumber numberWithFloat:5.0f]];
[borderWidthAnimation setToValue:[NSNumber numberWithFloat:30.0f]];
[borderWidthAnimation setDuration:5.0f];

[layer addAnimation:boundsAnimation forKey:@"bounds"];
[layer addAnimation:positionAnimation forKey:@"position"];
[layer addAnimation:borderWidthAnimation forKey:@"borderWidth"];
}

```

Each animation has a duration of 5 seconds, and they begin to play back simultaneously in the next run loop and they end at the same time. The position of the layer moves to the bottom left corner, the border width grows to 30 pixels, and size of the layer grows from 100×100 pixels to 300×300 pixels.

Let's say that we would prefer that, rather than having all our animations play simultaneously, we want them to play back sequentially—one following the previous. We can achieve this by using a group animation and setting the `beginTime` field. I should mention now that in this case it might make more sense to use a keyframe animation instead, but you need to read Chapter 4, "Keyframe Animation," to see how that works.

We must explicitly specify the duration of our animation group so that the time for each individual animation can be split up accordingly. In our example, we set our duration of the animation group to last 15 seconds and get each of our individual animations to play back for 5 seconds. Listing 3-13 shows how we can take a previous example and instead use animation grouping for greater control over animation playback.

LISTING 3-13 Using Animation Grouping

```

- (IBAction)animate:(id)sender;
{
    NSRect oldRect = NSMakeRect(0.0, 0.0, 100.0, 100.0);
    NSRect newRect = NSMakeRect(0.0, 0.0, 300.0, 300.0);

    CABasicAnimation *boundsAnimation =
    [CABasicAnimation animationWithKeyPath:@"bounds"];

```

LISTING 3-13 Continued

```

[boundsAnimation setFromValue:[NSValue valueWithRect:oldRect]];
[boundsAnimation setToValue:[NSValue valueWithRect:newRect]];
[boundsAnimation setDuration:15.0f];
[boundsAnimation setBeginTime:0.0f];

CABasicAnimation *positionAnimation =
[CABasicAnimation animationWithKeyPath:@"position"];
[positionAnimation setFromValue:
[NSValue valueWithPoint:
NSPointFromCGPoint([layer position])]];
[positionAnimation setToValue:
[NSValue valueWithPoint:NSMakePoint(0.0, 0.0)]];
[positionAnimation setDuration:15.0f];
[positionAnimation setBeginTime:5.0f];

CABasicAnimation *borderWidthAnimation =
[CABasicAnimation animationWithKeyPath:@"borderWidth"];
[borderWidthAnimation setFromValue:[NSNumber numberWithFloat:5.0f]];
[borderWidthAnimation setToValue:[NSNumber numberWithFloat:30.0f]];
[borderWidthAnimation setDuration:15.0f];
[borderWidthAnimation setBeginTime:10.0f];

CAAnimationGroup *group = [CAAnimationGroup animation];
[group setDuration:15];
[group setAnimations:
[NSArray arrayWithObjects:boundsAnimation,
positionAnimation,
borderWidthAnimation, nil]];

[layer addAnimation:group forKey:nil];
}

```

Notice that we have set the duration for each of the individual animations to the full fifteen seconds, but each of the animations have their begin times set to start one after the other at 0.0, 5.0, and 10.0.

You also notice that the only thing we add to the layer is the group animation. The animation objects in the group have been added with a call to `-setAnimations`.

You can see that there is a good bit of flexibility provided through grouping. You just need to tweak your durations and begin times to suit your needs. If you want the animations to overlap, you just change the begin times to reflect when you want them to start playing back. You want to keep your duration times all the same; otherwise each keypath value (that is, bounds, position, and borderWidth) in the layer snaps back to its original

value when its duration has completed, which gives predictable, yet seemingly sporadic, behavior. Keeping all the durations the same makes them wait the full duration before they snap back. If you don't want them to snap back, you need to explicitly set their values when the animation finishes, which we previously discussed in the section, "Using CABasicAnimation."

Summary

Basic animation is powerful. You have a lot of flexibility in the way you can achieve your application animation goals. Often you won't need to go beyond what is provided for you in basic animation. Keep it simple. If all you need is the animator proxy, use it. If all you need is to set a layer property, call that property's set method and let Core Animation handle the rest. If you need to have more flexibility over the animation parameters, use a CABasicAnimation object and set all the animation properties yourself. Just remember that probably more often than not, you only need basic animation.

Index

A

-acceptsFirstResponder method, 189

-actionForLayer: method, 31

-action: method, 225-226

-addAnimation: method, 9

adding

blur animation to mouseDown events, 89

close box, 61-62

filters to Core animation layers, 84

Gaussian Blur Filter to layers, 84

-addSublayer method, 162

allocating layers, 35-36

alpha blending, 196-199

anchor points, 74-75

animation

adding animation to layers, 9

adding list of animations to layer, 41-42

animator proxy object, 28

applied filters, 85-89

attribute animation, 18

basic animation to keyframe animation, 50

CABasicAnimation object, 40-41

Cartesian coordinate system, 3-4

color animation, 16-17

content animation, 19-20

design principles, 22

- disabling, 31
 - filter animation, 19
 - grouping, 41-44
 - on iPhone. See iPhone, Core Animation on
 - layer animation
 - animating layer positions, 36-37
 - animating versus setting layer properties, 37-38
 - implicit layer animation, 38-39
 - visual stickiness, 39-40
 - layers, resizing, 31-32
 - masking animation, 20-21
 - monitoring progress of, 52
 - motion animation, 17-18
 - multithreaded animation, 202
 - with filters, 204
 - locking layer changes, 203
 - threads and run loops, 204
 - overlays, 118-119
 - pacing, 9-10
 - projects
 - application delegate class, 12, 33
 - creating Xcode projects, 33-34
 - setting up Xcode projects for OS X, 12-13
 - removing from layers, 9
 - simple animations, 27-28
 - Star Label Animation. See Star Label Animation application
 - time codes, 119-121
 - at UIView level
 - building application, 223
 - MainViewController, 223-226
 - views, resizing, 30
 - visibility field animation, 18
 - what not to animate, 23-24
 - when to use Core Animation, 23
 - windows, resizing, 28-29
 - animationDidStop: method, 214**
 - animator proxy object, 28**
 - AppDelegate, 12, 33**
 - awakeFromNib method, 183-184
 - buttonLayerHit method, 184
 - for filter slider, 96-98
 - for ripple transition
 - AppDelegate implementation with delegation, 106-109
 - AppDelegate implementation with encapsulation, 105-106
 - applicationDidFinishLaunching: method, 70, 76-77, 173, 213, 217**
 - applied filters, animating, 85-89**
 - atomic operations, 203**
 - attributes, animating, 18**
 - autoplay method, 112**
 - Autoreverses property (CABasicAnimation class), 40**
 - awakeFromNib method, 34, 113, 118-120, 126, 142-143, 153-154, 183-184, 200**
- ## B
- background color, animating from red to green, 16-17**
 - background threads, 5**
 - backgroundColorAnimation method, 17**
 - basic animation, 50**
 - beginTime property, 41-42**
 - binding objects in Star Label Animation application, 211-214**

- animationDidStop: method, 214
- applicationDidFinishLaunching: method, 213
- drawStar: method, 213
- StarStepsAppDelegate interface, 212

blur animation, adding to mouseDown events, 89

BlurView objects, initializing, 88

-buttonLayerHit method, 184-185

C

CAAnimation object, 9

CABasicAnimation class

- animating layer positions, 36-37
- animating versus setting layer properties, 37-38
- animation grouping, 41-44
- implicit layer animation and default timing function, 38-39
- properties, 40-41
- visual stickiness, 39-40

CAEAGLLayer class, 7

CAGradientLayer class, 8, 168-171

CALayer class, 5-6

- hit testing CALayer objects, 177-179

-canDrawInCGLContext method, 132-135, 159-160

CAOpenGLLayer class, 5

- overview, 5-7, 131
- and Quartz Composition layer, 159-160
- rendering multiple video channels, 138-139
 - OpenGLVidGridLayer, 139-148
 - VideoChannel, 139

- VideoChannelController, 139

rendering video in, 131-132

- advantages, 138

- canDrawInCGLContext method, 132-135, 159-160

- display link callback signature, 133

- display links, 133

- drawInCGLContext: method, 132, 136-137

- implementation of QuickTime visual context, 135-136

- layer timing, 133-138

capture sessions, initializing, 124-125

-captureOutput: method, 127

capturing current image, 127-129

CAReplicatorLayer class, 8

- adding ReplicatorView to window, 173-174
- creating ReplicatorView, 171-173
- properties, 171

Cartesian coordinate system, 3-4

CAScrollLayer class, 7

CAShapeLayer class, 8

- adding to layer tree, 161-163
- as layer mask, 165-167
- initializing, 161-163
- path stroke manipulations, 163-165

CATextLayer class, 7

CATiledLayer class, 7, 199-202

CATransaction class, 31-32

CATransform3DMakeRotation method, 73-74

CATransform3DMakeScale method, 72

centering layers, 35

central processing unit (CPU), 193

CGImageRed, obtaining image data as, 157

`-channelsReadyToDraw`: function, 144

circular locks, 203-204

classes

`AppDelegate`, 12, 33

`-awakeFromNib` method, 183-184

`-buttonLayerHit` method, 184

 for filter slider, 96-98

 for ripple transition, 105-109

`CABasicAnimation`

 animating layer positions, 36-37

 animating versus setting layer properties, 37-38

 animation grouping, 41-44

 implicit layer animation and default timing function, 38-39

 properties, 40-41

 visual stickiness, 39-40

`CAEAGLLayer`, 7

`CAGradientLayer`, 8, 168-171

`CALayer`, 5-6

 hit testing `CALayer` objects, 177-179

`CAOpenGLLayer`

 overview, 5-7, 131

 rendering multiple video channels, 138-148

 rendering video in, 131-138

 layer timing, 133-138

`CAReplicatorLayer`, 8

 adding `ReplicatorView` to window, 173-174

 creating `ReplicatorView`, 171-173

 properties, 171

`CAScrollLayer`, 7

`CAShapeLayer`, 8

 adding to layer tree, 161-163

 initializing, 161-163

 as layer mask, 165-167

 path stroke manipulations, 163-165

`CATextLayer`, 7

`CATiledLayer`, 7, 199-202

`CATransaction`, 31-32

`LZButton`, 181-182

`LZButtonLayer`, 179-182

`LZContentView`, 178

`MainViewController`, 223-226

`-action`: implementation method, 225-226

 header, 224

`OpenGLVidGridLayer`, 139-148

`-channelsReadyToDraw`: function, 144

`-drawAllInRect`: function, 146

`-drawChannel`: function, 147-148

 implementation of drawing functions, 143-144

 initialization code, 140-142

 initializing in `AppDelegate`, 142-143

`-readyToDrawNextFrame`: function, 145

`QCCompositionLayer`, 7, 153-154

`QCQuartzCompositionLayer`, 159-160

`QTCaptureLayer`, 7, 123

 creating and displaying, 125-127

 current image, capturing, 127-129

 initializing capture session, 124-125

`QTMovieLayer`

 action methods, 112

`contentsRect`, 122-123

- overlays, 118-119
- overview, 7, 111-112
- QTMovieLayer-based player, 112-118
- time codes, 119-121
- ReplicatorView
 - adding to window, 173-174
 - creating, 171-173
- TouchableView, 218-223
 - header, 218
 - initWithFrame: method, 219
 - throbAnimationDidStop:finished:context: method, 221
 - touchBegan:withEvent: method, 219
 - touchesCancelled:withEvent: method, 220
 - touchesEnded:withEvent: method, 220-221
 - touchesMoved:withEvent: method, 222
- TouchMeViewController, 216-218
 - Application Delegate's header, 217
 - Application Delegate's method, 217
 - TouchMeViewController.m method, 216
- VideoChannel, 139
- VideoChannelController, 139
- Cocoa Touch, Core Animation on**
 - animations at UIView level
 - building application, 223
 - MainViewController, 223-226
 - benefits, 208
 - limitations, 208-209
 - overview, 207
 - Star Label Animation application
 - QuartzCore framework and object binding, 211-214
 - setting up Xcode project, 209
 - UIColor, 210-211
 - Xcode project setup, 209
- touch system application
 - overview, 214-215
 - TouchableView, 218-223
 - TouchMeViewController, 216-218
 - Xcode project setup, 215-216
- transforms in, 69
- color animation**
 - animating background color from red to green, 16-17
 - Color Changer sample application
 - awakeFromNib method, 183-184
 - buttonLayerHit method, 184-185
 - hitTest: method, 185
 - Interface Builder, 182
 - LZButtonLayer, 179-182
 - mouseDown: method, 185-187
 - mouseEntered: method, 187
 - mouseExited: method, 187
 - mouseUp: method, 186-188
- combining transforms, 76-79**
- compositions (Quartz Composer)**
 - adding filters to, 156
 - creating, 150-152
 - getting image data from, 155-156
 - input parameters, 152
 - passing parameters to, 154-155
- constants, kCAFillModeForwards, 40**
- containers, NSAnimatablePropertyContainer, 28**
- content, animating, 19-20**
- contents property, 19**
- contentsRect property, 19, 122-123**

control buttons, adding to Movie Player with Overlay project, 113-115

controllers

MainViewController, 223-226

–action: implementation, 225-226

header, 224

TouchMeViewController, 216-218

Application Delegate’s header, 217

Application Delegate’s implementation, 217

TouchMeViewController.m implementation, 216

VideoChannelController, 139

controlling filter values with data bindings, 92-98

coordinate system, 3-4

Core Animation layers, adding filters to, 84

Core Graphics path, specifying, 46

Core Image filters, 83

CPU (central processing unit), 193

current image, capturing, 127-129

custom transitions, 101-108

delegating, 101-102

encapsulating, 101-102

Ripple transition, 102-106

AppDelegate implementation with delegation, 106-109

AppDelegate implementation with encapsulation, 105-106

RippleLayer interface, 104-105

D

Dashboard effect, 10

data bindings, controlling filter values, 92-98

deepest layer, 185

default timing function, 38-39

default transitions, 98-100

–defaultActionForKey: method, 102

delegating custom transitions, 101-102, 106-109

design elements, 23

design principles, 22

disabling animation, 31

display links

callback signature, 133

definition of, 133

–doIn: method, 11

–doTransition method, 109

–drawAllInRect: function, 146

–drawChannel: function, 147-148

–drawInCGLContext method, 159-160

–drawInCGLContext: method, 132, 136-137

–drawInContext: method, 195

–drawLayer: method, 201

–drawStar: method, 213

dreadlocks, 203-204

Duration property (CABasicAnimation class), 40

E

Edit Mode (iPhone/iPod Touch), 56

encapsulating custom transitions, 101-102, 105-106

events

- keyboard events, 188-190
- mouse events
 - Color Changer sample application, 179-188
 - hit testing CALayer objects, 177-179
 - overview, 177

F**Fade transitions, 99****filters**

- adding to Core animation layers, 84
- adding to Quartz Compositions, 156
- animating, 19
- applied filters, 85-89
- controlling values with data binding, 92-98
- key paths, 95
- multithreading with, 204
- performance issues, 195
- “sticky” effect, 91-92
- transitions
 - custom transitions, 101-108
 - default transitions, 98-100
 - user input, receiving, 89-90
- filters method, 94-95**
- frameOrigin property, 30**
- frameSize property, 30**
- From Bottom transitions, 99**
- From Left transitions, 99**
- From Right transitions, 99**

From Top transitions, 99

functions. See methods

G

- Gaussian Blur Filter, 84**
- getCurrentImage method, 127-128**
- getDotScreenImage: method, 157**
- gotoBeginning method, 112**
- gotoEnd method, 112**
- gotoNextSelectionPoint method, 112**
- gotoPosterFrame method, 112**
- gotoPreviousSelectionPoint method, 112**
- grabImage: method, 128**
- gradient layer. See CAGradientLayer class**
- graphic artists, 22**
- graphics processing unit (GPU), 194**
- green background color, animating, 16-17**
- group animation, 41-44**

H

- hardware acceleration, 193-194**
- HIG (Human Interface Guidelines), 22**
- hit testing CALayer objects, 177-179**
- hitTest: method, 179, 185**
- Human Interface Guidelines (HIG), 22**

I

-(IBAction)action:(id)sender method, 77-79

Icon Dance application, 58, 65

icon shaking, implementing with keyframe animation, 57-61

 close box, adding, 61-62

 rotation axis and layer geometry, 61

 starting/stopping, 64-65

images

 capturing current image, 127-129

 as layer content, 63

 reflection, applying with CAGradientLayer class, 168-171

implicit layer animation, 38-39

-init method, 181

-initCaptureSession method, 124-125

initializing CAShapeLayer, 161-163

-initWithChannels method, 140-142

-initWithFrame: method, 172, 219

-initWithFrame:reuseIdentifier: method, 197-199

interaction

 keyboard events, 188-190

 layer-backed views, 190

 mouse events

 Color Changer sample application, 179-188

 hit testing CALayer objects, 177-179

 overview, 177

Interface Builder for Color Changer sample application, 182

interfaces

 RippleLayer, 104-105

 StarStepsAppDelegate, 212

interpolation, 46

iPhone, Core Animation on, 11

 animations at UIView level

 building application, 223

 MainViewController, 223-226

 benefits, 208

 Edit Mode, 56

 Icon Dance application, 65

 keyframe steps animation, 56

 limitations, 208-209

 nondeletable applications, 57

 overview, 207

 Star Label Animation application

 QuartzCore framework and object binding, 211-214

 setting up Xcode project, 209

 UIWindow, 210-211

 Xcode project setup, 209

 touch system application

 overview, 214-215

 TouchableView, 218-223

 TouchMeViewController, 216-218

 Xcode project setup, 215-216

iPod Touch

 Edit Mode, 56

 nondeletable applications, 57

ivars, 86

J-K

kCAFillModeForwards constant, 40

kCAMediaTimingFunctionEaseInEaseOut method, 39

kCATransitionFade constant, 99

kCATransitionFromBottom constant, 99
kCATransitionFromLeft constant, 99
kCATransitionFromRight constant, 99
kCATransitionFromTop constant, 99
kCATransitionMoveIn constant, 99
kCATransitionPush constant, 99
kCATransitionReveal constant, 99
key paths (filter), 95
key times, 51
key-value coding (KVC), 86
keyboard events, 188-190
keyframe animation, 45-50

- from basic animation, 50
- implementing icon shake, 57-65
- interpolation, 46
- monitoring keyframe destinations, 52-56
- steps for iPhone, 56
- timing, 50-51
- UI, 56-57

keyframe destinations

- methods to, 46-49
- monitoring, 52-56
- path animation, 47

keypaths, KVC (key-value coding), 86
keywords, @synthesize, 93
KVC (key-value coding), 86

L

layerClass method, overriding, 172

layers

- adding animation to, 9
- adding list of animations to, 41-42

- adding to root layer, 34-35
- alpha blending, 196-199
- animation
 - animating layer positions, 36-37
 - animating versus setting layer properties, 37-38
 - animation pacing, 9-10
 - implicit layer animation and default timing function, 38-39
 - visual stickiness, 39-40
- CAEAGLLayer, 7
- CAGradientLayer, 8, 168-171
- CALayer, 5-6
 - hit testing CALayer objects, 177-179
- CAOpenGLLayer
 - overview, 5-7, 131
 - rendering multiple video channels, 138-148
 - rendering video in, 131-138
 - layer timing, 133-138
- CAReplicatorLayer, 8
 - adding ReplicatorView to window, 173-174
 - creating ReplicatorView, 171-173
 - properties, 171
- CAScrollLayer, 7
- CAShapeLayer, 8
 - adding to layer tree, 161-163
 - initializing, 161-163
 - as layer mask, 165-167
 - path stroke manipulations, 163-165
- CATextLayer, 7
- CATiledLayer, 7, 199-202
- centering, 35

- content, 63
- deepest layer, 185
- definition of, 5-6
- Gaussian Blur Filter, 84
- layer allocation considerations, 35-36
- layer-backed views, 190
- locking layer changes, 203
- LZButtonLayer, 179-182
- masks, CAShapeLayer as, 165-167
- OpenGLVidGridLayer, 139-148
 - channelsReadyToDraw: function, 144
 - drawAllInRect: function, 146
 - drawChannel: function, 147-148
 - implementation of drawing functions, 143-144
 - initialization code, 140-142
 - initializing in AppDelegate, 142-143
 - readyToDrawNextFrame: function, 145
- properties, 37-38
- purpose of, 8
- QCCompositionLayer
 - adding to window, 153-154
- QCCompositionLayer class, 7
- QTCaptureLayer, 7, 123
 - creating and displaying, 125-127
 - current image, capturing, 127-129
 - initializing capture session, 124-125
- QTMovieLayer
 - action methods, 112
 - contentsRect, 122-123
 - overlays, 118-119
 - overview, 7, 111-112
 - QTMovieLayer-based player, 112-118
 - time codes, 119-121

- removing animation from, 9
- resizing, 31-32
- rotating
 - along one axis, 72-73
 - along two axes, 73-74
 - magnitude of rotation, 73
- scaling, 10-11, 70-72
- shake animation, 61
- tiled layers, 199-202
- transforms
 - anchor points, 74-75
 - in Cocoa Touch, 69
 - combining, 76-79
 - definition of, 69
 - rotate3DTransform:, 73-74
 - rotateTransform:, 72-73
 - scale versus bounds, 79-80
 - scaleTransform:, 70-72

- locking layer changes, 203
- LZButton class, 181-182**
- LZButtonLayer class, 179-182**
- LZContentView class, 178**

M

- macro patches, 157
- magnitude of rotation, 73
- MainViewController class, 223-226
- mask property, 21
- masking fields, animating, 20-21
- masks, CAShapeLayer as, 165-167
- masksToBounds property, 21

methods

- acceptsFirstResponder method, 189
- actionForLayer:, 31
- action:, 225-226
- addAnimation:, 9
- addSublayer, 162
- animationDidStop:, 214
- applicationDidFinishLaunching:, 70, 76-77, 173, 213, 217
- autoplay method, 112
- awakeFromNib, 34, 113, 118-120, 126, 142-143, 153-154, 183-184, 200
- backgroundColorAnimation, 17
- buttonLayerHit, 184-185
- canDrawInCGLContext, 132-135, 159-160
- captureOutput:, 127
- CATransform3DMakeRotation, 73-74
- CATransform3DMakeScale, 72
- channelsReadyToDraw:, 144
- defaultActionForKey:, 102
- doIn:, 11
- doTransition, 109
- drawAllInRect:, 146
- drawChannel:, 147-148
- drawInCGLContext, 132, 136-137, 159-160
- drawInContext:, 195
- drawLayer:, 201
- drawStar:, 213
- filters, 94-95
- getCurrentImage, 127-128
- getDotScreenImage:, 157
- gotoBeginning, 112
- gotoEnd, 112
- gotoNextSelectionPoint, 112
- gotoPosterFrame, 112
- gotoPreviousSelectionPoint, 112
- grabImage:, 128
- hitTest:, 179, 185
- (IBAction)action:(id)sender:, 77-79
- init, 181
- initCaptureSession, 124-125
- initVideoChannels, 140-142
- initWithFrame:, 172, 219
- initWithFrame:reuseIdentifier:, 197-199
- kCAMediaTimingFunctionEaseInEaseOut, 39
- layerClass, 172
- mouseDown:, 185-187
- mouseEntered:, 187
- mouseExited:, 187
- mouseUp:, 186-188
- performSelectorOnMainThread:, 204
- play, 112
- readyToDrawNextFrame:, 145
- removeAnimation:, 9
- rotate3DTransform:, 73-74
- rotateTransform:, 72-73
- scaleTransform:, 70-72
- setAnimations:, 43
- setBounds:, 32, 109
- setCurrentTime:, 112
- setFillMode:, 40
- setFilters, 93
- setFrame:, 28-30
- setNeedsDisplay, 204
- setPosition, 4

- setRemovedOnCompletion:, 40
- setSelected:, 182
- setString:, 181
- setStrokeColor, 163
- setupVisualContext:, 135-136
- sliderMoved:, 115
- stepBackward, 112
- stepForward, 112
- stop, 112
- string:, 181
- throbAnimationDidStop:finished:context:, 221
- togglePlayback:, 117
- touchBegan:withEvent:, 219
- touchesCancelled:withEvent:, 220
- touchesEnded:withEvent:, 220-221
- touchesMoved:withEvent:, 222
- updateSlider:, 117, 121
- updateTimeStamp, 120
- valueForOutputKey:, 157
- viewDidLoad, 162-169
- zoom:, 201

Model–View–Controller (MVC) design pattern, 5

monitoring

- animation progress, 52
- keyframe destinations, 52-56

motion fields, animating, 17-18

mouse events

- Color Changer sample application, 179
 - awakeFromNib method, 183-184
 - buttonLayerHit method, 184-185
 - hitTest: method, 185
- Interface Builder, 182

- LZButtonLayer, 179-182

- mouseDown: method, 185-187

- mouseEntered: method, 187

- mouseExited: method, 187

- mouseUp: method, 186-188

- hit testing CALayer objects, 177-179

- mouseDown events, 89

- overview, 177

- mouseDown: method, 185-187**

- mouseEntered: method, 187**

- mouseExited: method, 187**

- mouseUp: method, 186-188**

Move In transitions, 99

Movie Player with Overlay (QTMovieLayer–based player), 112-118

- control buttons, 113-115

- simple movie playback, 113

- slider, 115

- timer, 117-118

movies

- adding overlays to, 118-119

- adding time codes to, 119-121

- Movie Player with Overlay (QTMovieLayer–based player), 112-118

- control buttons, 113-115

- simple movie playback, 113

- slider, 115

- timer, 117-118

multivideo streams, creating with Quartz Composer

- compositions

- adding filters to, 156

- creating, 150-152

- getting image data from, 155-156

- input parameters, 152
- passing parameters to, 154-155
- obtaining current image in code, 157
- overview, 149-150
- QCCompositionLayer, 153-154
- Xcode projects, creating, 152-153

multiple video channels, rendering, 138-139

- OpenGLVidGridLayer, 139-148
 - channelsReadyToDraw: function, 144
 - drawAllInRect: function, 146
 - drawChannel: function, 147-148
- implementation of drawing functions, 143-144
- initialization code, 140-142
- initializing in AppDelegate, 142-143
- readyToDrawNextFrame: function, 145

VideoChannel, 139

VideoChannelController, 139

multithreaded animation, 202

- locking layer changes, 203
- threads and run loops, 204
- with filters, 204

multithreading, 5

MVC (Model-View-Controller) design pattern, 5

N

nested transforms, 195

nondeletable applications, 57

NSAnimatablePropertyContainer, 28

NSAnimationContext object, 29-30

O

objects

- animator proxy object, 28
- binding in Star Label Animation application, 211, 213-214
- CAAnimation, 9
- NSAnimationContext, 29-30

offscreen rendering, 194

opacity property, 100

OpenGL layer. See CAOpenGLLayer

OpenGLVidGridLayer, 139-148

- channelsReadyToDraw: function, 144
- drawAllInRect: function, 146
- drawChannel: function, 147-148
- readyToDrawNextFrame: function, 145
- implementation of drawing functions, 143-144
- initialization code, 140-142
- initializing in AppDelegate, 142-143

OS X, setting up Xcode projects for, 12-13

overlays, adding to movies, 118-119

overriding layerClass method, 172

P

spacing, 9-10

parameters, passing to Quartz Compositions, 154-155

path animation, 47

path stroke, manipulating with CAShapeLayer, 163-165

-performSelectorOnMainThread: method, 204

performance

- alpha blending, 196-199
- filters and shadows, 195
- hardware acceleration, 193-194
- multithreaded animation, 202
 - with filters, 204
 - locking layer changes, 203
 - threads and run loops, 204
- nested transforms, 195
- offscreen rendering, 194
- overview, 193
- tiled layers, 199-202
- transition effects, 195

Photo Capture project

- capture session, initializing, 124-125
- current image, capturing, 127-129
- QTCaptureLayer, creating and displaying, 125-127

-play method, 112**positions of layers, animating, 36-37****positionUse property, 18****projects. See also specific projects**

- application delegate class, 12, 33
- setting up Xcode projects for OS X, 12-13
- Xcode projects, creating, 33-34, 152-153

properties. See specific properties**Push transitions, 99****Q****QCCompositionLayer class, 7, 153-154****QCQuartzCompositionLayer class, 159-160****QTCaptureLayer, 123**

- creating and displaying, 125-127
- current image, capturing, 127-129
- initializing capture session, 124-125

QTCaptureLayer class, 7**QTMovieLayer**

- action methods, 112
- contentsRect, 122-123
- overlays, 118-119
- overview, 7, 111-112
- QTMovieLayer-based player, 112-118
 - control buttons, 113-115
 - simple movie playback, 113
 - slider, 115
 - timer, 117-118
- time codes, 119-121

Quartz Composer, creating multivideo streams with

- compositions
 - adding filters to, 156
 - creating, 150-152
 - getting image data from, 155-156
 - input parameters, 152
 - passing parameters to, 154-155
- obtaining current image in code, 157
- and OpenGL, 159-160
- overview, 149-150
- QCCompositionLayer, 153-154
- Xcode projects, creating, 152-153

QuartzCore framework for Star Label Animation application, 211-214

- animationDidStop: method, 214
- applicationDidFinishLaunching: method, 213
- drawStar: method, 213
- StarStepsAppDelegate interface, 212

QuickTime layers

- QTCaptureLayer, 123
 - creating and displaying, 125-127
 - current image, capturing, 127-129
 - initializing capture session, 124-125
- QTMovieLayer
 - action methods, 112
 - contentsRect, 122-123
 - overlays, 118-119
 - overview, 111-112
 - QTMovieLayer-based player, 112-118
 - time codes, 119-121

QuickTime visual context, 135-136**R****random shaking, 61**

-readyToDrawNextFrame: function, 145

***Real-time Motion Graphics with Quartz Composer* (Robinson and Buchwald), 149**

receiving user input, 89-90

red background color, animating to green, 16-17

references, weak references, 180

reflection, applying with CAGradientLayer class, 168-171

-removeAnimation: method, 9

RemovedOnCompletion property (CABasicAnimation class), 40**rendering video**

- in CAOpenGLLayer, 131-132
 - advantages, 138
 - canDrawInCGLContext method, 132-135, 159-160
 - display link callback signature, 133
 - display links, 133
 - drawInCGLContext: method, 132, 136-137
 - implementation of QuickTime visual context, 135-136
 - layer timing, 133-138
 - multiple video channels, 138-139
 - OpenGLVidGridLayer, 139-148
 - VideoChannel, 139
 - VideoChannelController, 139

RepeatCount property (CABasicAnimation class), 41**RepeatDuration property (CABasicAnimation class), 41****ReplicatorView class**

- adding to window, 173-174
- creating, 171-173

resizing

- layers, 31-32
- views, 30
- windows, 28-29

resolution independence, 164**Reveal transitions, 99**

Ripple transition, 102-103

AppDelegate implementation with delegation, 106-109

AppDelegate implementation with encapsulation, 105-106

RippleLayer interface, 104-105

RippleLayer interface, 104-105

root layers, adding animation layers to, 34-35

-rotate3DTransform:, 73-74

-rotateTransform:, 72-73

rotating layers

along one axis, 72-73

along two axes, 73-74

magnitude of rotation, 73

shake animation, 61

run loops, 204

S

-scaleTransform:, 70-72

scaling layers, 10-11, 70-72

selector methods, 54

-setAnimations: method, 43

-setBounds method, 32, 109

-setCurrentTime: method, 112

-setFillMode: method, 40

-setFilters method, 93

-setFrame: method

view resizing, 30

window resizing, 28-29

-setNeedsDisplay method, 204

-setPosition method, 4

-setRemovedOnCompletion: method, 40

-setSelected: method, 182

-setString: method, 181

-setStrokeColor method, 163

-setupVisualContext: method, 135-136

shadows, 195

shake animation, 57-61

close box, adding, 61-62

random shaking, 61

rotation axis and layer geometry, 61

starting/stopping, 64-65

shape layer. *See* CAShapeLayer

simplicity in design, 22

single keyframe animation, 66

-sliderMoved: method, 115

sliders, adding to Movie Player with Overlay project, 115

Speed property (CABasicAnimation class), 40

Star Label Animation application

QuartzCore framework and object binding, 211-214

-animationDidStop: method, 214

-applicationDidFinishLaunching: method, 213

-drawStar: method, 213

StarStepsAppDelegate interface, 212

setting up Xcode project, 209

UIWindow, 210-211

Xcode project setup, 209

StarStepsAppDelegate interface, 212

starting shake animation, 64-65

-stepBackward method, 112

-stepForward method, 112

stickiness (layer animation), 39-40

“sticky” effect, 91-92

–stop method, 112

stopping shake animation, 64-65

–string: method, 181

stroke, manipulating with `CAShapeLayer`, 163-165

@synthesize keyword, 93

T

–throbAnimationDidStop:finished:context: method, 221

tiled layers, 199-202

time codes, adding to movies, 119-121

`TimeOffset` property (`CABasicAnimation` class), 41

timing

default timing function, 38-39

keyframe animation, 50-51

monitoring keyframe destinations, 52-56

timers

adding to Movie Player with Overlay project, 117-118

selector methods for, 54

toggle functionality, 91

–togglePlayback: method, 117

touch system application

overview, 214-215

`TouchableView`, 218-223

header, 218

–initWithFrame: method, 219

–throbAnimationDidStop:finished:context: method, 221

–touchBegan:withEvent: method, 219

–touchesCancelled:withEvent: method, 220

–touchesEnded:withEvent: method, 220-221

–touchesMoved:withEvent: method, 222

`TouchMeViewController`, 216-218

Application Delegate’s header, 217

Application Delegate’s implementation, 217

`TouchMeViewController.m` implementation, 216

Xcode project setup, 215-216

`TouchableView` class, 218-223

header, 218

–initWithFrame: method, 219

–throbAnimationDidStop:finished:context: method, 221

–touchBegan:withEvent: method, 219

–touchesCancelled:withEvent: method, 220

–touchesEnded:withEvent: method, 220-221

–touchesMoved:withEvent: method, 222

–touchBegan:withEvent: method, 219

–touchesCancelled:withEvent: method, 220

–touchesEnded:withEvent: method, 220-221

–touchesMoved:withEvent: method, 222

`TouchMeViewController` class, 216-218

Application Delegate’s header, 217

Application Delegate’s implementation, 217

`TouchMeViewController.m` implementation, 216

transforms

anchor points, 74-75

combining, 76-79

definition of, 69

in Cocoa Touch, 69

- nested transforms, 195
- rotate3DTransform:, 73-74
- rotateTransform:, 72-73
- scale versus bounds, 79-80
- scaleTransform:, 70-72

transitions

- filters
 - custom transitions, 101-108
 - default transitions, 98-100
- performance issues, 195

tweening, 16, 45

24 frames per second (fps), 45

U

UI keyframe animation, 56-57

- icon shake, 57-65

UIView, animations at UIView level

- building application, 223
- MainViewController, 223-226

UIWindow for Star Label Animation application, 210-211

- updateSlider: method, 117, 121
- updateTimeStamp method, 120

user interaction

- keyboard events, 188-190
- layer-backed views, 190
- mouse events
 - Color Changer sample application, 179-188
 - hit testing CALayer objects, 177-179
 - overview, 177
- user input, 23
 - receiving (filters), 89-90

V

- valueForKey: method, 157

values animation, 49

video

- rendering in CAOpenGLLayer, 131-132
 - advantages, 138
 - canDrawInCGLContext method, 132-135, 159-160
 - display link callback signature, 133
 - display links, 133
 - drawInCGLContext: method, 132, 136-137
 - implementation of QuickTime visual context, 135-136
 - layer timing, 133-138
 - rendering multiple video channels, 138-139
 - OpenGLVidGridLayer, 139-148
 - VideoChannel, 139
 - VideoChannelController, 139

VideoChannel, 139

VideoChannelController, 139

- viewDidLoad method, 162-169

views

- layer-backed views, 190
- LZContentView, 178
- resizing, 30
- TouchableView, 218-223
 - initWithFrame: method, 219
 - throbAnimationDidStop:finished:context: method, 221
 - touchBegan:withEvent: method, 219
 - touchesCancelled:withEvent: method, 220
 - touchesEnded:withEvent: method, 220-221

-touchesMoved:withEvent:
method, 222

header, 218

visibility fields, animating, 18

visual stickiness, 39-40

W-X-Y-Z

weak references, 180

windows, resizing, 28-29

Xcode projects

creating, 33-34, 152-153

setting up, 209

for OS X, 12-13

touch system application, 215-216

-zoom: method, 201

zPosition property, 18