Core Animation Programming Guide

Developer

Contents

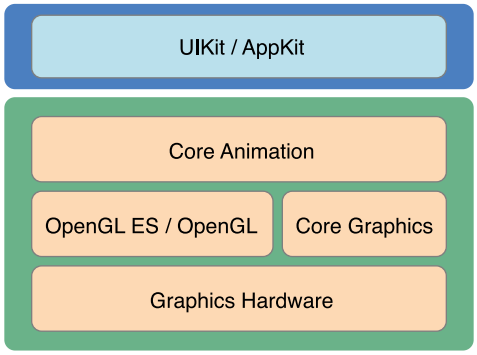
Figures, Tables, and Listings

SwiftObjective-C

About Core Animation

Core Animation is a graphics rendering and animation infrastructure available on both iOS and OS X that you use to animate the views and other visual elements of your app. With Core Animation, most of the work required to draw each frame of an animation is done for you. All you have to do is configure a few animation parameters (such as the start and end points) and tell Core Animation to start. Core Animation does the rest, handing most of the actual drawing work off to the onboard graphics hardware to accelerate the rendering. This automatic graphics acceleration results in high frame rates and smooth animations without burdening the CPU and slowing down your app.

If you are writing iOS apps, you are using Core Animation whether you know it or not. And if you are writing OS X apps, you can take advantage of Core Animation with extremely little effort. Core Animation sits beneath AppKit and UIKit and is integrated tightly into the view workflows of Cocoa and Cocoa Touch. Of course, Core Animation also has interfaces that extend the capabilities exposed by your app's views and give you more fine-grained control over your app's animations.



At a Glance

You may never need to use Core Animation directly, but when you do you should understand the role that Core Animation plays as part of your app's infrastructure.

Core Animation Manages Your App's Content

Core Animation is not a drawing system itself. It is an infrastructure for compositing and manipulating your app's content in hardware. At the heart of this infrastructure are *layer objects*, which you use to manage and manipulate your content. A layer captures your content into a bitmap that can be manipulated easily by the graphics hardware. In most apps, layers are used as a way to manage the content of views but you can also create standalone layers depending on your needs.

Relevant Chapter: Core Animation Basics (page 13), Setting Up Layer Objects (page 26),

Layer Modifications Trigger Animations

Most of the animations you create using Core Animation involve the modification of the layer's properties. Like views, layer objects have a bounds rectangle, a position onscreen, an opacity, a transform, and many other visually-oriented properties that can be modified. For most of these properties, changing the property's value results in the creation of an implicit animation whereby the layer animates from the old value to the new value. You can also explicitly animate these properties in cases where you want more control over the resulting animation behavior.

Relevant Chapters: , Animating Layer Content (page 43), Advanced Animation Tricks (page 63), Layer Style Property Animations (page 77), Animatable Properties (page 86)

Layers Can Be Organized into Hierarchies

Layers can be arranged hierarchically to create parent-child relationships. The arrangement of layers affects the visual content that they manage in a way that is similar to views. The hierarchy of a set of layers that are attached to views mirrors the corresponding view hierarchy. You can also add standalone layers into a layer hierarchy to extend the visual content of your app beyond just your views.

Relevant Chapters: Building a Layer Hierarchy (page 53)

Actions Let You Change a Layer's Default Behavior

Implicit layer animations are achieved using *action objects*, which are generic objects that implement a predefined interface. Core Animation uses action objects to implement the default set of animations normally associated with layers. You can create your own action objects to implement custom animations or use them to implement other types of behaviors too. You then assign your action object to one of the layer's properties. When that property changes, Core Animation retrieves your action object and tells it to perform its action.

Relevant Chapters: Changing a Layer's Default Behavior (page 70)

How to Use This Document

This document is intended for developers who need more control over their app's animations or who want to use layers to improve the drawing performance of their apps. This document also provides information about the integration between layers and views for both iOS and OS X. The integration between layers and views is different on iOS and OS X and understanding those differences is important to being able to create efficient animations.

Prerequisites

You should already understand the view architecture of your target platform and be familiar with how to create view-based animations. If not, you should read one of the following documents:

- For iOS apps, you should understand the view architecture described in *View Programming Guide for iOS*.
- For OS X apps, you should understand the view architecture described in *View Programming Guide*.

See Also

For examples of how to implement specific types of animations using Core Animation, see *Core Animation Cookbook*.

Core Animation Basics

Objective-CSwift

Core Animation provides a general purpose system for animating views and other visual elements of your app. Core Animation is not a replacement for your app's views. Instead, it is a technology that integrates with views to provide better performance and support for animating their content. It achieves this behavior by caching the contents of views into bitmaps that can be manipulated directly by the graphics hardware. In some cases, this caching behavior might require you to rethink how you present and manage your app's content, but most of the time you use Core Animation without ever knowing it is there. In addition to caching view content, Core Animation also defines a way to specify arbitrary visual content, integrate that content with your views, and animate it along with everything else.

You use Core Animation to animate changes to your app's views and visual objects. Most changes relate to modifying the properties of your visual objects. For example, you might use Core Animation to animate changes to a view's position, size, or opacity. When you make such a change, Core Animation animates between the current value of the property and the new value you specify. You would typically not use Core Animation to replace the content of a view 60 times a second, such as in a cartoon. Instead, you use Core Animation to move a view's content around the screen, fade that content in or out, apply arbitrary graphics transformations to the view, or change the view's other visual attributes.

Layers Provide the Basis for Drawing and Animations

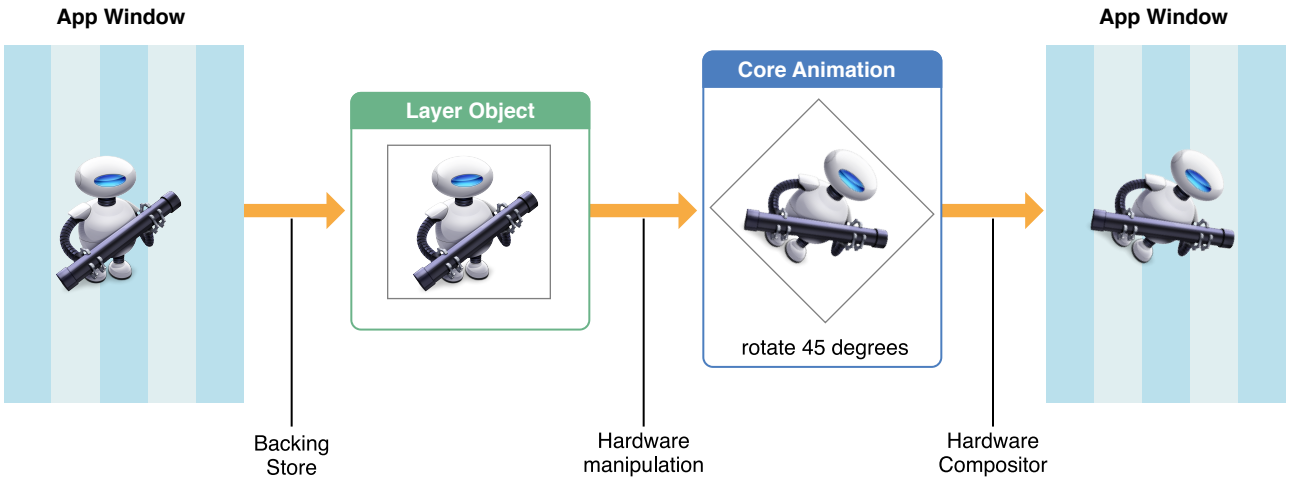
Layer objects are 2D surfaces organized in a 3D space and are at the heart of everything you do with Core Animation. Like views, layers manage information about the geometry, content, and visual attributes of their surfaces. Unlike views, layers do not define their own appearance. A layer merely manages the state information surrounding a bitmap. The bitmap itself can be the result of a view drawing itself or a fixed image that you specify. For this reason, the main layers you use in your app are considered to be model objects because they primarily manage data. This notion is important to remember because it affects the behavior of animations.

The Layer-Based Drawing Model

Most layers do not do any actual drawing in your app. Instead, a layer captures the content your app provides and caches it in a bitmap, which is sometimes referred to as the *backing store*. When you subsequently change a property of the layer, all you are doing is changing the state information associated with the layer object.

When a change triggers an animation, Core Animation passes the layer's bitmap and state information to the graphics hardware, which does the work of rendering the bitmap using the new information, as shown in Figure 1-1. Manipulating the bitmap in hardware yields much faster animations than could be done in software.

Figure 1-1 How Core Animation draws content



Because it manipulates a static bitmap, layer-based drawing differs significantly from more traditional view-based drawing techniques. With view-based drawing, changes to the view itself often result in a call to the view's `drawRect:` method to redraw content using the new parameters. But drawing in this way is expensive because it is done using the CPU on the main thread. Core Animation avoids this expense by whenever possible by manipulating the cached bitmap in hardware to achieve the same or similar effects.

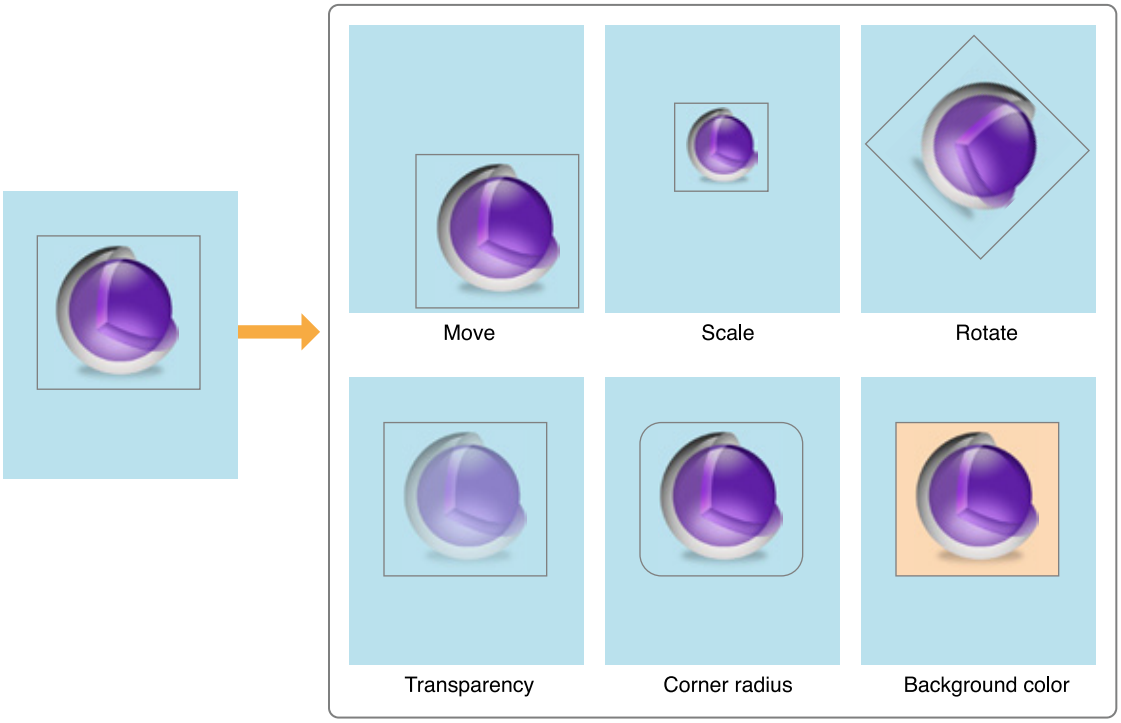
Although Core Animation uses cached content as much as possible, your app must still provide the initial content and update it from time to time. There are several ways for your app to provide a layer object with content, which are described in detail in Providing a Layer's Contents (page 30).

Layer-Based Animations

The data and state information of a layer object is decoupled from the visual presentation of that layer's content onscreen. This decoupling gives Core Animation a way to interpose itself and animate the change from the old state values to new state values. For example, changing a layer's position property causes Core Animation

to move the layer from its current position to the newly specified position. Similar changes to other properties cause appropriate animations. Figure 1-2 illustrates a few of the types of animations you can perform on layers. For a list of layer properties that trigger animations, see Animatable Properties (page 86).

Figure 1-2 Examples of animations you can perform on layers



During the course of an animation, Core Animation does all of the frame-by-frame drawing for you in hardware. All you have to do is specify the start and end points of the animation and let Core Animation do the rest. You can also specify custom timing information and animation parameters as needed; however, Core Animation provides suitable default values if you do not.

For more information about how to initiate animations and configure animation parameters, see Animating Layer Content (page 43).

Layer Objects Define Their Own Geometry

One of the jobs of a layer is to manage the visual geometry for its content. The visual geometry encompasses information about the bounds of that content, its position on the screen, and whether the layer has been rotated, scaled, or transformed in any way. Like a view, a layer has frame and bounds rectangles that you can use to position the layer and its content. Layers also have other properties that views do not have, such as an anchor point, which defines the point around which manipulations occur. The way you specify some aspects of layer geometry also differs from how you specify that information for a view.

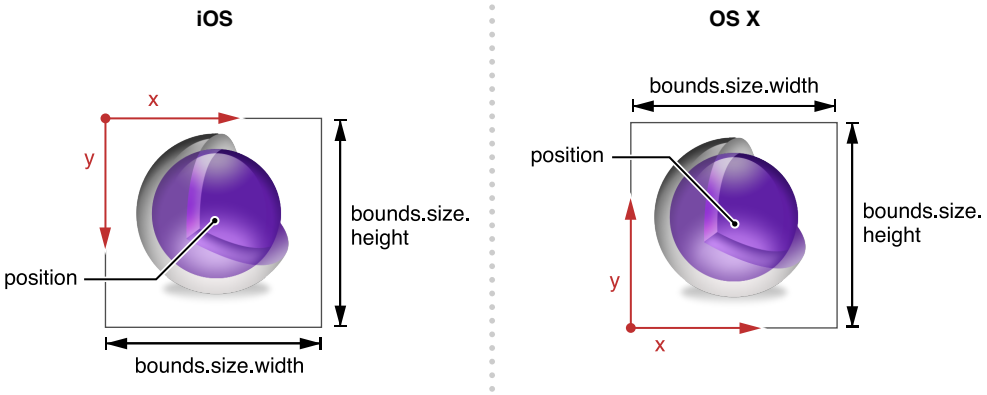
Layers Use Two Types of Coordinate Systems

Layers make use of both *point-based coordinate systems* and *unit coordinate systems* to specify the placement of content. Which coordinate system is used depends on the type of information being conveyed. Point-based coordinates are used when specifying values that map directly to screen coordinates or must be specified relative to another layer, such as for the layer's `position` property. Unit coordinates are used when the value should not be tied to screen coordinates because it is relative to some other value. For example, the layer's `anchorPoint` property specifies a point relative to the bounds of the layer itself, which can change.

Among the most common uses for point-based coordinates is to specify the size and position of the layer, which you do using the layer's `bounds` and `position` properties. The `bounds` defines the coordinate system of the layer itself and encompasses the layer's size on the screen. The `position` property defines the location of the layer relative to its parent's coordinate system. Although layers have a `frame` property, that property is actually derived from the values in the `bounds` and `position` properties and is used less frequently.

The orientation of a layer's `bounds` and `frame` rectangles always matches the default orientation of the underlying platform. Figure 1-3 shows the default orientations of the bounds rectangle on both iOS and OS X. In iOS, the origin of the bounds rectangle is in the top-left corner of the layer by default, and in OS X it is in the bottom-left corner. If you share Core Animation code between iOS and OS X versions of your app, you must take such differences into consideration.

Figure 1-3 The default layer geometries for iOS and OS X

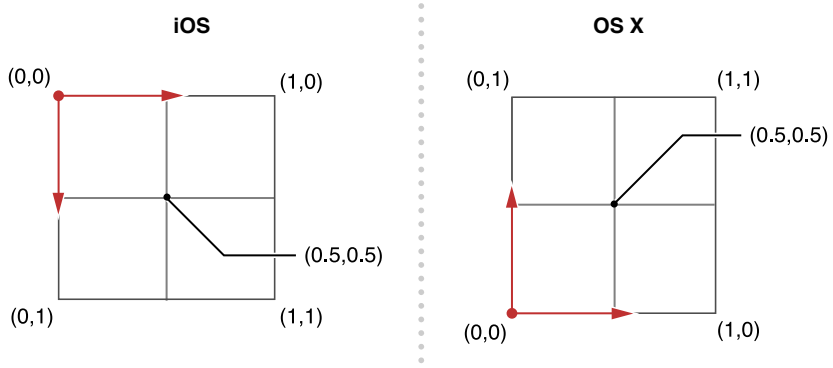


One thing to note in Figure 1-3 (page 16) is that the `position` property is located in the middle of the layer. That property is one of several whose definition changes based on the value in the layer's `anchorPoint` property. The anchor point represents the point from which certain coordinates originate and is described in more detail in Anchor Points Affect Geometric Manipulations (page 17).

The anchor point is one of several properties that you specify using the unit coordinate system. Core Animation uses unit coordinates to represent properties whose values might change when the layer's size changes. You can think of the unit coordinates as specifying a percentage of the total possible value. Every coordinate in

the unit coordinate space has a range of `0.0` to `1.0`. For example, along the x-axis, the left edge is at the coordinate `0.0` and the right edge is at the coordinate `1.0`. Along the y-axis, the orientation of unit coordinate values changes depending on the platform, as shown in Figure 1-4.

Figure 1-4 The default unit coordinate systems for iOS and OS X



Note: Until OS X 10.8, the `geometryFlipped` property was a way to change the default orientation of a layer's y-axis when needed. Use of this property was sometimes necessary to correct the orientation of a layer when flip transforms were in involved. For example, if a parent view used a flip transform, the contents of its child views (and their corresponding layers) would often be inverted. In such cases, setting the `geometryFlipped` property of the child layers to YES was an easy way to correct the problem. In OS X 10.8 and later, AppKit manages this property for you and you should not modify it. For iOS apps, it is recommended that you do not use the `geometryFlipped` property at all.

All coordinate values, whether they are points or unit coordinates are specified as floating-point numbers. The use of floating-point numbers allows you to specify precise locations that might fall between normal coordinate values. The use of floating-point values is convenient, especially during printing or when drawing to a Retina display where one point might be represented by multiple pixels. Floating-point values allow you to ignore the underlying device resolution and just specify values at the precision you need.

Anchor Points Affect Geometric Manipulations

Geometry related manipulations of a layer occur relative to that layer's anchor point, which you can access using the layer's `anchorPoint` property. The impact of the anchor point is most noticeable when manipulating the `position` or `transform` properties of the layer. The position property is always specified relative to the layer's anchor point, and any transformations you apply to the layer occur relative to the anchor point as well.

Figure 1-5 demonstrates how changing the anchor point from its default value to a different value affects the `position` property of a layer. Even though the layer has not moved within its parents' bounds, moving the anchor point from the center of the layer to the layer's bounds origin changes the value in the `position` property.

Figure 1-5 How the anchor point affects the layer's position property

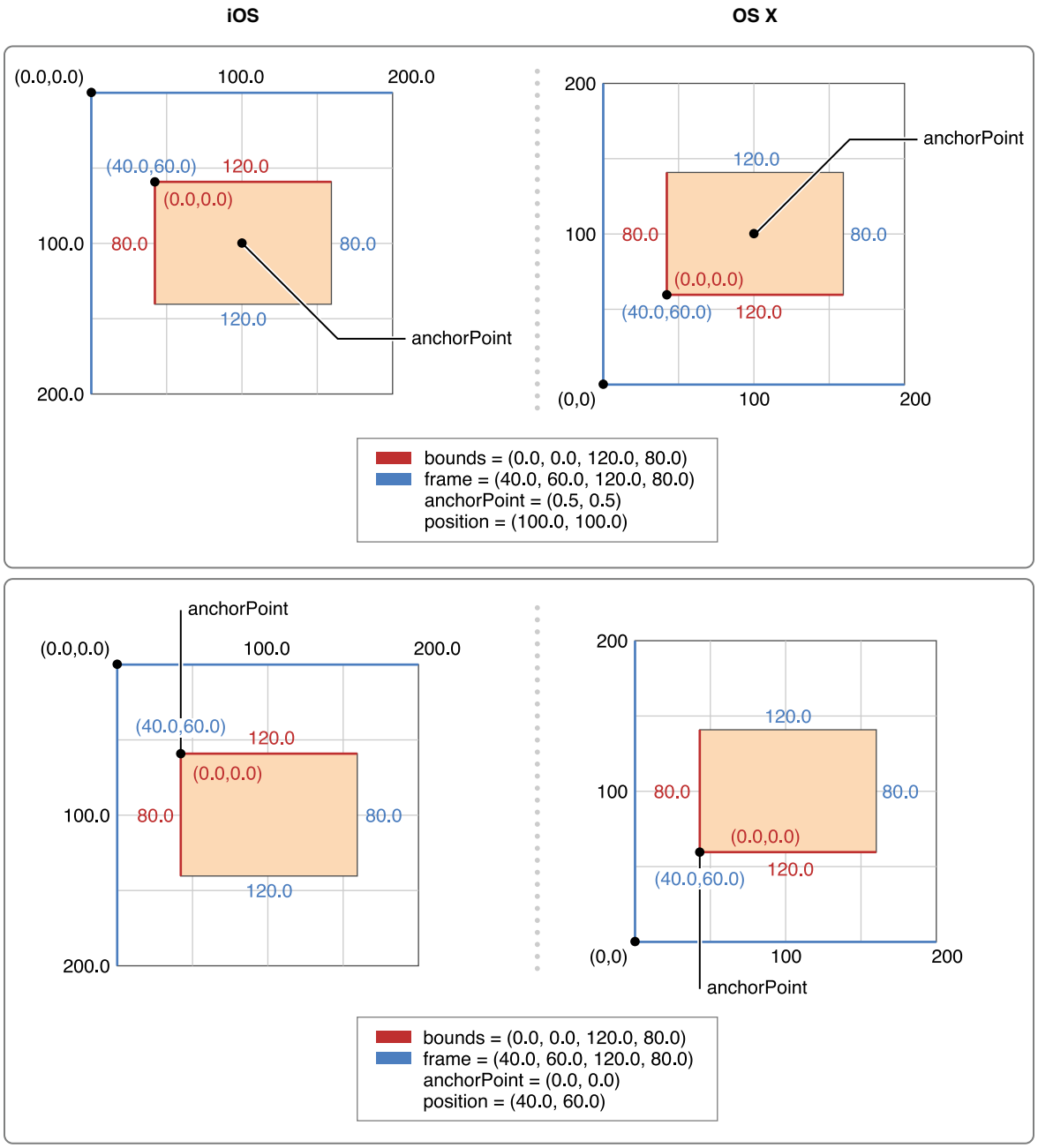
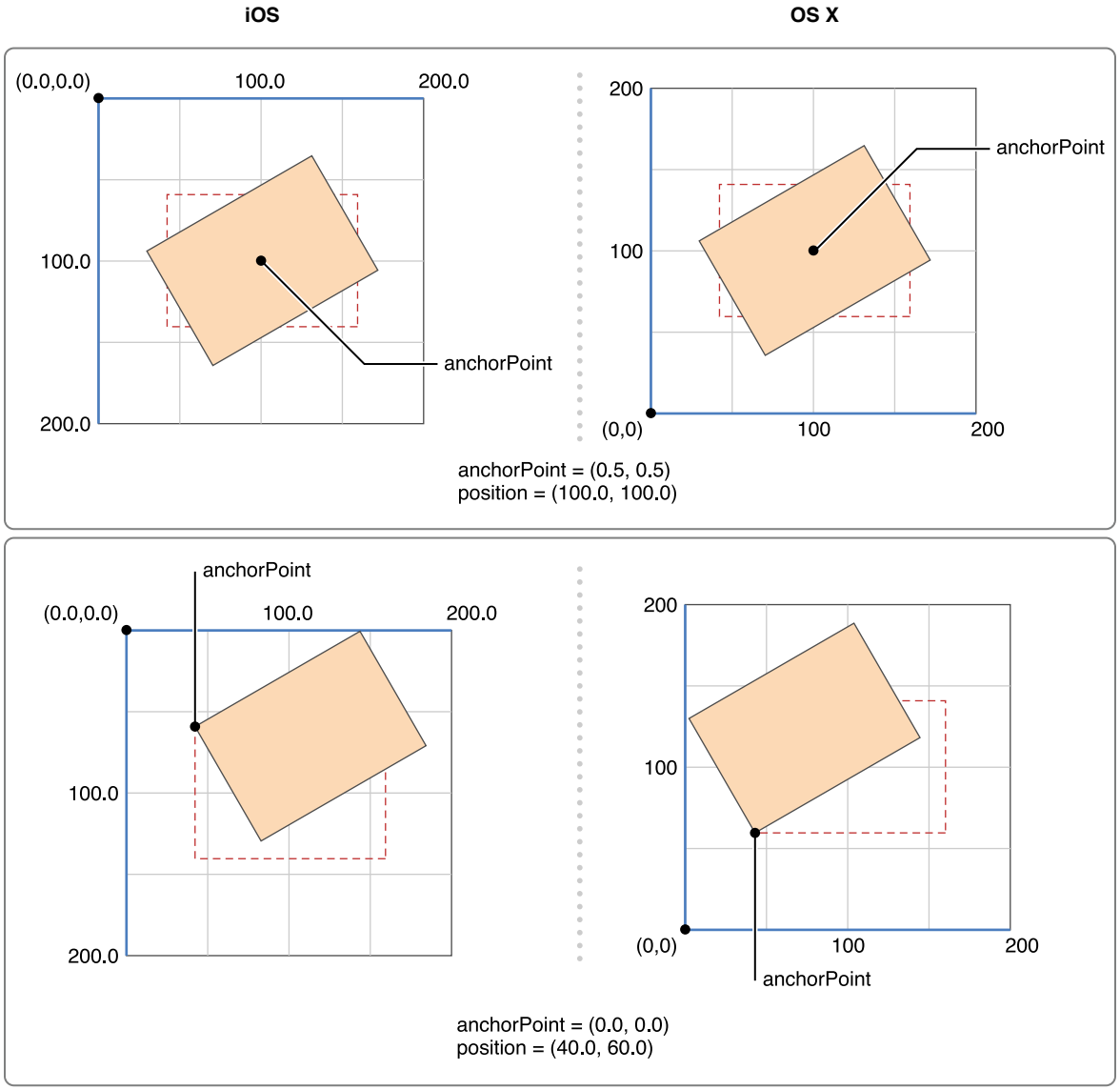


Figure 1-6 shows how changing the anchor point affects transforms applied to the layer. When you apply a rotation transform to the layer, the rotations occur around the anchor point. Because the anchor point is set to the middle of the layer by default, this normally creates the kind of rotation behavior that you would expect. However, if you change the anchor point, the results of the rotation are different.

Figure 1-6 How the anchor point affects layer transformations



Layers Can Be Manipulated in Three Dimensions

Every layer has two transform matrices that you can use to manipulate the layer and its contents. The `transform` property of `CALayer` specifies the transforms that you want to apply both to the layer and its embedded sublayers. Normally you use this property when you want to modify the layer itself. For example, you might

use that property to scale or rotate the layer or change its position temporarily. The `sublayerTransform` property defines additional transformations that apply only to the sublayers and is used most commonly to add a perspective visual effect to the contents of a scene.

Transforms work by multiplying coordinate values through a matrix of numbers to get new coordinates that represent the transformed versions of the original points. Because Core Animation values can be specified in three dimensions, each coordinate point has four values that must be multiplied through a four-by-four matrix, as shown in Figure 1-7. In Core Animation, the transform in the figure is represented by the `CATransform3D` type. Fortunately, you do not have to modify the fields of this structure directly to perform standard transformations. Core Animation provides a comprehensive set of functions for creating scale, translation, and rotation matrices and for doing matrix comparisons. In addition to manipulating transforms using functions, Core Animation extends key-value coding support to allow you to modify a transform using key paths. For a list of key paths you can modify, see CATransform3D Key Paths (page 92).

Figure 1-7 Converting a coordinate using matrix math

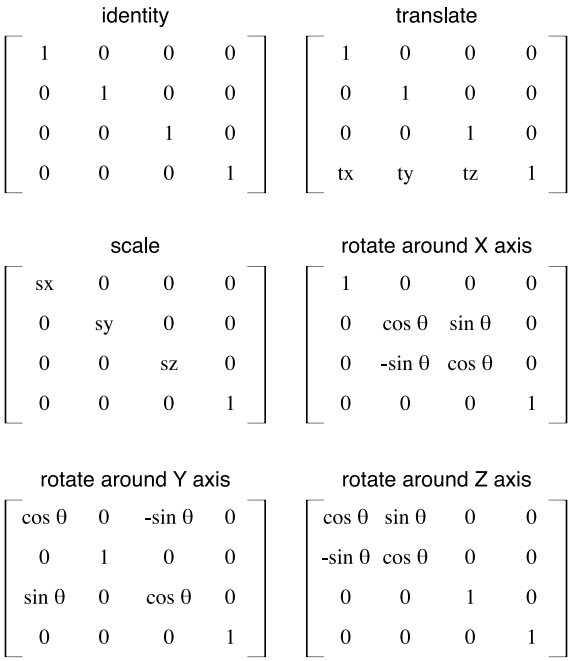
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} * \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

coordinate | transform | transformed coordinate

Figure 1-8 shows the matrix configurations for some of the more common transformations you can make. Multiplying any coordinate by the identity transform returns the exact same coordinate. For other transformations, how the coordinate is modified depends entirely on which matrix components you change.

For example, to translate along the x-axis only, you would supply a nonzero value for the `tx` component of the translation matrix and leave the `ty` and `tz` values to 0. For rotations, you would provide the appropriate sine and cosine values of the target rotation angle.

Figure 1-8 Matrix configurations for common transformations



For information about the functions you use to create and manipulate transforms, see *Core Animation Function Reference* .

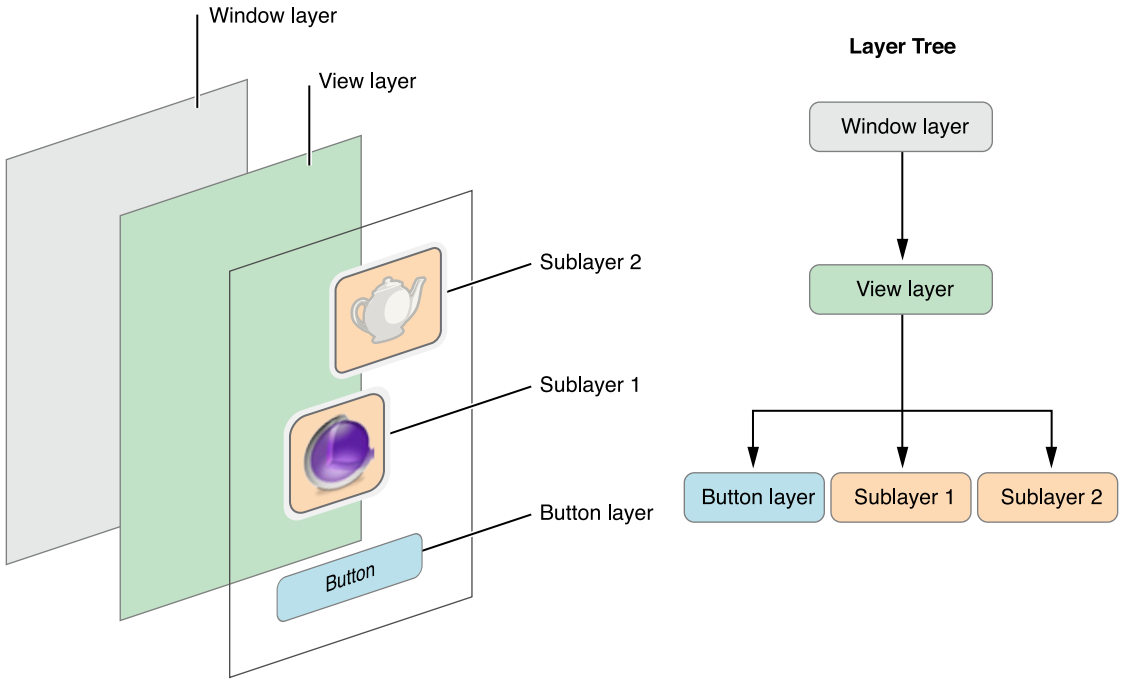
Layer Trees Reflect Different Aspects of the Animation State

An app using Core Animation has three sets of layer objects. Each set of layer objects has a different role in making the content of your app appear onscreen:

- Objects in the *model layer tree* (or simply “layer tree”) are the ones your app interacts with the most. The objects in this tree are the model objects that store the target values for any animations. Whenever you change the property of a layer, you use one of these objects.
- Objects in the *presentation tree* contain the in-flight values for any running animations. Whereas the layer tree objects contain the target values for an animation, the objects in the presentation tree reflect the current values as they appear onscreen. You should never modify the objects in this tree. Instead, you use these objects to read current animation values, perhaps to create a new animation starting at those values.
- Objects in the *render tree* perform the actual animations and are private to Core Animation.

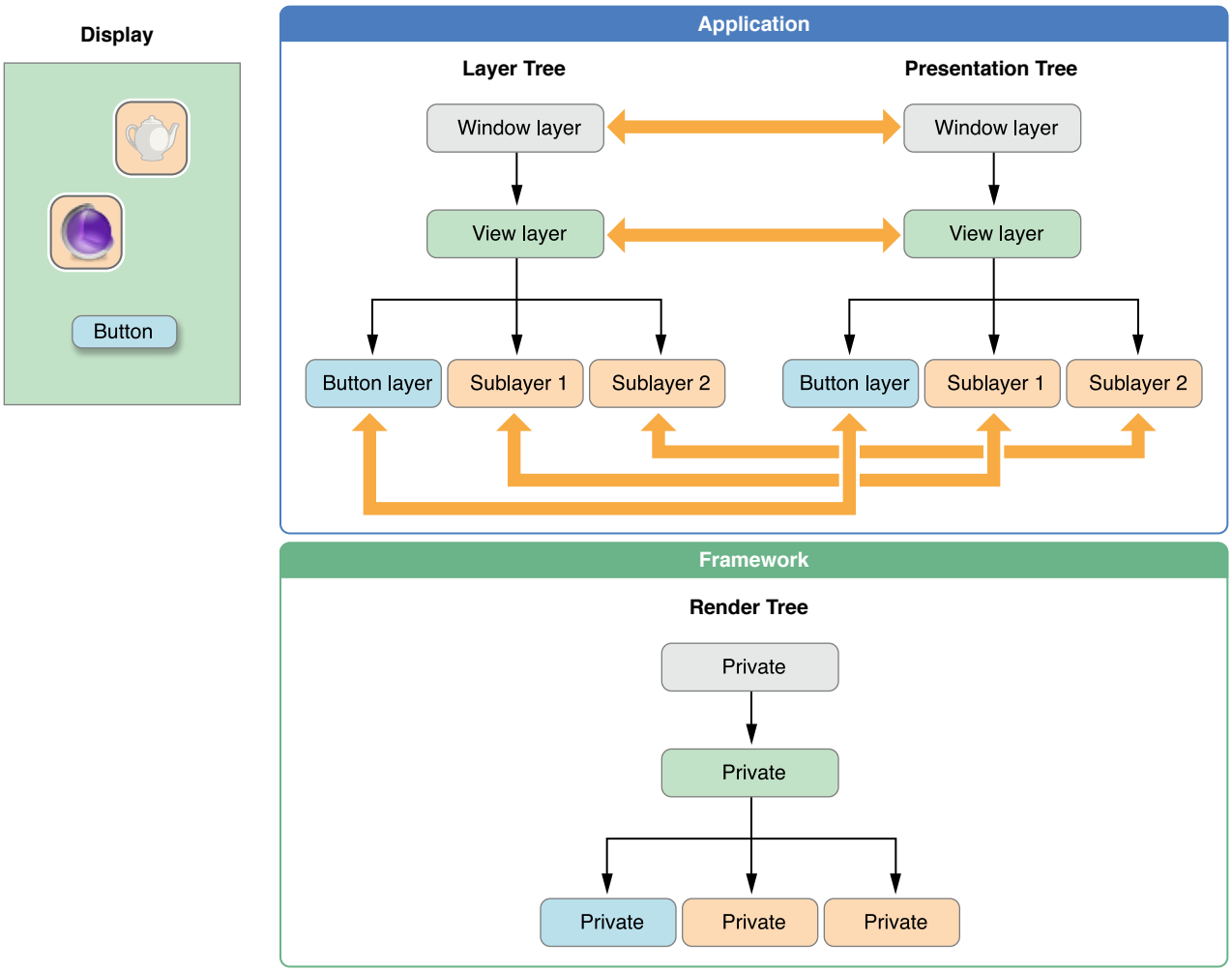
Each set of layer objects is organized into a hierarchical structure like the views in your app. In fact, for an app that enables layers for all of its views, the initial structure of each tree matches the structure of the view hierarchy exactly. However, an app can add additional layer objects—that is, layers not associated with a view—into the layer hierarchy as needed. You might do this in situations to optimize your app's performance for content that does not require all the overhead of a view. Figure 1-9 shows the breakdown of layers found in a simple iOS app. The window in the example contains a content view, which itself contains a button view and two standalone layer objects. Each view has a corresponding layer object that forms part of the layer hierarchy.

Figure 1-9 Layers associated with a window



For every object in the layer tree, there is a matching object in the presentation and render trees, as shown in Figure 1-10. As was previously mentioned, apps primarily work with objects in the layer tree but may at times access objects in the presentation tree. Specifically, accessing the `presentationLayer` property of an object in the layer tree returns the corresponding object in the presentation tree. You might want to access that object to read the current value of a property that is in the middle of an animation.

Figure 1-10 The layer trees for a window



Important: You should access objects in the presentation tree only while an animation is in flight. While an animation is in progress, the presentation tree contains the layer values as they appear onscreen at that instant. This behavior differs from the layer tree, which always reflects the last value set by your code and is equivalent to the final state of the animation.

The Relationship Between Layers and Views

Layers are not a replacement for your app's views—that is, you cannot create a visual interface based solely on layer objects. Layers provide infrastructure for your views. Specifically, layers make it easier and more efficient to draw and animate the contents of views and maintain high frame rates while doing so. However, there are many things that layers do not do. Layers do not handle events, draw content, participate in the responder chain, or do many other things. For this reason, every app must still have one or more views to handle those kinds of interactions.

In iOS, every view is backed by a corresponding layer object but in OS X you must decide which views should have layers. In OS X v10.8 and later, it probably makes sense to add layers to all of your views. However, you are not required to do so and can still disable layers in cases where the overhead is unwarranted and unneeded. Layers do increase your app's memory overhead somewhat but their benefits often outweigh the disadvantage, so it is always best to test the performance of your app before disabling layer support.

When you enable layer support for a view, you create what is referred to as a *layer-backed view*. In a layer-backed view, the system is responsible for creating the underlying layer object and for keeping that layer in sync with the view. All iOS views are layer-backed and most views in OS X are as well. However, in OS X, you can also create a *layer-hosting view*, which is a view where you supply the layer object yourself. For a layer-hosting view, AppKit takes a hands off approach with managing the layer and does not modify it in response to view changes.

Note: For layer-backed views, it is recommended that you manipulate the view, rather than its layer, whenever possible. In iOS, views are just a thin wrapper around layer objects, so any manipulations you make to the layer usually work just fine. But there are cases in both iOS and OS X where manipulating the layer instead of the view might not yield the desired results. Wherever possible, this document points out those pitfalls and tries to provide ways to help you work around them.

In addition to the layers associated with your views, you can also create layer objects that do not have a corresponding view. You can embed these standalone layer objects inside of any other layer object in your app, including those that are associated with a view. You typically use standalone layer objects as part of a specific optimization path. For example, if you wanted to use the same image in multiple places, you could

load the image once and associate it with multiple standalone layer objects and add those objects to the layer tree. Each layer then refers to the source image rather than trying to create its own copy of that image in memory.

For information about how to enable layer support for your app's views, see Enabling Core Animation Support in Your App (page 26). For information on how to create a layer object hierarchy, and for tips on when you might do so, see Building a Layer Hierarchy (page 53).

Setting Up Layer Objects

SwiftObjective-C

Layer objects are at the heart of everything you do with Core Animation. Layers manage your app's visual content and provide options for modifying the style and visual appearance of that content. Although iOS apps have layer support enabled automatically, developers of OS X apps must enable it explicitly before they can take advantage of the performance benefits. Once enabled, you need to understand how to configure and manipulate your app's layers to get the effects you want.

Enabling Core Animation Support in Your App

In iOS apps, Core Animation is always enabled and every view is backed by a layer. In OS X, apps must explicitly enable Core Animation support by doing the following:

- Link against the QuartzCore framework. (iOS apps must link against this framework only if they use Core Animation interfaces explicitly.)
- Enable layer support for one or more of your `NSView` objects by doing one of the following:
    - In your nib files, use the View Effects inspector to enable layer support for your views. The inspector displays checkboxes for the selected view and its subviews. It is recommended that you enable layer support in the content view of your window whenever possible.
    - For views you create programmatically, call the view's `setWantsLayer:` method and pass a value of `YES` to indicate that the view should use layers.

Enabling layer support in one of the preceding ways creates a layer-backed view. With a layer-backed view, the system takes responsibility for creating the underlying layer object and for keeping that layer updated. In OS X, it is also possible to create a layer-hosting view, whereby your app actually creates and manages the underlying layer object. (You cannot create layer-hosting views in iOS.) For more information on how to create a layer-hosting view, see Layer Hosting Lets You Change the Layer Object in OS X (page 28).

Changing the Layer Object Associated with a View

Layer-backed views create an instance of the `CALayer` class by default, and in most cases you might not need a different type of layer object. However, Core Animation provides different layer classes, each of which provides specialized capabilities that you might find useful. Choosing a different layer class might enable you to improve performance or support a specific type of content in a simple way. For example, the `CATiledLayer` class is optimized for displaying large images in an efficient manner.

Changing the Layer Class Used by UIView

You can change the type of layer used by an iOS view by overriding the view's `layerClass` method and returning a different class object. Most iOS views create a `CALayer` object and use that layer as the backing store for its content. For most of your own views, this default choice is a good one and you should not need to change it. But you might find that a different layer class is more appropriate in certain situations. For example, you might want to change the layer class in the following situations:

- Your view draws content using Metal or OpenGL ES, in which case you would use a `CAMetalLayer` or `CAEAGLLayer` object.
- There is a specialized layer class that offers better performance.
- You want to take advantage of some specialized Core Animation layer classes, such as particle emitters or replicators.

Changing the layer class of a view is very straightforward; an example is shown in Listing 2-1. All you have to do is override the `layerClass` method and return the class object you want to use instead. Prior to display, the view calls the `layerClass` method and uses the returned class to create a new layer object for itself. Once created, a view's layer object cannot be changed.

Listing 2-1 Specifying the layer class of an iOS view

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

For a list of layer classes and how you use them, see Different Layer Classes Provide Specialized Behaviors (page 29).

Changing the Layer Class Used By NSView

You can change the default layer class used by an `NSView` object by overriding the `makeBackingLayer` method. In your implementation of this method, create and return the layer object that you want AppKit to use to back your custom view. You might override this method in situations where you want to use a custom layer such as a scrolling or tiled layer.

For a list of layer classes and how you use them, see Different Layer Classes Provide Specialized Behaviors (page 29).

Layer Hosting Lets You Change the Layer Object in OS X

A layer-hosting view is an `NSView` object for which you create and manage the underlying layer object yourself. You might use layer hosting in situations where you want to control the type of layer object associated with the view. For example, you might create a layer-hosting view so that you can assign a layer class other than the default `CALayer` class. You might also use it in situations where you want to use a single view to manage a hierarchy of standalone layers.

When you call the `setLayer:` method of your view and provide a layer object, AppKit takes a hands-off approach to that layer. Normally, AppKit updates a view's layer object but in the layer-hosting situation it does not for most properties.

To create a layer-hosting view, create your layer object and associate it with the view before displaying the view onscreen, as shown in Listing 2-2. In addition to setting the layer object, you must still call the `setWantsLayer:` method to let the view know that it should use layers.

Listing 2-2 Creating a layer-hosting view

```
// Create myView...

[myView setWantsLayer:YES];
CATiledLayer* hostedLayer = [CATiledLayer layer];
[myView setLayer:hostedLayer];

// Add myView to a view hierarchy.
```

If you choose to host layers yourself, you must set the `contentsScale` property yourself and provide high-resolution content at appropriate times. For more information about high-resolution content and scale factors, see Working with High-Resolution Images (page 35).

Different Layer Classes Provide Specialized Behaviors

Core Animation defines many standard layer classes, each of which was designed for a specific use case. The `CALayer` class is the root class for all layer objects. It defines the behavior that all layer objects must support and is the default type used by layer-backed views. However, you can also specify one of the layer classes in Table 2-1.

Table 2-1 `CALayer` subclasses and their uses

Class	Usage
`CAEmitterLayer`	Used to implement a Core Animation–based particle emitter system. The emitter layer object controls the generation of the particles and their origin.
`CAGradientLayer`	Used to draw a color gradient that fills the shape of the layer (within the bounds of any rounded corners).
`CAMetalLayer`	Used to set up and vend drawable textures for rendering layer content using Metal.
`CAEAGLLayer/CAOpenGLLayer`	Used to set up the backing store and context for rendering layer content using OpenGL ES (iOS) or OpenGL (OS X).
`CAReplicatorLayer`	Used when you want to make copies of one or more sublayers automatically. The replicator makes the copies for you and uses the properties you specify to alter the appearance or attributes of the copies.
`CAScrollLayer`	Used to manage a large scrollable area composed of multiple sublayers.
`CAShapeLayer`	Used to draw a cubic Bezier spline. Shape layers are advantageous for drawing path-based shapes because they always result in a crisp path, as opposed to a path you draw into a layer's backing store, which would not look as good when scaled. However, the crisp results do involve rendering the shape on the main thread and caching the results.
`CATextLayer`	Used to render a plain or attributed string of text.
`CATiledLayer`	Used to manage a large image that can be divided into smaller tiles and rendered individually with support for zooming in and out of the content.
`CATransformLayer`	Used to render a true 3D layer hierarchy, rather than the flattened layer hierarchy implemented by other layer classes.

Class	Usage
`QCCompositionLayer`	Used to render a Quartz Composer composition. (OS X only)

Providing a Layer's Contents

Layers are data objects that manage content provided by your app. A layer's content consists of a bitmap containing the visual data you want to display. You can provide the content for that bitmap in one of three ways:

- Assign an image object directly to the layer object's `contents` property. (This technique is best for layer content that never, or rarely, changes.)
- Assign a delegate object to the layer and let the delegate draw the layer's content. (This technique is best for layer content that might change periodically and can be provided by an external object, such as a view.)
- Define a layer subclass and override one of its drawing methods to provide the layer contents yourself. (This technique is appropriate if you have to create a custom layer subclass anyway or if you want to change the fundamental drawing behavior of the layer.)

The only time you need to worry about providing content for a layer is when you create the layer object yourself. If your app contains nothing but layer-backed views, you do not have to worry about using any of the preceding techniques to provide layer content. Layer-backed views automatically provide the contents for their associated layers in the most efficient way possible.

Using an Image for the Layer's Content

Because a layer is just a container for managing a bitmap image, you can assign an image directly to the layer's `contents` property. Assigning an image to the layer is easy and lets you specify the exact image you want to display onscreen. The layer uses the image object you provide directly and does not attempt to create its own copy of that image. This behavior can save memory in cases where your app uses the same image in multiple places.

The image you assign to a layer must be a `CGImageRef` type. (In OS X v10.6 and later, you can also assign an `NSImage` object.) When assigning images, remember to provide an image whose resolution matches the resolution of the native device. For devices with Retina displays, this might also require you to adjust the `contentsScale` property of the image. For information on using high-resolution content with your layers, see Working with High-Resolution Images (page 35).

Using a Delegate to Provide the Layer's Content

If the content of your layer changes dynamically, you can use a delegate object to provide and update that content when needed. At display time, the layer calls the methods of your delegate to provide the needed content:

- If your delegate implements the `displayLayer:` method, that implementation is responsible for creating a bitmap and assigning it to the layer's `contents` property.
- If your delegate implements the `drawLayer:inContext:` method, Core Animation creates a bitmap, creates a graphics context to draw into that bitmap, and then calls your delegate method to fill the bitmap. All your delegate method has to do is draw into the provided graphics context.

The delegate object must implement either the `displayLayer:` or `drawLayer:inContext:` method. If the delegate implements both the `displayLayer:` and `drawLayer:inContext:` method, the layer calls only the `displayLayer:` method.

Overriding the `displayLayer:` method is most appropriate for situations when your app prefers to load or create the bitmaps it wants to display. Listing 2-3 shows a sample implementation of the `displayLayer:` delegate method. In this example, the delegate uses a helper object to load and display the image it needs. The delegate method selects which image to display based on its own internal state, which in the example is a custom property called `displayYesImage`.

Listing 2-3 Setting the layer contents directly

```
- (void)displayLayer:(CALayer *)theLayer {
    // Check the value of some state property
    if (self.displayYesImage) {
        // Display the Yes image
        theLayer.contents = [someHelperObject loadStateYesImage];
    }
    else {
        // Display the No image
        theLayer.contents = [someHelperObject loadStateNoImage];
    }
}
```

If you do not have prerendered images or a helper object to create bitmaps for you, your delegate can draw the content dynamically using the `drawLayer:inContext:` method. Listing 2-4 shows a sample implementation of the `drawLayer:inContext:` method. In this example, the delegate draws a simple curved path using a fixed width and the current rendering color.

Listing 2-4 Drawing the contents of a layer

```
- (void)drawLayer:(CALayer *)theLayer inContext:(CGContextRef)theContext {
    CGMutablePathRef thePath = CGPathCreateMutable();

    CGPathMoveToPoint(thePath,NULL,15.0f,15.f);
    CGPathAddCurveToPoint(thePath,
                          NULL,
                          15.f,250.0f,
                          295.0f,250.0f,
                          295.0f,15.0f);

    CGContextBeginPath(theContext);
    CGContextAddPath(theContext, thePath);

    CGContextSetLineWidth(theContext, 5);
    CGContextStrokePath(theContext);

    // Release the path
    CFRelease(thePath);
}
```

For layer-backed views with custom content, you should continue to override the view's methods to do your drawing. A layer-backed view automatically makes itself the delegate of its layer and implements the needed delegate methods, and you should not change that configuration. Instead, you should implement your view's `drawRect:` method to draw your content.

In OS X v10.8 and later, an alternative to drawing is to provide a bitmap by overriding the `wantsUpdateLayer` and `updateLayer` methods of your view. Overriding `wantsUpdateLayer` and returning `YES` causes the `NSView` class to follow an alternate rendering path. Instead of calling `drawRect:`, the view calls your `updateLayer` method, the implementation of which must assign a bitmap directly to the layer's `contents` property. This is the one scenario where AppKit expects you to set the contents of a view's layer object directly.

Providing Layer Content Through Subclassing

If you are implementing a custom layer class anyway, you can override the drawing methods of your layer class to do any drawing. It is uncommon for a layer object to generate custom content itself, but layers certainly can manage the display of content. For example, the `CATiledLayer` class manages a large image by breaking it into smaller tiles that can be managed and rendered individually. Because only the layer has information about which tiles need to be rendered at any given time, it manages the drawing behavior directly.

When subclassing, you can use either of the following techniques to draw your layer's content:

- Override the layer's `display` method and use it to set the `contents` property of the layer directly.
- Override the layer's `drawInContext:` method and use it to draw into the provided graphics context.

Which method you override depends on how much control you need over the drawing process. The `display` method is the main entry point for updating the layer's contents, so overriding that method puts you in complete control of the process. Overriding the `display` method also means that you are responsible for creating the `CGImageRef` to be assigned to the `contents` property. If you just want to draw content (or have your layer manage the drawing operation), you can override the `drawInContext:` method instead and let the layer create the backing store for you.

Tweaking the Content You Provide

When you assign an image to the `contents` property of a layer, the layer's `contentsGravity` property determines how that image is manipulated to fit the current bounds. By default, if an image is bigger or smaller than the current bounds, the layer object scales the image to fit within the available space. If the aspect ratio of the layer's bounds is different than the aspect ratio of the image, this can cause the image to be distorted. You can use the `contentsGravity` property to ensure that your content is presented in the best way possible.

The values you can assign to the `contentsGravity` property are divided into two categories:

- The position-based gravity constants allow you to pin your image to a particular edge or corner of the layer's bounds rectangle without scaling the image.
- The scaling-based gravity constants allow you to stretch the image using one of several options, some of which preserve the aspect ratio and some of which do not.

Figure 2-1 shows the how the position-based gravity settings affect your images. With the exception of the `kCAGravityCenter` constant, each constant pins the image to a particular edge or corner of the layer's bounds rectangle. The `kCAGravityCenter` constant centers the image in the layer. None of these constants

cause the image to be scaled in any way, so the image is always rendered at its original size. If the image is bigger than the layer's bounds, this may result in portions of the image being clipped, and if the image is smaller, the portions of the layer that are not covered by the image reveal the layer's background color, if set.

Figure 2-1 Position-based gravity constants for layers

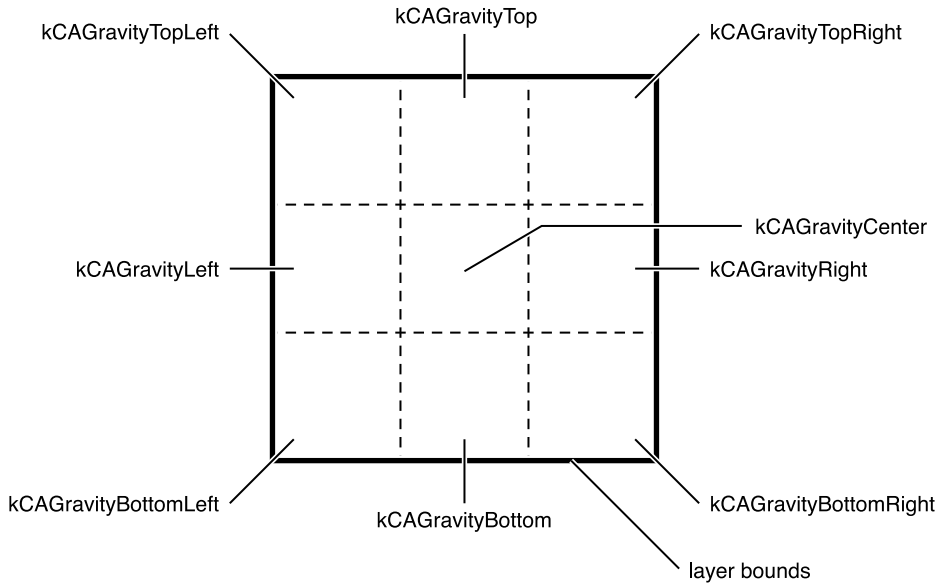
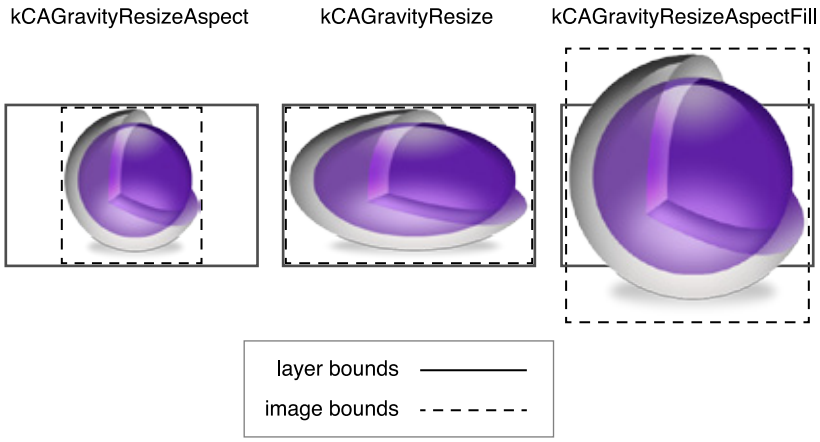


Figure 2-2 shows how the scaling-based gravity constants affect your images. All of these constants scale the image if it does not fit exactly within the bounds rectangle of the layer. The difference between the modes is how they deal with the image's original aspect ratio. Some modes preserve it and others do not. By default, a layer's `contentsGravity` property is set to the `kCAGravityResize` constant, which is the only mode that does not preserve the image aspect ratio.

Figure 2-2 Scaling-based gravity constants for layers



Working with High-Resolution Images

Layers do not have any inherent knowledge of the resolution of the underlying device's screen. A layer simply stores a pointer to your bitmap and displays it in the best way possible given the available pixels. If you assign an image to a layer's `contents` property, you must tell Core Animation about the image's resolution by setting the layer's `contentsScale` property to an appropriate value. The default value of the property is `1.0`, which is appropriate for images intended to be displayed on standard resolution screens. If your image is intended for a Retina display, set the value of this property to `2.0`.

Changing the value of the `contentsScale` property is only necessary if you are assigning a bitmap to your layer directly. A layer-backed view in UIKit and AppKit automatically sets the scale factor of its layer to an appropriate value based on the screen resolution and the content managed by the view. For example, if you assign an `NSImage` object to the `contents` property of a layer in OS X, AppKit looks to see if there are both standard- and high-resolution variants of the image. If there are, AppKit uses the correct variant for the current resolution and sets the value of the `contentsScale` property to match.

In OS X, the position-based gravity constants affect the way image representations are chosen from an `NSImage` object assigned to the layer. Because these constants do not cause the image to be scaled, Core Animation relies on the `contentsScale` property to pick the image representation with the most appropriate pixel density.

In OS X, the layer's delegate can implement the `layer:shouldInheritContentsScale:fromWindow:` method and use it to respond to changes in the scale factor. AppKit automatically calls that method whenever the resolution for a given window changes, possibly because the window moved between a standard-resolution and high-resolution screens. Your implementation of this method should return `YES` if the delegate supports changing the resolution of the layer's image. The method should then update the layer's contents as needed to reflect the new resolution.

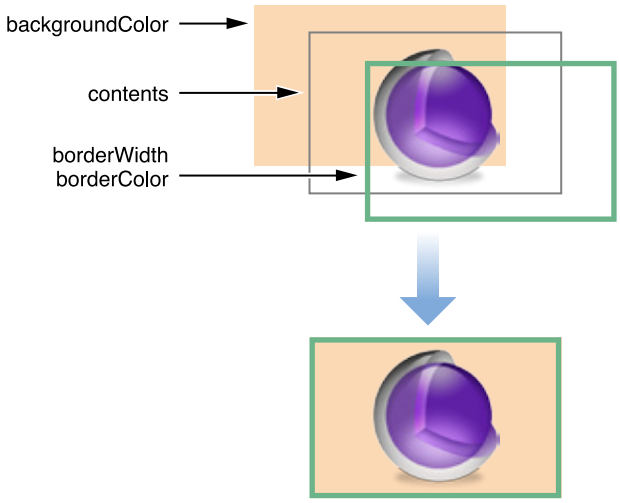
Adjusting a Layer's Visual Style and Appearance

Layer objects have built in visual adornments such as a border and background color that you can use to supplement the layer's main contents. Because these visual adornments do not require any rendering on your part, they make it possible to use layers as standalone entities in some situations. All you have to do is set a property on the layer and the layer handles the necessary drawing, including any animations. For additional illustrations of how these visual adornments affect the appearance of a layer, see Layer Style Property Animations (page 77).

Layers Have Their Own Background and Border

A layer can display a filled background and a stroked border in addition to its image-based contents. The background color is rendered behind the layer's contents image and the border is rendered on top of that image, as shown in Figure 2-3. If the layer contains sublayers, they also appear underneath the border. Because the background color sits behind your image, that color shines through any transparent portions of your image.

Figure 2-3 Adding a border and background to a layer



Listing 2-5 shows the code needed to set the background color and border for a layer. All of these properties are animatable.

Listing 2-5 Setting the background color and border of a layer

```
myLayer.backgroundColor = [NSColor greenColor].CGColor;
myLayer.borderColor = [NSColor blackColor].CGColor;
myLayer.borderWidth = 3.0;
```

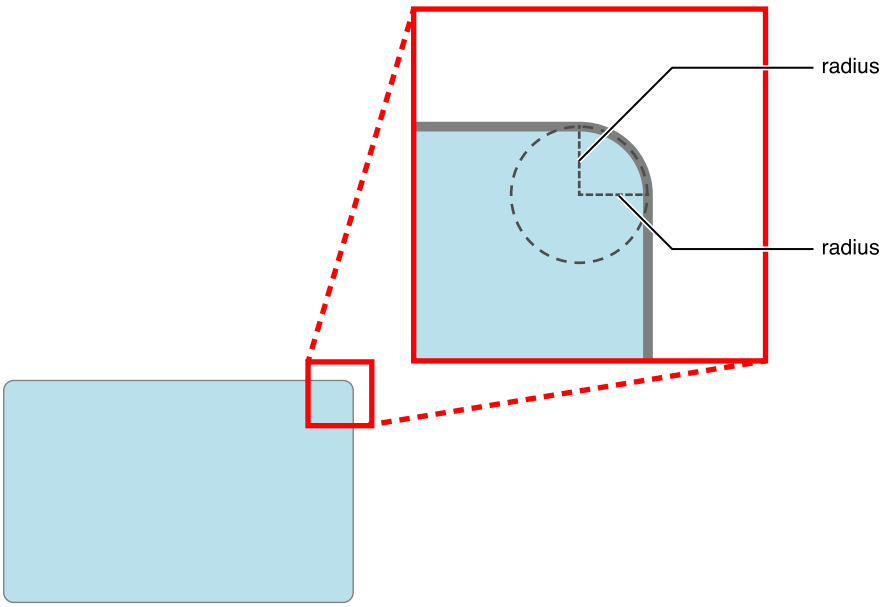
Note: You can use any type of color for the background of a layer, including colors that have transparency or use a pattern image. When using pattern images, though, be aware that Core Graphics handles the rendering of the pattern image and does so using its standard coordinate system, which is different than the default coordinate system in iOS. As such, images rendered on iOS appear upside down by default unless you flip the coordinates.

If you set your layer's background color to an opaque color, consider setting the layer's opaque property to `YES`. Doing so can improve performance when compositing the layer onscreen and eliminates the need for the layer's backing store to manage an alpha channel. You must not mark a layer as opaque if it also has a nonzero corner radius, though.

Layers Support a Corner Radius

You can create a rounded rectangle effect for your layer by adding a corner radius to it. A corner radius is a visual adornment that masks part of the corners of the layer's bounds rectangle to allow the underlying content to show through, as shown in Figure 2-4. Because it involves applying a transparency mask, the corner radius does not affect the image in the layer's `contents` property unless the `masksToBounds` property is set to `YES`. However, the corner radius always affects how the layer's background color and border are drawn.

Figure 2-4 A corner radius on a layer



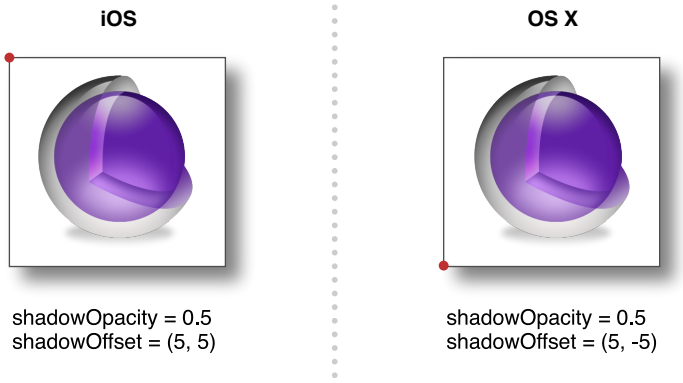
To apply a corner radius to your layer, specify a value for the `cornerRadius` property of the layer. The radius value you specify is measured in points and applied to all four corners of the layer prior to display.

Layers Support Built-In Shadows

The `CALayer` class includes several properties for configuring a shadow effect. A shadow adds depth to the layer by making it appear as if it is floating above its underlying content. This is another type of visual adornment that you might find useful in specific situations for your app. With layers, you can control the shadow's color, placement relative to the layer's content, opacity, and shape.

The opacity value for layer shadows is set to `0` by default, which effectively hides the shadow. Changing the opacity to a nonzero value causes Core Animation to draw the shadow. Because shadows are positioned directly under the layer by default, you might also need to change the shadow's offset before you can see it. It is important to remember, though, that the offsets you specify for the shadow are applied using the layer's native coordinate system, which is different on iOS and OS X. Figure 2-5 shows a layer with a shadow that extends down and to the right of the layer. In iOS, this requires specifying a positive value for the y axis but in OS X the value needs to be negative.

Figure 2-5 Applying a shadow to a layer



When adding shadows to a layer, the shadow is part of the layer's content but actually extends outside the layer's bounds rectangle. As a result, if you enable the `masksToBounds` property for the layer, the shadow effect is clipped around the edges. If your layer contains any transparent content, this can cause an odd effect where the portion of the shadow directly under your layer is still visible but the part extending beyond your layer is not. If you want a shadow but also want to use bounds masking, you use two layers instead of one. Apply the mask to the layer containing your content and then embed that layer inside a second layer of the exact same size that has the shadow effect enabled.

For examples of how shadows are applied to layers, see Shadow Properties (page 82).

Filters Add Visual Effects to OS X Views

In OS X apps, you can apply Core Image filters directly to the contents of your layers. You might do this to blur or sharpen your layer's contents, to change the colors, to distort the content, or to perform many other types of operations. For example, an image processing program might use these filters to modify an image nondestructively while a video editing program might use them to implement different types of video transition effects. And because the filters are applied to the layer's content in hardware, rendering is fast and smooth.

> **Note:** You cannot add filters to layer objects in iOS.

For a given layer, you can apply filters to both the foreground and background content of the layer. The foreground content consists of everything that the layer itself contains, including the image in its `contents` property, its background color, its border, and the content of its sublayers. The background content is the content that is directly under the layer but not actually part of the layer itself. The background content of most layers is the content of its immediate superlayer, which may be wholly or partially obscured by the layer. For example, you might apply a blur filter to the background content when you want the user to focus on the layer's foreground content.

You specify filters by adding `CIFilter` objects to the following properties of your layer:

- The `filters` property contains an array of filters that affect the layer's foreground content only.
- The `backgroundFilters` property contains an array of filters that affect the layer's background content only.
- The `compositingFilter` property defines how the layer's foreground and background content are composited together.

To add a filter to a layer, you must first locate and create the `CIFilter` object and then configure it before adding it to your layer. The `CIFilter` class includes several class methods for locating the available Core Image filters, such as the `filterWithName:` method. Creating the filter is only the first step, though. Many filters have parameters that define how the filter modifies an image. For example, a box blur filter has an input radius parameter that affects the amount of blur that is applied. You should always provide values for these parameters as part of the filter configuration process. However, one common parameter that you do not need to specify is the input image, which is provided by the layer itself.

When adding filters to layers, it is best to configure the filter parameters prior to adding the filter to the layer. The main reason for doing so is that once added to the layer, you cannot modify the `CIFilter` object itself. However, you can use the layer's `setValue:forKeyPath:` method to change filter values after the fact.

Listing 2-6 shows how to create and apply a pinch distortion filter to a layer object. This filter pinches the source pixels of the layer inward, distorting those pixels closest to the specified center point the most. Notice in the example that you do not need to specify the input image for the filter because the layer's image is used automatically.

Listing 2-6 Applying a filter to a layer

```
CIFilter* aFilter = [CIFilter filterWithName:@"CIPinchDistortion"];
[aFilter setValue:[NSNumber numberWithFloat:500.0] forKey:@"inputRadius"];
[aFilter setValue:[NSNumber numberWithFloat:1.25] forKey:@"inputScale"];
[aFilter setValue:[CIVector vectorWithX:250.0 Y:150.0] forKey:@"inputCenter"];

myLayer.filters = [NSArray arrayWithObject:aFilter];
```

For information about the available Core Image filters, see *Core Image Filter Reference*.

The Layer Redraw Policy for OS X Views Affects Performance

In OS X, layer-backed views support several different policies for determining when to update the underlying layer's contents. Because there are differences between the native AppKit drawing model and the one introduced by Core Animation, these policies make it easier to migrate your older code over to Core Animation. You can configure these policies on a view-by-view basis to ensure the best performance for each of your views.

Each view defines a `layerContentsRedrawPolicy` method that returns the redraw policy for the view's layer. You set the policy using the `setLayerContentsRedrawPolicy:` method. To preserve compatibility with its traditional drawing model, AppKit sets the redraw policy to `NSViewLayerContentsRedrawDuringViewResize` by default. However, you can change the policy to any of the values in Table 2-2. Notice that the recommended redraw policy is not the default policy.

Table 2-2 Layer redraw policies for OS X views

Policy	Usage
`NSViewLayerContentsRedrawOnSetNeedsDisplay`	This is the recommended policy. With this policy, view geometry changes do not automatically cause the view to update its layer's contents. Instead, the layer's existing contents are stretched and manipulated to facilitate the geometry changes. To force the view to redraw itself and update the layer's contents, you must explicitly call the view's `setNeedsDisplay:` method. This policy most closely represents the standard behavior for Core Animation layers. However, it is not the default policy and must be set explicitly.
`NSViewLayerContentsRedrawDuringViewResize`	This is the default redraw policy. This policy maintains maximum compatibility with traditional AppKit drawing by recaching the layer's contents whenever the view's geometry changes. This behavior results in the view's `drawRect:` method being called multiple times on your app's main thread during the resize operation.
`NSViewLayerContentsRedrawBeforeViewResize`	With this policy, AppKit draws the layer at its final size prior to any resize operations and caches that bitmap. The resize operation uses the cached bitmap as the starting image, scaling it to fit the old bounds rectangle. It then animates the bitmap to its final size. This behavior can cause the view's contents to appear stretched or distorted at the beginning of an animation and is better in situations where the initial appearance is not important or not noticeable.
`NSViewLayerContentsRedrawNever`	With this policy, AppKit does not update the layer at all, even when you call the `setNeedsDisplay:` method. This policy is most appropriate for views whose contents never change and where the size of the view changes infrequently if at all. For example, you might use this for views that display fixed-size content or background elements.

View redraw policies alleviate the need to use standalone sublayers to improve drawing performance. Prior to the introduction of view redraw policies, there were some layer-backed views that drew more frequently than was needed and thereby caused performance issues. The solution to these performance issues was to use sublayers to present those portions of the view's content that did not require regular redrawing. With the introduction of redraw policies in OS X v10.6, it is now recommended that you set a layer-backed view's redraw policy to an appropriate value, rather than creating explicit sublayer hierarchies.

Adding Custom Properties to a Layer

The `CAAnimation` and `CALayer` classes extend the key-value coding conventions to support custom properties. You can use this behavior to add data to a layer and retrieve it using a custom key you define. You can even associate actions with your custom properties so that when you change the property, a corresponding animation is performed.

For information about how to set and get custom properties, see Key-Value Coding Compliant Container Classes (page 90). For information about adding actions to your layer objects, see Changing a Layer's Default Behavior (page 70).

Printing the Contents of a Layer-Backed View

During printing, layers redraw their contents as needed to accommodate the printing environment. Whereas Core Animation normally relies on cached bitmaps when rendering to the screen, it redraws that content when printing. In particular, if a layer-backed view uses the `drawRect:` method to provide the layer contents, Core Animation calls `drawRect:` again during printing to generate the printed layer contents.

Animating Layer Content

SwiftObjective-C

The infrastructure provided by Core Animation makes it easy to create sophisticated animations of your app's layers, and by extension to any views that own those layers. Examples include changing the size of a layer's frame rectangle, changing its position onscreen, applying a rotation transform, or changing its opacity. With Core Animation, initiating an animation is often as simple as just changing the property but you can also create animations and set the animation parameters explicitly.

For information about creating more advanced animations, see Advanced Animation Tricks (page 63).

Animating Simple Changes to a Layer's Properties

You can perform simple animations implicitly or explicitly depending on your needs. Implicit animations use the default timing and animation properties to perform an animation, whereas explicit animations require you to configure those properties yourself using an animation object. So implicit animations are perfect for situations where you want to make a change without a lot of code and the default timing works well for you.

Simple animations involve changing the properties of a layer and letting Core Animation animate those changes over time. Layers define many properties that affect the visible appearance of the layer. Changing one of these properties is a way to animate the appearance change. For example, changing the opacity of the layer from `1.0` to `0.0` causes the layer to fade out and become transparent.

> **Important:** Although you can sometimes animate layer-backed views directly using Core Animation interfaces, doing so often requires extra steps. For more information about how how to use Core Animation in conjunction with layer-backed views, see How to Animate Layer-Backed Views (page 50).

To trigger implicit animations, all you have to do is update the properties of your layer object. When modifying layer objects in the layer tree, your changes are reflected immediately by those objects. However, the visual appearance of the layer objects does not change immediately. What happens instead is that Core Animation uses your changes as a trigger to create and schedule one or more implicit animations for execution. Thus, making a change like the one in Listing 3-1 causes Core Animation to create an animation object for you and schedule that animation to run starting in the next update cycle.

Listing 3-1 Animating a change implicitly

```
theLayer.opacity = 0.0;
```

To make the same change explicitly using an animation object, create a `CABasicAnimation` object and use that object to configure the animation parameters. You can set the start and end values for the animation, change the duration, or change any other animation parameters before adding the animation to a layer. Listing 3-2 shows how to fade out a layer using an animation object. When creating the object, you specify the key path for the property you want to animate and then set your animation parameters. To execute the animation, you use the `addAnimation:forKey:` method to add it to the layers you want to animate.

Listing 3-2 Animating a change explicitly

```
CABasicAnimation* fadeAnim = [CABasicAnimation animationWithKeyPath:@"opacity"];
fadeAnim.fromValue = [NSNumber numberWithFloat:1.0];
fadeAnim.toValue = [NSNumber numberWithFloat:0.0];
fadeAnim.duration = 1.0;
[theLayer addAnimation:fadeAnim forKey:@"opacity"];

// Change the actual data value in the layer to the final value.
theLayer.opacity = 0.0;
```

> **Tip:** When creating an explicit animation, it is recommended that you always assign a value to the `fromValue` property of the animation object. If you do not specify a value for this property, Core animation uses the layer's current value as the starting value. If you already updated the property to its final value, that might not yield the results you want.

Unlike an implicit animation, which updates the layer object's data value, an explicit animation does not modify the data in the layer tree. Explicit animations only produce the animations. At the end of the animation, Core Animation removes the animation object from the layer and redraws the layer using its current data values. If you want the changes from an explicit animation to be permanent, you must also update the layer's property as shown in the preceding example.

Implicit and explicit animations normally begin executing after the current run loop cycle ends, and the current thread must have a run loop in order for animations to be executed. If you change multiple properties, or if you add multiple animation objects to a layer, all of those property changes are animated at the same time.

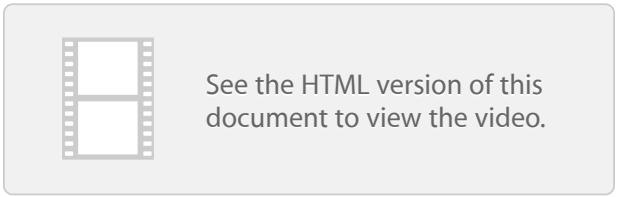
For example, you can fade a layer while moving it offscreen by configuring both animations at the same time. However, you can also configure animation objects to start at a particular time. For more information about modifying animation timing, see Customizing the Timing of an Animation (page 65).

Using a Keyframe Animation to Change Layer Properties

Whereas a property-based animation changes a property from a start value to an end value, a `CAKeyframeAnimation` object lets you animate through a set of target values in a way that might or might not be linear. A key frame animation consists of a set of target data values and the times at which each value should be reached. In the simplest configuration, you specify both the values and times using an array. For changes to a layer's position, you can also have the changes follow a path. The animation object takes the key frames you specify and builds the animation by interpolating from one value to the next over the given time periods.

Figure 3-1 shows a 5-second animation of a layer's `position` property. The position is animated to follow a path, which was specified using a `CGPathRef` data type. The code for this animation is shown in Listing 3-3 (page 45).

Figure 3-1 5-second keyframe animation of a layer's position property



Listing 3-3 shows the code used to implement the animation in Figure 3-1. The path object in this example is used to define the position of the layer for each frame of the animation.

Listing 3-3 Creating a bounce keyframe animation

```
// create a CGPath that implements two arcs (a bounce)
CGMutablePathRef thePath = CGPathCreateMutable();
CGPathMoveToPoint(thePath,NULL,74.0,74.0);
CGPathAddCurveToPoint(thePath,NULL,74.0,500.0,
                                   320.0,500.0,
                                   320.0,74.0);
CGPathAddCurveToPoint(thePath,NULL,320.0,500.0,
                                   566.0,500.0,
```

```
                                    566.0,74.0);

CAKeyframeAnimation * theAnimation;

// Create the animation object, specifying the position property as the key path.
theAnimation=[CAKeyframeAnimation animationWithKeyPath:@"position"];
theAnimation.path=thePath;
theAnimation.duration=5.0;

// Add the animation to the layer.
[theLayer addAnimation:theAnimation forKey:@"position"];
```

Specifying Keyframe Values

The key frame values are the most important part of a keyframe animation. These values define the behavior of the animation over the course of its execution. The main way to specify keyframe values is as an array of objects but for values that contain a `CGPoint` data type (such as the layer's `anchorPoint` and `position` properties), you can specify a `CGPathRef` data type instead.

When specifying an array of values, what you put into the array depends on the data type required by the property. You can add some objects to an array directly; however, some objects must be cast to `id` before being added, and all scalar types or structs must be wrapped by an object. For example:

- For properties that take a `CGRect` (such as the bounds and frame properties), wrap each rectangle in an `NSValue` object.
- For the layer's transform property, wrap each `CATransform3D` matrix in an `NSValue` object. Animating this property causes the keyframe animation to apply each transform matrix to the layer in turn.
- For the `borderColor` property, cast each `CGColorRef` data type to the type `id` before adding it to the array.
- For properties that take a `CGFloat` value, wrap each value in an `NSNumber` object before adding it to the array.
- When animating the layer's `contents` property, specify an array of `CGImageRef` data types.

For properties that take a `CGPoint` data type, you can create an array of points (wrapped in `NSValue` objects) or you can use a `CGPathRef` object to specify the path to follow. When you specify an array of points, the keyframe animation object draws a straight line between each successive point and follows that path. When you specify a `CGPathRef` object, the animation starts at the beginning point of the path and follows its outline, including along any curved surfaces. You can use either an open or closed path.

Specifying the Timing of a Keyframe Animation

The timing and pacing of keyframe animations is more complex than those of basic animations and there are several properties you can use to control it:

- The `calculationMode` property defines the algorithm to use in calculating the animation timing. The value of this property affects how the other timing-related properties are used.
    - Linear and cubic animations—that is, animations where the `calculationMode` property is set to `kCAAnimationLinear` or `kCAAnimationCubic`—use the provided timing information to generate the animation. These modes give you the maximum control over the animation timing.
    - Paced animations—that is, animations where the `calculationMode` property is set to `kCAAnimationPaced` or `kCAAnimationCubicPaced`—do not rely on the external timing values provided by the `keyTimes` or `timingFunctions` properties. Instead, timing values are calculated implicitly to provide the animation with a constant velocity.
    - Discrete animations—that is, animations where the `calculationMode` property is set to `kCAAnimationDiscrete`—cause the animated property to jump from one keyframe value to the next without any interpolation. This calculation mode uses the values in the `keyTimes` property but ignores the `timingFunctions` property
- The `keyTimes` property specifies time markers at which to apply each keyframe value. This property is used only if the calculation mode is set to `kCAAnimationLinear`, `kCAAnimationDiscrete`, or `kCAAnimationCubic`. It is not used for paced animations.
- The `timingFunctions` property specifies the timing curves to use for each keyframe segment. (This property replaces the inherited `timingFunction` property.)

If you want to handle the animation timing yourself, use the `kCAAnimationLinear` or `kCAAnimationCubic` mode and the `keyTimes` and `timingFunctions` properties. The `keyTimes` defines the points in time at which to apply each keyframe value. The timing for all intermediate values is controlled by the timing functions, which allow you to apply ease-in or ease-out curves to each segment. If you do not specify any timing functions, the timing is linear.

Stopping an Explicit Animation While It Is Running

Animations normally run until they are complete, but you can stop them early if needed using one of the following techniques:

- To remove a single animation object from the layer, call the layer's `removeAnimationForKey:` method to remove your animation object. This method uses the key that was passed to the `addAnimation:forKey:` method to identify the animation. The key you specify must not be `nil`.
- To remove all animation objects from the layer, call the layer's `removeAllAnimations` method. This method removes all ongoing animations immediately and redraws the layer using its current state information.

> **Note:** You cannot remove implicit animations from a layer directly.

When you remove an animation from a layer, Core Animation responds by redrawing the layer using its current values. Because the current values are usually the end values of the animation, this can cause the appearance of the layer to jump suddenly. If you want the layer's appearance to remain where it was on the last frame of the animation, you can use the objects in the presentation tree to retrieve those final values and set them on the objects in the layer tree.

For information about pausing an animation temporarily, see Listing 5-4 (page 66).

Animating Multiple Changes Together

If you want to apply multiple animations to a layer object simultaneously, you can group them together using a `CAAnimationGroup` object. Using a group object simplifies the management of multiple animation objects by providing a single configuration point. Timing and duration values applied to the group override those same values in the individual animation objects.

Listing 3-4 shows how you would use an animation group to perform two border-related animations at the same time and with the same duration.

Listing 3-4 Animating two animations together

```
// Animation 1

CAKeyframeAnimation* widthAnim = [CAKeyframeAnimation
animationWithKeyPath:@"borderWidth"];

NSArray* widthValues = [NSArray arrayWithObjects:@1.0, @10.0, @5.0, @30.0, @0.5,
@15.0, @2.0, @50.0, @0.0, nil];
```

```
widthAnim.values = widthValues;
widthAnim.calculationMode = kCAAnimationPaced;

// Animation 2
CAKeyframeAnimation* colorAnim = [CAKeyframeAnimation
animationWithKeyPath:@"borderColor"];
NSArray* colorValues = [NSArray arrayWithObjects:(id)[UIColor greenColor].CGColor,
          (id)[UIColor redColor].CGColor, (id)[UIColor blueColor].CGColor,  nil];
colorAnim.values = colorValues;
colorAnim.calculationMode = kCAAnimationPaced;

// Animation group
CAAnimationGroup* group = [CAAnimationGroup animation];
group.animations = [NSArray arrayWithObjects:colorAnim, widthAnim, nil];
group.duration = 5.0;

[myLayer addAnimation:group forKey:@"BorderChanges"];
```

A more advanced way to group animations together is to use a transaction object. Transactions provide more flexibility by allowing you to create nested sets of animations and assign different animation parameters for each. For information about how to use transaction objects, see Explicit Transactions Let You Change Animation Parameters (page 67).

Detecting the End of an Animation

Core Animation provides support for detecting when an animation begins or ends. These notifications are a good time to do any housekeeping tasks associated with the animation. For example, you might use a start notification to set up some related state information and use the corresponding end notification to tear down that state.

There are two different ways to be notified about the state of an animation:

- Add a completion block to the current transaction using the `setCompletionBlock:` method. When all of the animations in the transaction finish, the transaction executes your completion block.
- Assign a delegate to your `CAAnimation` object and implement the `animationDidStart:` and `animationDidStop:finished:` delegate methods.

If you want to chain two animations together so that one starts when the other finishes, do not use animation notifications. Instead, use the `beginTime` property of your animation objects to start each one at the desired time. To chain two animations together, set the start time of the second animation to the end time of the first animation. For more information about animation and timing values, see Customizing the Timing of an Animation (page 65).

How to Animate Layer-Backed Views

If a layer belongs to a layer-backed view, the recommended way to create animations is to use the view-based animation interfaces provided by UIKit or AppKit. There are ways to animate the layer directly using Core Animation interfaces but how you create those animations depends on the target platform.

Rules for Modifying Layers in iOS

Because iOS views always have an underlying layer, the `UIView` class itself derives most of its data from the layer object directly. As a result, changes you make to the layer are automatically reflected by the view object as well. This behavior means that you can use either the Core Animation or `UIView` interfaces to make your changes.

If you want to use Core Animation classes to initiate animations, you must issue all of your Core Animation calls from inside a view-based animation block. The `UIView` class disables layer animations by default but reenables them inside animation blocks. So any changes you make outside of an animation block are not animated. Listing 3-5 shows an example of how to change a layer's opacity implicitly and its position explicitly. In this example, the `myNewPosition` variable is calculated beforehand and captured by the block. Both animations start at the same time but the opacity animation runs with the default timing while the position animation runs with the timing specified in its animation object.

Listing 3-5 Animating a layer attached to an iOS view

```
[UIView animateWithDuration:1.0 animations:^{
   // Change the opacity implicitly.
   myView.layer.opacity = 0.0;

   // Change the position explicitly.
   CABasicAnimation* theAnim = [CABasicAnimation animationWithKeyPath:@"position"];
   theAnim.fromValue = [NSValue valueWithCGPoint:myView.layer.position];
   theAnim.toValue = [NSValue valueWithCGPoint:myNewPosition];
   theAnim.duration = 3.0;
```

```
    [myView.layer addAnimation:theAnim forKey:@"AnimateFrame"];
}];
```

Rules for Modifying Layers in OS X

To animate changes to a layer-backed view in OS X, it is best to use the interfaces of the view itself. You should rarely, if ever, directly modify the layer that is attached to one of your layer-backed `NSView` objects. AppKit is responsible for creating and configuring those layer objects and for managing them while your app is running. Modifying the layer could cause it to get out of sync with the view object and could lead to unexpected results. For layer-backed views, your code must absolutely *not* modify any the following properties of the layer object:

- `anchorPoint`
- `bounds`
- `compositingFilter`
- `filters`
- `frame`
- `geometryFlipped`
- `hidden`
- `position`
- `shadowColor`
- `shadowOffset`
- `shadowOpacity`
- `shadowRadius`
- `transform`

Important: The preceding restrictions do not apply to layer-hosting views. If you created the layer object and associated it with a view manually, you are responsible for modifying the properties of that layer and keeping the corresponding view object in sync.

AppKit disables implicit animations for its layer-backed views by default. The view's animator proxy object reenables implicit animations automatically for you. If you want to animate layer properties directly, you can also programmatically reenable implicit animations by changing the `allowsImplicitAnimation` property of the current `NSAnimationContext` object to `YES`. Again, you should do this only for animatable properties that are not in the preceding list.

Remember to Update View Constraints as Part of Your Animation

If you are using constraint-based layout rules to manage the position of your views, you must remove any constraints that might interfere with an animation as part of configuring that animation. Constraints affect any changes you make to the position or size of a view. They also affect the relationships between the view and its child views. If you are animating changes to any of those items, you can remove the constraints, make the change, and then apply whatever new constraints are needed.

For more information on constraints and how you use them to manage the layout of your views, see *Auto Layout Guide*.

Building a Layer Hierarchy

Objective-CSwift

Most of the time, the best way to use layers in your app is to use them in conjunction with a view object. However, there are times when you might need to enhance your view hierarchy by adding additional layer objects to it. You might use use layers when doing so offers better performance or lets you implement a feature that would be difficult to do with views alone. In those situations, you need to know how to manage the layer hierarchies you create.

> **Important:** In OS X v10.8 and later, it is recommended that you minimize your use of layer hierarchies and just use layer-backed views. The layer redraw policies introduced in that version of OS X let you customize the behavior of your layer-backed views and still get the kind of performance you might have gotten previously using standalone layers.

Arranging Layers into a Layer Hierarchy

Layer hierarchies are similar in many ways to view hierarchies. You embed one layer inside another to create a parent-child relationship between the layer being embedded (known as the *sublayer*) and the parent layer (known as the *superlayer*). This parent-child relationship affects various aspects of the sublayer. For example, its content sits above the content of its parent, its position is specified relative to the coordinate system of its parent, and it is affected by any transforms applied to the parent.

Adding, Inserting, and Removing Sublayers

Every layer object has methods for adding, inserting, and removing sublayers. Table 4-1 summarizes these methods and their behavior.

Table 4-1 Methods for modifying the layer hierarchy

Behavior	Methods	Description
Adding layers	`addSublayer:`	Adds a new sublayer object to the current layer. The sublayer is added to the end of the layer's list of sublayers. This causes the sublayer to appear on top of any siblings with the same value in their `zPosition` property.

Behavior	Methods	Description

Inserting layers	`insertSublayer:above:`
	`insertSublayer:atIndex:`
	`insertSublayer:below:`
		Inserts the sublayer into the sublayer hierarchy at the specified index or at a position relative to another sublayer. When inserting above or below another sublayer, you are only specifying the sublayer's position in the `sublayers` array. The actual visibility of the layers is determined primarily by the value in their `zPosition` property and secondarily by their position in the `sublayers` array.

Removing layers	`removeFromSuperlayer`	Removes the sublayer from its parent layer.
Exchanging layers	`replaceSublayer:with:`	Exchanges one sublayer for another. If the sublayer you are inserting is already in another layer hierarchy, it is removed from that hierarchy first.

You use the preceding methods when working with layer objects you created yourself. You would not use these methods to arrange layers that belong to layer-backed views. However, a layer-backed view can act as the parent for standalone layers you create yourself.

Positioning and Sizing Sublayers

When adding and inserting sublayers, you must set the size and position of the sublayer before it appears onscreen. You can modify the size and position of a sublayer after adding it to your layer hierarchy but should get in the habit of setting those values when you create the layer.

You set the size of a sublayer using the `bounds` property and set its position within its superlayer using the `position` property. The origin of the bounds rectangle is almost always (0, 0) and the size is whatever size you want for the layer specified in points. The value in the `position` property is interpreted relative to the layer's anchor point, which is located in the center of the layer by default. If you do not assign values to these properties, Core Animation sets the initial width and height of the layer to 0 and sets the position to (0, 0).

```
myLayer.bounds = CGRectMake(0, 0, 100, 100);
myLayer.position = CGPointMake(200, 200);
```

Important: Always use integral numbers for the width and height of your layer.

How Layer Hierarchies Affect Animations

Some superlayer properties can affect the behavior of any animations applied to its child layers. One such property is the `speed` property, which is a multiplier for the speed of the animation. The value of this property is set to `1.0` by default but changing it to `2.0` causes animations to run at twice their original speed and thereby finish in half the time. This property affects not only the layer for which it is set but also for that layer's sublayers. Such changes are multiplicative too. If both a sublayer and its superlayer have a speed of `2.0`, animations on the sublayer run at four times their original speed.

Most other layer changes affect any contained sublayers in predictable ways. For example, applying a rotation transform to a layer rotates that layer and all of its sublayers. Similarly, changing a layer's opacity changes the opacity of its sublayers. Changes to the size of a layer follow the rules for layout that are described in Adjusting the Layout of Your Layer Hierarchies (page 55).

Adjusting the Layout of Your Layer Hierarchies

Core Animation supports several options for adjusting the size and position of sublayers in response to changes to their superlayer. In iOS, the pervasive use of layer-backed views makes the creation of layer hierarchies less important; only manual layout updates are supported. For OS X, several other options are available that make it easier to manage your layer hierarchies.

Layer-level layout is only relevant if you are building layer hierarchies using standalone layer objects you created. If your app's layers are all associated with views, use the view-based layout support to update the size and position of your views in response to changes.

Using Constraints to Manage Your Layer Hierarchies in OS X

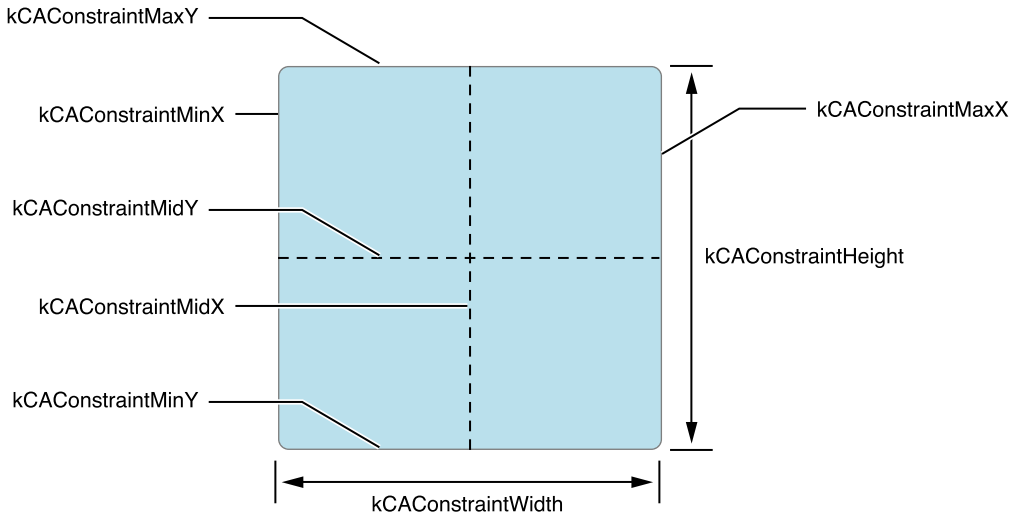
Constraints let you specify the position and size of a layer using a set of detailed relationships between the layer and its superlayer or sibling layers. Defining constraints requires the following steps:

1. Create one or more `CAConstraint` objects. Use those objects to define the constraint parameters.
2. Add your constraint objects to the layer whose attributes they modify.
3. Retrieve the shared `CAConstraintLayoutManager` object and assign to the immediate superlayer.

Figure 4-1 shows the attributes that you can use to define a constraint and the aspect of the layer that they impact. You can use constraints to change the position the layer based on the position of its edges of midpoints relative to another layer. You can also use them to change the size of the layer. The changes you make can be

proportional to the superlayer or relative to another layer. You can even add a scaling factor or constant to the resulting change. This extra flexibility makes it possible to control a layer's size and position very precisely using a simple set of rules.

Figure 4-1 Constraint layout manager attributes



Each constraint object encapsulates one geometry relationship between two layers along the same axis. A maximum of two constraint objects may be assigned to each axis and it is those two constraints that determine which attribute is changeable. For example, if you specify constraints for the left and right edge of the layer, the size of the layer changes. If you specify constraints for the left edge and width of the layer, the location of the layer's right edge changes. If you specify a single constraint for one of the layer's edges, Core Animation creates an implicit constraint that keeps the size of the layer fixed in the given dimension.

When creating constraints, you must always specify three pieces of information:

- The aspect of the layer that you want to constrain
- The layer to use as a reference
- The aspect of the reference layer to use in the comparison

Listing 4-1 shows a simple constraint that pins the vertical midpoint of a layer to the vertical midpoint of its superlayer. When referring to the superlayer, use the string `superlayer`. This string is a special name reserved for referring to the superlayer. Using it eliminates needing to have a pointer to the layer or know the layer's name. It also allows you to change the superlayer and have the constraint apply automatically to the new parent. (When creating constraints relative to sibling layers, you must identify the sibling layer using its `name` property.)

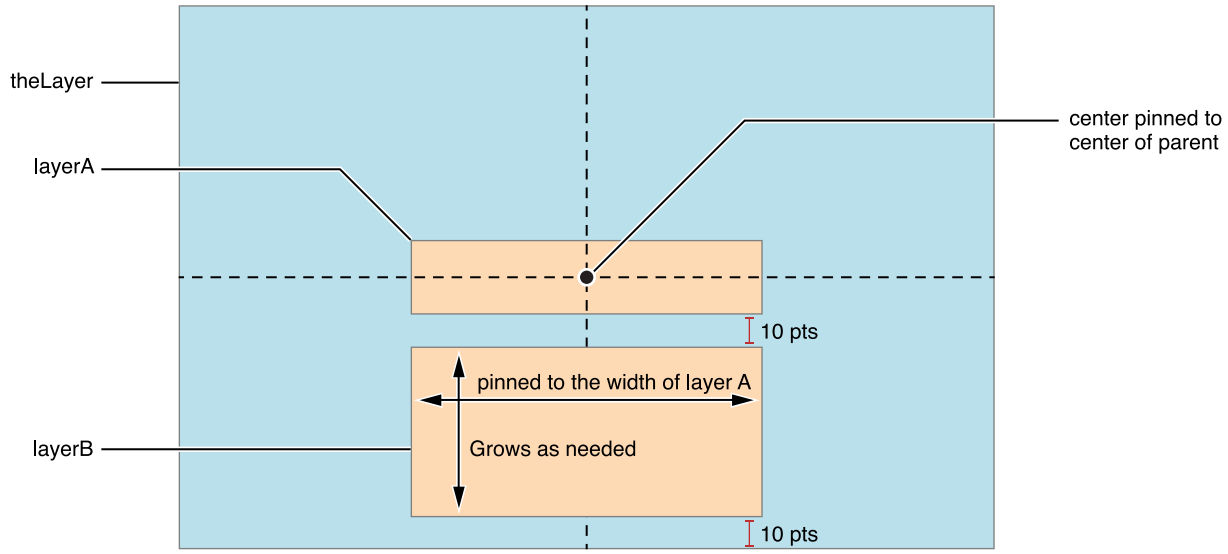
Listing 4-1 Defining a simple constraint

```
[myLayer addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidY
                                                  relativeTo:@"superlayer"
                                                   attribute:kCAConstraintMidY]];
```

To apply constraints at runtime, you must attach the shared `CAConstraintLayoutManager` object to the immediate superlayer. Each layer is responsible for managing the layout of its sublayers. Assigning the layout manager to the parent tells Core Animation to apply the constraints defined by its children. The layout manager object applies the constraints automatically. After assigning it to the parent layer, you do not have to tell it to update the layout.

To see how constraints work in a more specific scenario, consider Figure 4-2. In this example, the design requires that the width and height of `layerA` remain unchanged and that `layerA` remain centered inside its superlayer. In addition, the width of `layerB` must match that of `layerA`, the top edge of `layerB` must remain 10 points below the bottom edge of `layerA`, and the bottom edge of `layerB` must remain 10 points above the bottom edge of the superlayer. Listing 4-2 (page 57) shows the code that you would use to create the sublayers and constraints for this example.

Figure 4-2 Example constraints based layout



Listing 4-2 Setting up constraints for your layers

```
// Create and set a constraint layout manager for the parent layer.
theLayer.layoutManager=[CAConstraintLayoutManager layoutManager];
```

```
// Create the first sublayer.
CALayer *layerA = [CALayer layer];
layerA.name = @"layerA";
layerA.bounds = CGRectMake(0.0,0.0,100.0,25.0);
layerA.borderWidth = 2.0;

// Keep layerA centered by pinning its midpoint to its parent's midpoint.
[layerA addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidY
                                                 relativeTo:@"superlayer"
                                                  attribute:kCAConstraintMidY]];
[layerA addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                                 relativeTo:@"superlayer"
                                                  attribute:kCAConstraintMidX]];
[theLayer addSublayer:layerA];

// Create the second sublayer
CALayer *layerB = [CALayer layer];
layerB.name = @"layerB";
layerB.borderWidth = 2.0;

// Make the width of layerB match the width of layerA.
[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintWidth
                                                 relativeTo:@"layerA"
                                                  attribute:kCAConstraintWidth]];

// Make the horizontal midpoint of layerB match that of layerA
[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                                 relativeTo:@"layerA"
                                                  attribute:kCAConstraintMidX]];

// Position the top edge of layerB 10 points from the bottom edge of layerA.
[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMaxY
                                                 relativeTo:@"layerA"
                                                  attribute:kCAConstraintMinY
```

```
                                                    offset:-10.0]];

// Position the bottom edge of layerB 10 points
//  from the bottom edge of the parent layer.
[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMinY
                                                 relativeTo:@"superlayer"
                                                  attribute:kCAConstraintMinY
                                                    offset:+10.0]];

[theLayer addSublayer:layerB];
```

One interesting thing to note about Listing 4-2 (page 57) is that the code never sets the size of `layerB` explicitly. Because of the defined constraints, the width and height of `layerB` are set automatically every time the layout is updated. Therefore, setting the size using the bounds rectangle is unnecessary.

Warning: When creating constraints, do not create circular references among your constraints. Circular constraints make it impossible to calculate the needed layout information. When such circular references are encountered, the layout behavior is undefined.

Setting Up Autoresizing Rules for Your OS X Layer Hierarchies

Autoresizing rules are another way of adjusting the size and position of a layer in OS X. With autoresizing rules, you designate whether the edges of your layer should remain at a fixed or variable distance from the corresponding edges of the superlayer. You can similarly designate whether the width or height of your layer is fixed or variable. The relationships are always between the layer and its superlayer. You cannot use autoresizing rules to specify relationships between sibling layers.

To set up the autosizing rules for a layer, you must assign the appropriate constants to the `autoresizingMask` property of the layer. By default, layers are configured to have a fixed width and height. During layout, the precise size and position of the layer is computed for you automatically by Core Animation and involve a complex set of calculations based on many factors. Core Animation applies the autoresizing behaviors before it asks your delegate to do any manual layout updates, so you can use the delegate to tweak the results of the autoresizing layout as needed.

Manually Laying Out Your Layer Hierarchies

On iOS and OS X, you can handle layout manually by implementing the `layoutSublayersOfLayer:` method on the delegate object of the superlayer. You use that method to adjust the size and position of any sublayers currently embedded inside the layer. When doing manual layout updates, it is up to you to perform the necessary calculations to position each sublayer.

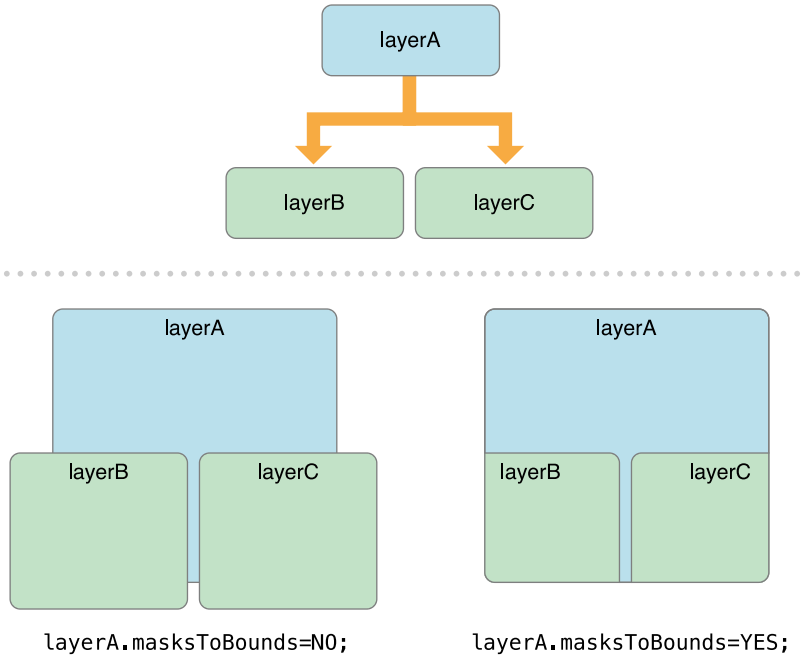
If you are implementing a custom layer subclass, your subclass can override the `layoutSublayers` method and use that method (instead of a delegate) to handle any layout tasks. You should only override this method in cases where you need complete control over the positioning of sublayers inside your custom layer class. Replacing the default implementation prevents Core Animation from applying constraints or autoresizing rules on OS X.

Sublayers and Clipping

Unlike views, a superlayer does not automatically clip the contents of sublayers that lie outside its bounds rectangle. Instead, the superlayer allows its sublayers to be displayed in their entirety by default. However, you can reenable clipping by setting the `masksToBounds` property of the layer to `YES`.

The shape of a layer's clipping mask includes the layer's corner radius, if one is specified. Figure 4-3 shows a layer that demonstrates how the `masksToBounds` property affects a layer with rounded corners. When the property is set to `NO`, sublayers are displayed in their entirety, even if they extend beyond the bounds of their parent layer. Changing the property to `YES` causes their content to be clipped.

Figure 4-3 Clipping sublayers to the parent's bounds



Converting Coordinate Values Between Layers

Occasionally, you might need to convert a coordinate value in one layer to a coordinate value at the same screen location in a different layer. The `CALayer` class provides a set of simple conversion routines that you can use for this purpose:

- `convertPoint:fromLayer:`
- `convertPoint:toLayer:`
- `convertRect:fromLayer:`
- `convertRect:toLayer:`

In addition to converting point and rectangle values, you can also convert time values between layers using the `convertTime:fromLayer:` and `convertTime:toLayer:` methods. Each layer defines its own local time space and uses that time space to synchronize the beginning and ending of animations with the rest of

the system. These time spaces are synchronized by default; however, if you change the animation speed for one set of layers, the time space for those layers changes accordingly. You can use the time conversion methods to to account for any such factors and ensure that the timing of both layers is synchronized.

Advanced Animation Tricks

There are many ways to configure your property-based or keyframe animations to do more for you. Apps that need to perform multiple animations together or sequentially can use more advanced behaviors to synchronize the timing of those animations or chain them together. You can also use other types of animation objects to create visual transitions and other interesting animated effects.

Transition Animations Support Changes to Layer Visibility

As the name implies, a transition animation object creates an animated visual transition for a layer. The most common use for transition objects is to animate the appearance of one layer and the disappearance of another in a coordinated manner. Unlike a property-based animation, where the animation changes one property of a layer, a transition animation manipulates a layer's cached image to create visual effects that would be difficult or impossible to do by changing properties alone. The standard types of transitions let you perform reveal, push, move, or crossfade animations. On OS X, you can also use Core Image filters to create transitions that use other types of effects such as wipes, page curls, ripples, or custom effects that you devise.

To perform a transition animation, you create a `CATransition` object and add it to the layers involved in the transition. You use the transition object to specify the type of transition to perform and the start and end points of the transition animation. You do not need to use the entire transition animation either. The transition object lets you specify the start and end progress values to use when animating. These values let you do things like start or end an animation at its midpoint.

Listing 5-1 shows the code used to create an animated push transition between two views. In the example, both `myView1` and `myView2` are located at the same position in the same parent view but only `myView1` is currently visible. The push transition causes `myView1` to slide out to the left and fade until it is hidden while `myView2` slides in from the right and becomes visible. Updating the hidden property of both views ensures that the visibility of both views is correct at the end of the animation.

Listing 5-1 Animating a transition between two views in iOS

```
CATransition* transition = [CATransition animation];
transition.startProgress = 0;
transition.endProgress = 1.0;
transition.type = kCATransitionPush;
```

```
transition.subtype = kCATransitionFromRight;
transition.duration = 1.0;

// Add the transition animation to both layers
[myView1.layer addAnimation:transition forKey:@"transition"];
[myView2.layer addAnimation:transition forKey:@"transition"];

// Finally, change the visibility of the layers.
myView1.hidden = YES;
myView2.hidden = NO;
```

When two layers are involved in the same transition, you can use the same transition object for both. Using the same transition object also simplifies the code you have to write. However, you can use different transition objects and would definitely need to do so if the transition parameters for each layer are different.

Listing 5-2 shows how to use a Core Image filter to implement a transition effect on OS X. After configuring the filter with the parameters you want, assign it to the `filter` property of the transition object. After that, the process for applying the animation is the same as for other types of animation objects.

Listing 5-2 Using a Core Image filter to animate a transition on OS X

```
// Create the Core Image filter, setting several key parameters.
CIFilter* aFilter = [CIFilter filterWithName:@"CIBarsSwipeTransition"];
[aFilter setValue:[NSNumber numberWithFloat:3.14] forKey:@"inputAngle"];
[aFilter setValue:[NSNumber numberWithFloat:30.0] forKey:@"inputWidth"];
[aFilter setValue:[NSNumber numberWithFloat:10.0] forKey:@"inputBarOffset"];

// Create the transition object
CATransition* transition = [CATransition animation];
transition.startProgress = 0;
transition.endProgress = 1.0;
transition.filter = aFilter;
transition.duration = 1.0;

[self.imageView2 setHidden:NO];
[self.imageView.layer addAnimation:transition forKey:@"transition"];
```

```
[self.imageView2.layer addAnimation:transition forKey:@"transition"];
[self.imageView setHidden:YES];
```

Note: When using Core Image filters in an animation, the trickiest part is configuring the filter. For example, with the bar swipe transition, specifying an input angle that is too high or too low might make it seem as if no transition is happening. If you are not seeing the animation you expected, try adjusting your filter parameters to different values to see if that changes the results.

Customizing the Timing of an Animation

Timing is an important part of animations, and with Core Animation you specify precise timing information for your animations through the methods and properties of the `CAMediaTiming` protocol. Two Core Animation classes adopt this protocol. The `CAAnimation` class adopts it so that you can specify timing information in your animation objects. The `CALayer` also adopts it so that you can configure some timing-related features for your implicit animations, although the implicit transaction object that wraps those animations usually provides default timing information that takes precedence.

When thinking about timing and animations, it is important to understand how layer objects work with time. Each layer has its own local time that it uses to manage animation timing. Normally, the local time of two different layers is close enough that you could specify the same time values for each and the user might not notice anything. However, the local time of a layer can be modified by its parent layers or by its own timing parameters. For example, changing the layer's `speed` property causes the duration of animations on that layer (and its sublayers) to change proportionally.

To assist you in making sure time values are appropriate for a given layer, the `CALayer` class defines the `convertTime:fromLayer:` and `convertTime:toLayer:` methods. You can use these methods to convert a fixed time value to the local time of a layer or to convert time values from one layer to another. The methods take into account the media timing properties that might affect the local time of the layer and return a value that you can use with the other layer. Listing 5-3 shows an example that you should use regularly to get the current local time for a layer. The `CACurrentMediaTime` function is a convenience function that returns the computer's current clock time, which the method takes and converts to the layer's local time.

Listing 5-3 Getting a layer's current local time

```
CFTimeInterval localLayerTime = [myLayer convertTime:CACurrentMediaTime()
fromLayer:nil];
```

Once you have a time value in the layer's local time, you can use that value to update the timing-related properties of an animation object or layer. With these timing properties, you can achieve some interesting animation behaviors, including:

- Use the `beginTime` property to set the start time of an animation. Normally, animations begin during the next update cycle. You can use the `beginTime` parameter to delay the animation start time by several seconds. The way to chain two animations together is to set the begin time of one animation to match the end time of the other animation.

  If you delay the start of an animation, you might also want to set the `fillMode` property to `kCAFillModeBackwards`. This fill mode causes the layer to display the animation's start value, even if the layer object in the layer tree contains a different value. Without this fill mode, you would see a jump to the final value before the animation starts executing. Other fill modes are available too.
- The `autoreverses` property causes an animation to execute for the specified duration and then return to the starting value of the animation. You can combine this property with the `repeatCount` property to animate back and forth between the start and end values. Setting the repeat count to a whole number (such as 1.0) for an autoreversing animation causes the animation to stop on its starting value. Adding an extra half step (such as a repeat count of 1.5) causes the animation to stop on its end value.
- Use the `timeOffset` property with group animations to start some animations at a later time than others.

Pausing and Resuming Animations

To pause an animation, you can take advantage of the fact that layers adopt the `CAMediaTiming` protocol and set the speed of the layer's animations to `0.0`. Setting the speed to zero pauses the animation until you change the value back to a nonzero value. Listing 5-4 shows a simple example of how to both pause and resume the animations later.

Listing 5-4 Pausing and resuming a layer's animations

```
-(void)pauseLayer:(CALayer*)layer {
   CFTimeInterval pausedTime = [layer convertTime:CACurrentMediaTime()
fromLayer:nil];
   layer.speed = 0.0;
   layer.timeOffset = pausedTime;
}

-(void)resumeLayer:(CALayer*)layer {
   CFTimeInterval pausedTime = [layer timeOffset];
```

```
    layer.speed = 1.0;
    layer.timeOffset = 0.0;
    layer.beginTime = 0.0;
    CFTimeInterval timeSincePause = [layer convertTime:CACurrentMediaTime()
fromLayer:nil] - pausedTime;
    layer.beginTime = timeSincePause;
}
```

Explicit Transactions Let You Change Animation Parameters

Every change you make to a layer must be part of a transaction. The `CATransaction` class manages the creation and grouping of animations and their execution at the appropriate time. In most cases, you do not need to create your own transactions. Core Animation automatically creates an implicit transaction whenever you add explicit or implicit animations to one of your layers. However, you can also create explicit transactions to manage those animations more precisely.

You create and manage transactions using the methods of the `CATransaction` class. To start (and implicitly create) a new transaction call the `begin` class method; to end that transaction, call the `commit` class method. In between those calls are the changes that you want to be part of the transaction. For example, to change two properties of a layer, you could use the code in Listing 5-5.

Listing 5-5 Creating an explicit transaction

```
[CATransaction begin];
theLayer.zPosition=200.0;
theLayer.opacity=0.0;
[CATransaction commit];
```

One of the main reasons to use transactions is that within the confines of an explicit transaction, you can change the duration, timing function, and other parameters. You can also assign a completion block to the entire transaction so that your app can be notified when the group of animations finishes. Changing animation parameters requires modifying the appropriate key in the transaction dictionary using the `setValue:forKey:` method. For example, to change the default duration to 10 seconds, you would change the `kCATransactionAnimationDuration` key, as shown in Listing 5-6.

Listing 5-6 Changing the default duration of animations

```
[CATransaction begin];
```

```
[CATransaction setValue:[NSNumber numberWithFloat:10.0f]
                 forKey:kCATransactionAnimationDuration];
// Perform the animations
[CATransaction commit];
```

You can nest transactions in situations where you want to provide different default values for different sets of animations. To nest one transaction inside of another, just call the `begin` class method again. Each `begin` call must be matched by a corresponding call to the `commit` method. Only after you commit the changes for the outermost transaction does Core Animation begin the associated animations.

Listing 5-7 shows an example of one transaction nested inside another. In this example, the inner transaction changes the same animation parameter as the outer transaction but uses a different value.

Listing 5-7 Nesting explicit transactions

```
[CATransaction begin]; // Outer transaction

// Change the animation duration to two seconds
[CATransaction setValue:[NSNumber numberWithFloat:2.0f]
                forKey:kCATransactionAnimationDuration];
// Move the layer to a new position
theLayer.position = CGPointMake(0.0,0.0);

[CATransaction begin]; // Inner transaction
// Change the animation duration to five seconds
[CATransaction setValue:[NSNumber numberWithFloat:5.0f]
                 forKey:kCATransactionAnimationDuration];

// Change the zPosition and opacity
theLayer.zPosition=200.0;
theLayer.opacity=0.0;

[CATransaction commit]; // Inner transaction

[CATransaction commit]; // Outer transaction
```

Adding Perspective to Your Animations

Apps can manipulate layers in three spatial dimensions, but for simplicity Core Animation displays layers using a parallel projection, which essentially flattens the scene into a two-dimensional plane. This default behavior causes identically sized layers with different `zPosition` values to appear as the same size, even if they are far apart on the z axis. The perspective that you would normally have viewing such a scene in three dimensions is gone. However, you can change that behavior by modifying the transformation matrix of your layers to include perspective information.

When modifying the perspective of a scene, you need to modify the `sublayerTransform` matrix of the superlayer that contains the layers being viewed. Modifying the superlayer simplifies the code you have to write by applying the same perspective information to all of the child layers. It also ensures that the perspective is applied correctly to sibling sublayers that overlap each other in different planes.

Listing 5-8 shows the way to create a simple perspective transform for a parent layer. In this case the custom `eyePosition` variable specifies the relative distance along the z axis from which to view the layers. Usually you specify a positive value for `eyePosition` to keep the layers oriented in the expected way. Larger values result in a flatter scene while smaller values cause more dramatic visual differences between the layers.

Listing 5-8 Adding a perspective transform to a parent layer

```
CATransform3D perspective = CATransform3DIdentity;
perspective.m34 = -1.0/eyePosition;

// Apply the transform to a parent layer.
myParentLayer.sublayerTransform = perspective;
```

With the parent layer configured, you can change the `zPosition` property of any child layers and observe how their size changes based on their relative distance from the eye position.

Changing a Layer's Default Behavior

Core Animation implements its implicit animation behaviors for layers using action objects. An action object is an object that conforms to the `CAAction` protocol and defines some relevant behavior to perform on a layer. All `CAAnimation` objects implement the protocol, and it is these objects that are usually assigned to be executed whenever a layer property changes.

Animating properties is one type of action but you can define actions with almost any behavior you want. To do that, though, you have to define your action objects and associate them with your app's layer objects.

Custom Action Objects Adopt the CAAction Protocol

To create your own action object, adopt the `CAAction` protocol from one of your classes and implement the `runActionForKey:object:arguments:` method. In that method, use the available information to perform whatever actions you want to take on the layer. You might use the method to add an animation object to the layer or you might use it to perform other tasks.

When you define an action object, you must decide how you want that action to be triggered. The trigger for an action defines the key you use to register that action later. Action objects can be triggered by any of the following situations:

- The value of one of the layer's properties changed. This can be any of the layer's properties and not just the animatable ones. (You can also associate actions with custom properties you add to your layers.) The key that identifies this action is the name of the property.
- The layer became visible or was added to a layer hierarchy. The key that identifies this action is `kCAOnOrderIn`.
- The layer was removed from a layer hierarchy. The key that identifies this action is `kCAOnOrderOut`.
- The layer is about to be involved in a transition animation. The key that identifies this action is `kCATransition`.

Action Objects Must Be Installed On a Layer to Have an Effect

Before an action can be performed, the layer needs to find the corresponding action object to execute. The key for layer-related actions is either the name of the property being modified or a special string that identifies the action. When an appropriate event occurs on the layer, the layer calls its `actionForKey:` method to search for the action object associated with the key. Your app can interpose itself at several points during this search and provide a relevant action object for that key.

Core Animation looks for action objects in the following order:

1. If the layer has a delegate and that delegate implements the `actionForLayer:forKey:` method, the layer calls that method. The delegate must do one of the following:
    - Return the action object for the given key.
    - Return `nil` if it does not handle the action, in which case the search continues.
    - Return the `NSNull` object, in which case the search ends immediately.
2. The layer looks for the given key in the layer's `actions` dictionary.
3. The layer looks in the `style` dictionary for an actions dictionary that contains the key. (In other word, the `style` dictionary contains an `actions` key whose value is also a dictionary. The layer looks for the given key in this second dictionary.)
4. The layer calls its `defaultActionForKey:` class method.
5. The layer performs the implicit action (if any) defined by Core Animation.

If you provide an action object at any of the appropriate search points, the layer stops its search and executes the returned action object. When it finds an action object, the layer calls that object's `runActionForKey:object:arguments:` method to perform the action. If the action you define for a given key is already an instance of the `CAAnimation` class, you can use the default implementation of that method to perform the animation. If you are defining your own custom object that conforms to the `CAAction` protocol, you must use your object's implementation of that method to take whatever actions are appropriate.

Where you install your action objects depends on how you intend to modify the layer.

- For actions that you might apply only in specific circumstances, or for layers that already use a delegate object, provide a delegate and implement its `actionForLayer:forKey:` method.
- For layer objects that do not normally use a delegate, add the action to the layer's `actions` dictionary.
- For actions related to custom properties that you define on the layer object, include the action in the layer's `style` dictionary.
- For actions that are fundamental to the behavior of the layer, subclass the layer and override the `defaultActionForKey:` method.

Listing 6-1 shows an implementation of the delegate method used to provide action objects. In this case, the delegate looks for changes to the layer's `contents` property and swaps the new contents into place using a transition animation.

Listing 6-1 Providing an action using a layer delegate object

```
- (id<CAAction>)actionForLayer:(CALayer *)theLayer
                        forKey:(NSString *)theKey {
    CATransition *theAnimation=nil;

    if ([theKey isEqualToString:@"contents"]) {

        theAnimation = [[CATransition alloc] init];
        theAnimation.duration = 1.0;
        theAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseIn];
        theAnimation.type = kCATransitionPush;
        theAnimation.subtype = kCATransitionFromRight;
    }
    return theAnimation;
}
```

Disable Actions Temporarily Using the CATransaction Class

You can temporarily disable layer actions using the `CATransaction` class. When you change the property of a layer, Core Animation usually creates an implicit transaction object to animate the change. If you do not want to animate the change, you can disable implicit animations by creating an explicit transaction and setting its `kCATransactionDisableActions` property to `true`. Listing 6-2 shows a snippet of code that disables animations when removing the specified layer from the layer tree.

Listing 6-2 Temporarily disabling a layer's actions

```
[CATransaction begin];
[CATransaction setValue:(id)kCFBooleanTrue
                 forKey:kCATransactionDisableActions];
[aLayer removeFromSuperlayer];
```

```
[CATransaction commit];
```

For more information about using transaction objects to manage animation behavior, see Explicit Transactions Let You Change Animation Parameters (page 67).

Improving Animation Performance

Objective-CSwift

Core Animation is a great way to improve the frame rates for app-based animations but its use is not a guarantee of improved performance. Especially in OS X, you must still make choices about the most effective way to use Core Animation behaviors. And as with all performance-related issues, you should use Instruments to measure and track the performance of your app over time so that you can ensure that performance is improving and not regressing.

Choose the Best Redraw Policy for Your OS X Views

The default redraw policy for the `NSView` class preserves the original drawing behavior of that class, even if the view is layer-backed. If you are using layer-backed views in your app, you should examine the redraw policy choices and choose the one that offers the best performance for your app. In most cases, the default policy is not the one that is likely to offer the best performance. Instead, the `NSViewLayerContentsRedrawOnSetNeedsDisplay` policy is more likely to reduce the amount of drawing your app does and improve performance. Other policies might also offer better performance for specific types of views.

For more information on view redraw policies, see The Layer Redraw Policy for OS X Views Affects Performance (page 40).

Update Layers in OS X to Optimize Your Rendering Path

In OS X v10.8 and later, views have two options for updating the underlying layer's contents. When you update a layer-backed view in OS X v10.7 and earlier, the layer captures the drawing commands from the view's `drawRect:` method into the backing bitmap image. Caching the drawing commands is effective but is not the most efficient option in all cases. If you know how to provide the layer's contents directly without actually rendering them, you can use the `updateLayer` method to do so.

For information about the different paths for rendering, including those that involve the `updateLayer` method, see Using a Delegate to Provide the Layer's Content (page 31).

General Tips and Tricks

There are several ways to make your layer implementations more efficient. As with any such optimizations, though, you should always measure the current performance of your code before attempting to optimize. This gives you a baseline against that you can use to determine if the optimizations are working.

Use Opaque Layers Whenever Possible

Setting the `opaque` property of your layer to `YES` lets Core Animation know that it does not need to maintain an alpha channel for the layer. Not having an alpha channel means that the compositor does not need to blend the contents of your layer with its background content, which saves time during rendering. However, this property is relevant primarily for layers that are part of a layer-backed view or situations where Core Animation creates the underlying layer bitmap. If you assign an image directly to the layer's `contents` property, the alpha channel of that image is preserved regardless of the value in the `opaque` property.

Use Simpler Paths for CAShapeLayer Objects

The `CAShapeLayer` class creates its content by rendering the path you provide into a bitmap image at composite time. The advantage is that the layer always draws the path at the best possible resolution but that advantage comes at the cost of additional rendering time. If the path you provide is complex, rasterizing that path might get too expensive. And if the size of the layer changes frequently (and thus must be redrawn frequently), the amount of time spent drawing can add up and become a performance bottleneck.

One way to minimize drawing time for shape layers is to break up complex shapes into simpler shapes. Using simpler paths and layering multiple `CAShapeLayer` objects on top of one another in the compositor can be much faster than drawing one large complex path. That is because the drawing operations happen on the CPU whereas compositing takes place on the GPU. As with any simplifications of this nature, though, the potential performance gains are dependent on your content. Therefore, it is especially important to measure the performance of your code before optimizing so that you have a baseline to use for comparisons.

Set the Layer Contents Explicitly for Identical Layers

If you are using the same image in multiple layer objects, load the image yourself and assign it directly to the `contents` property of those layer objects. Assigning an image to the `contents` property prevents the layer from allocating memory for a backing store. Instead, the layer uses the image you provide as its backing store. When several layers use the same image, this means that all of those layers are sharing the same memory rather than allocating a copy of the image for themselves.

Always Set a Layer's Size to Integral Values

For best results, always set the width and height of your layer objects to integral values. Although you specify the width and height of your layer's bounds using floating-point numbers, the layer bounds are ultimately used to create a bitmap image. Specifying integral values for the width and height simplifies the work that Core Animation must do to create and manage the backing store and other layer information.

Use Asynchronous Layer Rendering As Needed

Any drawing that you do in your delegate's `drawLayer:inContext:` method or your view's `drawRect:` method normally occurs synchronously on your app's main thread. In some situations, though, drawing your content synchronously might not offer the best performance. If you notice that your animations are not performing well, you might try enabling the `drawsAsynchronously` property on your layer to move those operations to a background thread. If you do so, make sure your drawing code is thread safe. And as always, you should always measure the performance of drawing asynchronously before putting it into your production code.

Specify a Shadow Path When Adding a Shadow to Your Layer

Letting Core Animation determine the shape of a shadow can be expensive and impact your app's performance. Rather than letting Core Animation determine the shape of the shadow, specify the shadow shape explicitly using the `shadowPath` property of `CALayer`. When you specify a path object for this property, Core Animation uses that shape to draw and cache the shadow effect. For layers whose shape never changes or rarely changes, this greatly improves performance by reducing the amount of rendering done by Core Animation.

Layer Style Property Animations

Objective-CSwift

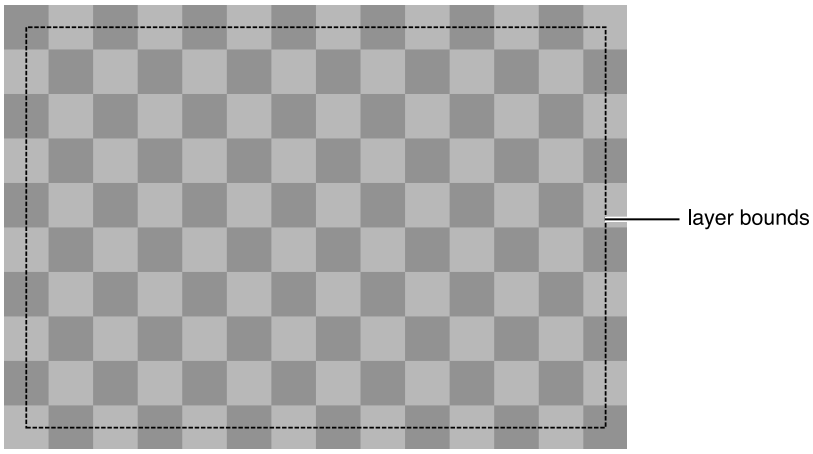
During the rendering process, Core Animation takes different attributes of the layer and renders them in a specific order. This order determines the final appearance of the layer. This chapter illustrates the rendered results achieved by setting different layer style properties.

> **Note:** The layer style properties available on Mac OS X and iOS differ and are noted throughout this chapter.

Geometry Properties

A layer's geometry properties specify how it is displayed relative to its parent layer. The geometry also specifies the radius used to round the layer corners and a transform that is applied to the layer and its sublayers. Figure A-1 shows the bounding rectangle of the example layer.

Figure A-1 Layer geometry



The following `CALayer` properties specify a layer's geometry:

- `bounds`
- `position`
- `frame` (computed from the `bounds` and `position` and is not animatable)

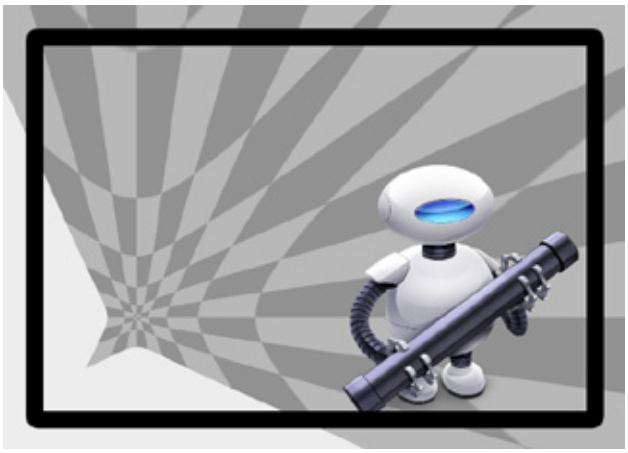
- `anchorPoint`
- `cornerRadius`
- `transform`
- `zPosition`

> **iOS Note:** The `cornerRadius` property is supported only in iOS 3.0 and later.

Background Properties

The first thing Core Animation renders is the layer's background. You can specify a color for the background. In OS X, you can also specify a Core Image filter that you want to apply to the background content. Figure A-2 shows two versions of a sample layer. The layer on the left has its `backgroundColor` property set while the layer on the right has no background color but does have a border some content and a pinch distortion filter assigned to its `backgroundFilters` property.

Figure A-2 Layer with background color



The background filter is applied to the content that lies behind the layer, which primarily consists of the parent layer's content. You might use a background filter to make the foreground layer content stand out; for example, by applying a blur filter.

The following `CALayer` properties affect the display of a layer's background:

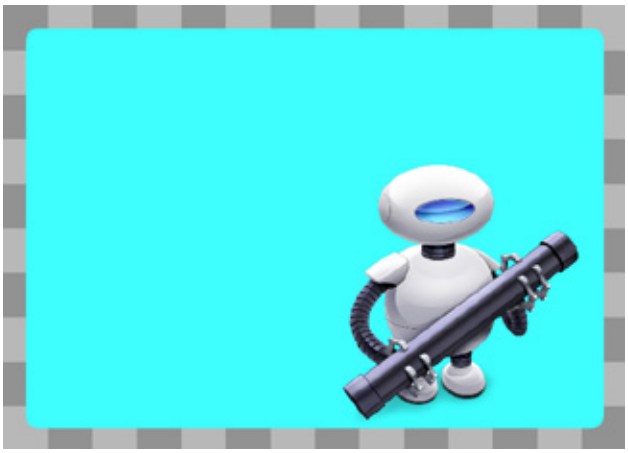
- `backgroundColor`
- `backgroundFilters` (not supported in iOS)

Platform Note: In iOS, the `backgroundFilters` property is exposed in the `CALayer` class but the filters you assign to this property are ignored.

Layer Content

If the layer has any content, that content is rendered on top of the background color. You can provide layer content by setting a bitmap directly, by using a delegate to specify the content, or by subclassing the layer and drawing the content directly. And you can use many different drawing technologies (including Quartz, Metal, OpenGL, and Quartz Composer) to provide that content. Figure A-3 shows a sample layer whose contents are a bitmap that was set directly. The bitmap content consists of a largely transparent space with the Automator icon in the lower right corner.

Figure A-3 Layer displaying a bitmap image



Layers with a corner radius do not automatically clip their contents; however, setting the layer's `masksToBounds` property to `YES` does cause the layer to clip to its corner radius.

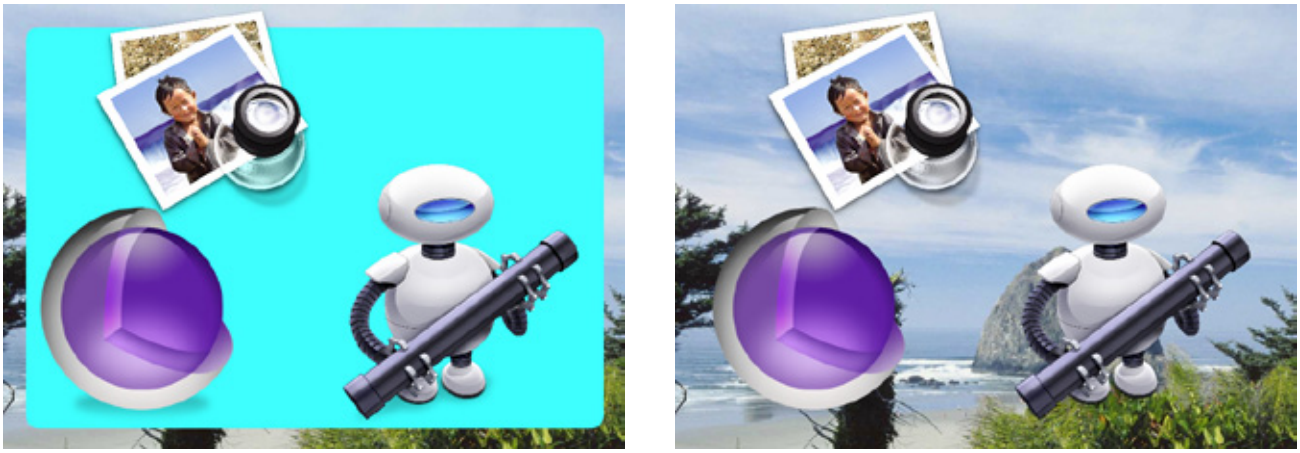
The following `CALayer` properties affect the display of a layer's content:

- `contents`
- `contentsGravity`
- `masksToBounds`

Sublayers Content

Any layer may contain one or more child layers, known as sublayers. Sublayers are rendered recursively and positioned relative to the parent layer's bounds rectangle. In addition, Core Animation applies the parent layer's `sublayerTransform` to each sublayer relative to the parent layer's anchor point. You can use the sublayer transform to apply perspective and other effects to all of the layers equally. Figure A-4 shows a sample layer with two sublayers. The version on the left includes a background color while the version on the right does not.

Figure A-4 Layer displaying the sublayers content



Setting the `masksToBounds` property of a layer to `YES` causes any sublayers to be clipped to the bounds of the layer.

The following `CALayer` properties affect the display of a layer's sublayers:

- `sublayers`
- `masksToBounds`
- `sublayerTransform`

Border Attributes

A layer can display an optional border using a specified color and width. The border follows the bounds rectangle of the layer and takes into account any corner radius values. Figure A-5 shows a sample layer after applying a border. Notice that content and sublayers that are outside the layer's bounds are rendered underneath the border.

Figure A-5 Layer displaying the border attributes content

The following `CALayer` properties affect the display of a layer's borders:

- `borderColor`
- `borderWidth`

Platform Note: The `borderColor` and `borderWidth` properties are supported only in iOS 3.0 and later.

Filters Property

In OS X, you may apply one or more filters to the layer's content and use a custom compositing filter to specify how the layer's contents blend with the content of its underlying layer. Figure A-6 shows a sample layer with the Core Image posterize filter applied.

Figure A-6 Layer displaying the filters properties

The following `CALayer` property specifies a layers content filters:

- `filters`
- `compositingFilter`

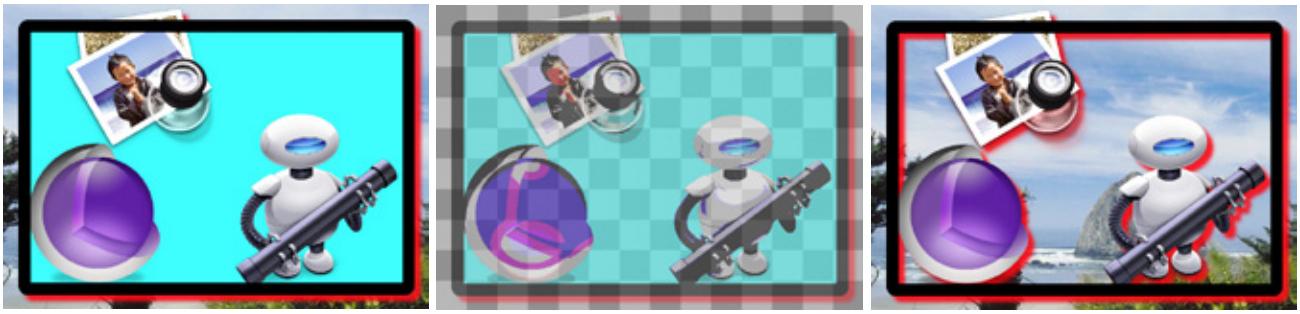
Platform Note: In iOS, layers ignore any filters you assign to them.

Shadow Properties

Layers can display shadow effects and configure their shape, opacity, color, offset, and blur radius. If you do not specify a custom shadow shape, the shadow is based on the portions of the layer that are not fully transparent. Figure A-7 shows several different versions of the same sample layer with a red shadow applied.

The left and middle versions include a background color so the shadow appears only around the border of the layer. However, the version on the right does not include a background color. In this case, the shadow is applied to the layer's content, border, and sublayers.

Figure A-7 Layer displaying the shadow properties



The following `CALayer` properties affect the display of a layer's shadow:

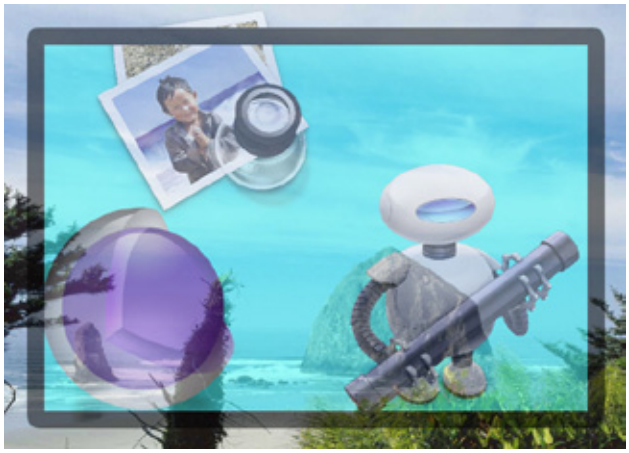
- `shadowColor`
- `shadowOffset`
- `shadowOpacity`
- `shadowRadius`
- `shadowPath`

Platform Note: The `shadowColor`, `shadowOffset`, `shadowOpacity`, and `shadowRadius` properties are supported in iOS 3.2 and later. The `shadowPath` property is supported in iOS 3.2 and later and in OS X v10.7 and later.

Opacity Property

The opacity property of a layer determines how much background content shows through the layer. Figure A-8 shows a sample layer whose opacity is set to `0.5`. This allows portions of the background image to show through.

Figure A-8 Layer including the opacity property



The following `CALayer` property specifies the opacity of a layer:

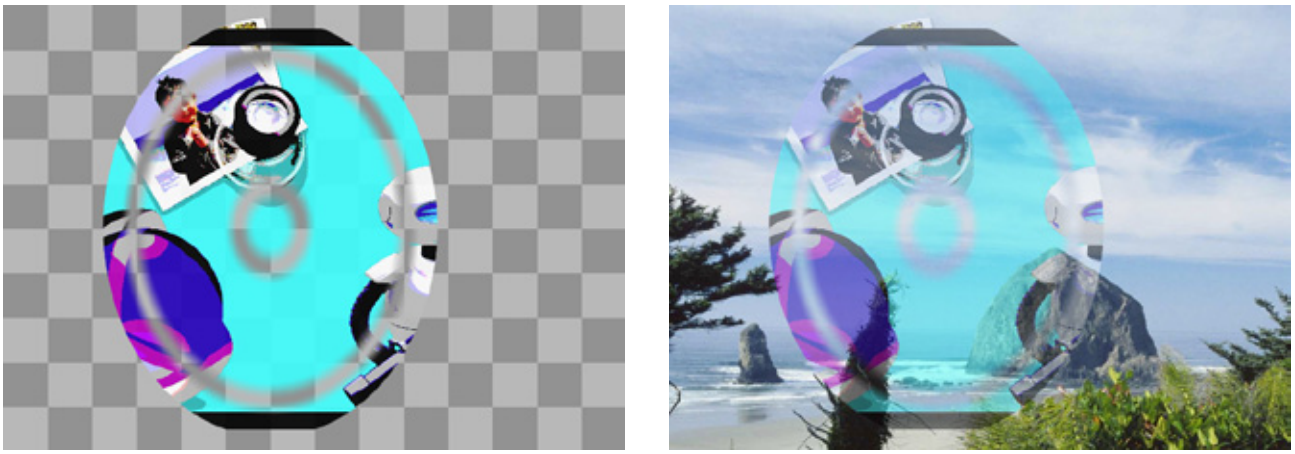
- `opacity`

Mask Properties

You can use a mask to obscure all or part of a layer's contents. The mask is itself a layer object whose alpha channel is used to determine what is blocked and what is transmitted. Opaque portions of the mask layer's contents allow the underlying layer content to show through while transparent portions partially or fully obscure the underlying content. Figure A-9 shows a sample layer composited with a mask layer and two

different backgrounds. In the left version, the layer's opacity is set to 1.0. In the right version, the layer's opacity is set to 0.5, which increases the amount of background content that is transmitted through the masked portion of the layer.

Figure A-9 Layer composited with the mask property



The following `CALayer` property specifies the mask for a layer:

- `mask`

> **Platform Note:** The `mask` property is supported in iOS 3.0 and later.

Animatable Properties

Many of the properties in `CALayer` and `CIFilter` can be animated. This appendix lists those properties, along with the animation used by default.

CALayer Animatable Properties

Table B-1 lists the properties of the `CALayer` class that you might consider animating. For each property, the table also lists the type of default animation object that is created to execute an implicit animation.

Table B-1 Layer properties and their default animations

Property	Default animation
`anchorPoint`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`backgroundColor`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`backgroundFilters`	Uses the default implied `CATransition` object, described in Table B-3 (page 88). Sub-properties of the filters are animated using the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`borderColor`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`borderWidth`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`bounds`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`compositingFilter`	Uses the default implied `CATransition` object, described in Table B-3 (page 88). Sub-properties of the filters are animated using the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`contents`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).

Property	Default animation
`contentsRect`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`cornerRadius`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`doubleSided`	There is no default implied animation.
`filters`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88). Sub-properties of the filters are animated using the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`frame`	This property is not animatable. You can achieve the same results by animating the `bounds` and `position` properties.
`hidden`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`mask`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`masksToBounds`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`opacity`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`position`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`shadowColor`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`shadowOffset`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`shadowOpacity`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`shadowPath`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`shadowRadius`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).

Property	Default animation
`sublayers`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`sublayerTransform`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`transform`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).
`zPosition`	Uses the default implied `CABasicAnimation` object, described in Table B-2 (page 88).

Table B-2 lists the animation attributes for the default property-based animations.

Table B-2 Default Implied Basic Animation

Description	Value
Class	`CABasicAnimation`
Duration	0.25 seconds, or the duration of the current transaction
Key path	Set to the property name of the layer.

Table B-3 lists the animation object configuration for default transition-based animations.

Table B-3 Default Implied Transition

Description	Value
Class	`CATransition`
Duration	0.25 seconds, or the duration of the current transaction
Type	Fade (`kCATransitionFade`)
Start progress	`0.0`
End progress	`1.0`

CIFilter Animatable Properties

Core Animation adds the following animatable properties to Core Image's `CIFilter` class. These properties are available only on OS X.

- `name`
- `enabled`

For more information about these additions, see *CIFilter Core Animation Additions*.

Key-Value Coding Extensions

Core Animation extends the `NSKeyValueCoding` protocol as it pertains to the `CAAnimation` and `CALayer` classes. This extension adds default values for some keys, expands wrapping conventions, and adds key path support for `CGPoint`, `CGRect`, `CGSize`, and `CATransform3D` types.

Key-Value Coding Compliant Container Classes

The `CAAnimation` and `CALayer` classes are key-value coding compliant container classes, which means that you can set values for arbitrary keys. Even if the key `someKey` is not a declared property of the `CALayer` class, you can still set a value for it as follows:

```
[theLayer setValue:[NSNumber numberWithInteger:50] forKey:@"someKey"];
```

You can also retrieve the value for arbitrary keys like you would retrieve the value for other key paths. For example, to retrieve the value of the `someKey` path set previously, you would use the following code:

```
someKeyValue=[theLayer valueForKey:@"someKey"];
```

> **OS X Note:** The `CAAnimation` and `CALayer` classes, which automatically archive any additional keys that you set up for instances of those classes, support the `NSCoding` protocol.

Default Value Support

Core Animation adds a convention to key value coding whereby a class can provide a default value for a key that has no set value. The `CAAnimation` and `CALayer` classes support this convention using the `defaultValueForKey:` class method.

To provide a default value for a key, create a subclass of the desired class and override its `defaultValueForKey:` method. Your implementation of this method should examine the key parameter and return the appropriate default value. Listing C-1 shows a sample implementation of the `defaultValueForKey:` method for a layer object that provides a default value for the `masksToBounds` property.

Listing C-1 Example implementation of defaultValueForKey:

```
+ (id)defaultValueForKey:(NSString *)key
{
    if ([key isEqualToString:@"masksToBounds"])
         return [NSNumber numberWithBool:YES];

    return [super defaultValueForKey:key];
}
```

Wrapping Conventions

When the data for a key consists of a scalar value or C data structure, you must wrap that type in an object before assigning it to the layer. Similarly, when accessing that type, you must retrieve an object and then unwrap the appropriate values using the extensions to the appropriate class. Table C-1 lists the C types commonly used and the Objective-C class you use to wrap them.

Table C-1 Wrapper classes for C types

C type	Wrapping class
`CGPoint`	`NSValue`
`CGSize`	`NSValue`
`CGRect`	`NSValue`
`CATransform3D`	`NSValue`
`CGAffineTransform`	`NSAffineTransform` (OS X only)

Key Path Support for Structures

The `CAAnimation` and `CALayer` classes lets you access the fields of selected data structures using key paths. This feature is a convenient way to specify the field of a data structure that you want to animate. You can also use these conventions in conjunction with the `setValue:forKeyPath:` and `valueForKeyPath:` methods to set and get those fields.

CATransform3D Key Paths

You can use the enhanced key path support to retrieve specific transformation values for a property that contains a `CATransform3D` data type. To specify the full key path for a layer's transforms, you would use the string value `transform` or `sublayerTransform` followed by one of the field key paths in Table C-2. For example, to specify a rotation factor around the layer's z axis, you would specify the key path `transform.rotation.z`.

Table C-2 Transform field value key paths

Field Key Path	Description
`rotation.x`	Set to an `NSNumber` object whose value is the rotation, in radians, in the x axis.
`rotation.y`	Set to an `NSNumber` object whose value is the rotation, in radians, in the y axis.
`rotation.z`	Set to an `NSNumber` object whose value is the rotation, in radians, in the z axis.
`rotation`	Set to an `NSNumber` object whose value is the rotation, in radians, in the z axis. This field is identical to setting the `rotation.z` field.
`scale.x`	Set to an `NSNumber` object whose value is the scale factor for the x axis.
`scale.y`	Set to an `NSNumber` object whose value is the scale factor for the y axis.
`scale.z`	Set to an `NSNumber` object whose value is the scale factor for the z axis.
`scale`	Set to an `NSNumber` object whose value is the average of all three scale factors.
`translation.x`	Set to an `NSNumber` object whose value is the translation factor along the x axis.
`translation.y`	Set to an `NSNumber` object whose value is the translation factor along the y axis.
`translation.z`	Set to an `NSNumber` object whose value is the translation factor along the z axis.
translation	Set to an `NSValue` object containing an `NSSize` or `CGSize` data type. That data type indicates the amount to translate in the x and y axis.

The following example shows how you can modify a layer using the `setValue:forKeyPath:` method. The example sets the translation factor for the x axis to 10 points, causing the layer to shift by that amount along the indicated axis.

```
[myLayer setValue:[NSNumber numberWithFloat:10.0]
forKeyPath:@"transform.translation.x"];
```

> **Note:** Setting values using key paths is not the same as setting them using Objective-C properties. You cannot use property notation to set transform values. You must use the `setValue:forKeyPath:` method with the preceding key path strings.

CGPoint Key Paths

If the value of a given property is a `CGPoint` data type, you can append one of the field names in Table C-3 to the property to get or set that value. For example, to change the x component of a layer's `position` property, you could write to the key path `position.x`.

Table C-3 CGPoint data structure fields

Structure Field	Description
`x`	The x component of the point.
`y`	The y component of the point.

CGSize Key Paths

If the value of a given property is a `CGSize` data type, you can append one of the field names in Table C-4 to the property to get or set that value.

Table C-4 CGSize data structure fields

Structure Field	Description
`width`	The width component of the size.
`height`	The height component of the size.

CGRect Key Paths

If the value of a given property is a `CGRect` data type, you can append the following field names in Table C-3 to the property to get or set that value. For example, to change the width component of a layer's `bounds` property, you could write to the key path `bounds.size.width`.

Table C-5 CGRect data structure fields

Structure Field	Description
`origin`	The origin of the rectangle as a `CGPoint`.
`origin.x`	The x component of the rectangle origin.
`origin.y`	The y component of the rectangle origin.
`size`	The size of the rectangle as a `CGSize`.
`size.width`	The width component of the rectangle size.
`size.height`	The height component of the rectangle size.

Document Revision History

This table describes the changes to *Core Animation Programming Guide*.

Date	Notes
2015-03-09	Added brief discussion of CAMetalLayer among possible layer classes.
2013-01-28	Major revamp, reorganization, and expansion to cover modern Core Animation behavior in iOS and OS X. Incorporated the content of *Animation Types and Timing Programming Guide* into this document.
2010-09-24	Updated the document to reflect Core Animation support in iOS 4.2.
2010-08-12	Corrected iOS origin information. Clarified that the coordinate system origin used in the examples are based on the OS X model.
2010-05-25	Corrected autoresizing masks table.
2010-03-24	Added missing constant to the contentGravity property resizing table in Providing Layer Content.
2010-02-24	Updated Core Animation Kiosk Style Menu tutorial project.
2010-01-20	Updated infinite value for repeatCount.
2009-10-19	Modified section headings.
2009-08-13	Corrected availability of cornerRadius on iOS v 3.0 and later.
2008-11-13	Introduces iOS SDK content to OS X content. Corrects frame animation capabilities.
2008-09-09	Corrected typos.

Date	Notes
2008-06-18	Updated for iOS.
2008-05-06	Corrected typos.
2008-03-11	Corrected typos.
2008-02-08	Corrected typos. Corrected RadiansToDegrees() calculation.
2007-12-11	Corrected typos.
2007-10-31	Added information on the presentation tree. Added example application walkthough.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010