Core Animation Programming Guide

Graphics & Imaging > Quartz

2008-11-13

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

Contents

Figures, Tables, and Listings

Animation 39

Actions 43

Transactions 47

Laying Out Core Animation Layers 51

Core Animation Extensions To Key-Value Coding 55

Layer Style Properties 59

Introduction to Core Animation Programming Guide

This document describes the fundamental concepts involved in using Core Animation. Core Animation is an Objective-C framework that combines a high-performance compositing engine with a simple to use animation programming interface.

You should read this document to gain an understanding of working with Core Animation in a Cocoa application. *The Objective-C 2.0 Programming Language* should be considered a prerequisite because Core Animation makes extensive use of Objective-C properties. You should also be familiar with key-value coding as described in *Key-Value Coding Programming Guide*. Familiarity with the Quartz 2D imaging technologies described in *Quartz 2D Programming Guide* is also helpful, although not required.

You can build Cocoa applications for two platforms: the Mac OS X operating system and iPhone OS, the operating system for multi-touch devices such as iPhone and iPod touch. Core Animation Programming Guide presents Cocoa-related information for both platforms, integrating the information as much as possible and pointing out platform differences when necessary.

Organization of This Document

Core Animation Programming Guide consists of the following articles:

See Also

These programming guides discuss some of the technologies that are used by Core Animation:

- *Animation Types and Timing Programming Guide* describes the animation classes and timing features used by Core Animation.
- *Core Animation Cookbook* contains code fragments that demonstrate common Core Animation tasks.
- *Quartz 2D Programming Guide* describes the two-dimensional drawing engine used to draw the content of a `CALayer` instance.
- *Core Image Programming Guide* describes the Mac OS X image processing technology and shows how to use the Core Image API.

What Is Core Animation?

Core Animation is a collection of Objective-C classes for graphics rendering, projection, and animation. It provides fluid animations using advanced compositing effects while retaining a hierarchical layer abstraction that is familiar to developers using the Application Kit and Cocoa Touch view architectures.

Dynamic, animated user interfaces are hard to create, but Core Animation makes creating these interfaces easier by providing:

- High performance compositing with a simple approachable programming model.
- A familiar view-like abstraction that allows you to create complex user interfaces using a hierarchy of layer objects.
- A lightweight data structure. You can display and animate hundreds of layers simultaneously.
- An abstract animation interface that allows animations to run on a separate thread, independent of your application's run loop. Once an animation is configured and starts, Core Animation assumes full responsibility for running it at frame rate.
- Improved application performance. Applications need only redraw content when it changes. Minimal application interaction is required for resizing and providing layout services layers. Core Animation also eliminates application code that runs at the animation frame-rate.
- A flexible layout manager model, including a manager that allows the position and size of a layer to be set relative to attributes of sibling layers.

Using Core Animation, developers can create dynamic user interfaces for their applications without having to use low-level graphics APIs such as OpenGL to get respectable animation performance.

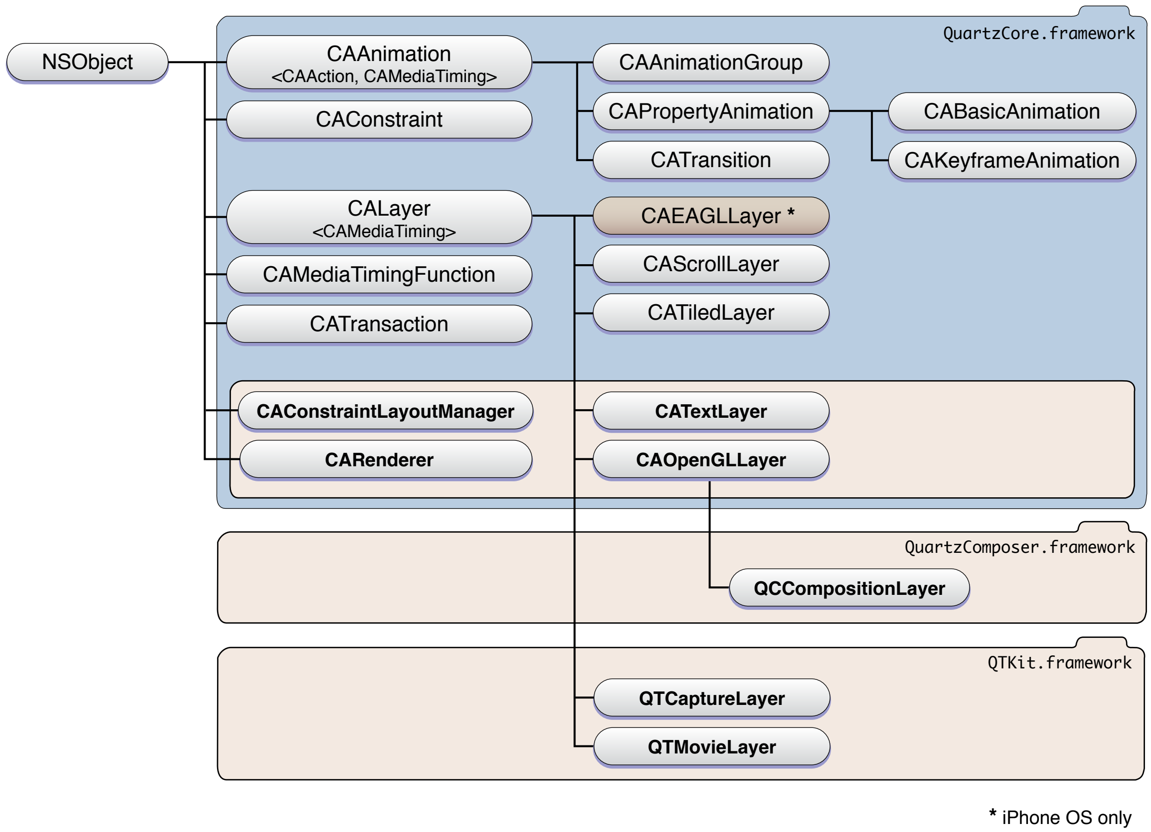
Core Animation Classes

Core Animation classes can be grouped into several categories:

- Layer classes that provide content for display
- Animation and timing classes
- Layout and constraint classes
- A transaction class that groups multiple layer changes into an atomic update

The basic Core Animation classes are contained in the Quartz Core framework, although additional layer classes can be defined in other frameworks. "Core Animation Classes" shows the class hierarchy of Core Animation.

Figure 1 Core Animation class hierarchy



Layer Classes

The layer classes are the foundation of Core Animation and provide an abstraction that should be familiar to developers who have used `NSView` or `UIView`. Basic layer functionality is provided by the `CALayer` class, which is the parent class for all types of Core Animation layers.

As with an instance of a view class, a `CALayer` instance has a single parent layer (the superlayer) and a collection of sublayers, creating a hierarchy of layers that is referred to as the layer tree. Layers are drawn from back to front just like views and specify their geometry relative to their superlayer, creating a local coordinate system. However, layers allow a more complex visual display by incorporating transform matrices that allow you to rotate, skew, scale, and project the layer content. "Layer Geometry and Transforms" (page 21) discusses layer geometry and transforms in more detail.

`CALayer` diverges from the Application Kit and Cocoa Touch view classes in that it is not necessary to subclass `CALayer` in order to display content. The content displayed by a `CALayer` instance can be provided by:

- Setting the layer's content property to a Core Graphics image representation directly, or through delegation.
- Providing a delegate that draws directly into a Core Graphics image context.
- Setting any of the number of visual style properties that all layer types have in common, for example, background colors, opacity, and masking. Mac OS X applications also have access to visual properties that make use of Core Image filters.

- Subclassing CALayer and implementing any of the above techniques in a more encapsulated manner.

"Providing Layer Content" (page 33) describes the available techniques for providing the content for a layer. The visual style properties and the order in which they are applied to the content of a layer is discussed in "Layer Style Properties" (page 59).

In addition to the `CALayer` class, the Core Animation class collection provides additional classes that allow applications to display other types of content. The available classes differ slightly between Mac OS X and iPhone OS. The following classes are available on both Mac OS X and iPhone OS:

- `CAScrollLayer` class is a subclass of `CALayer` that simplifies displaying a portion of a layer. The extent of the scrollable area of a `CAScrollLayer` object is defined by the layout of its sublayers. `CAScrollLayer` does not provide keyboard or mouse event-handling, nor does it provide visible scrollers.
- `CATiledLayer` allows the display of large and complex images in incremental stages.

Mac OS X provides these additional classes:

- `CATextLayer` is a convenience class that creates a layer's content from a string or attributed string.
- `CAOpenGLLayer` provides an OpenGL rendering environment. You must subclass this class to provide content using OpenGL. The content can be static or can be updated over time.
- `QCCompositionLayer` (provided by the Quartz Composer framework) animates a Quartz Composer composition as its content.
- `QTMovieLayer` and `QTCaptureLayer` (provided by the QTKit framework) provides playback of QuickTime movies and live video.

iPhone OS adds the following class:

- `CAEAGLLayer` provides an OpenGLES rendering environment.

The `CALayer` class introduces the concept of a **key-value coding compliant container class**–that is, a class that can store arbitrary values, using key-value coding compliant methods, without having to create a subclass. `CALayer` also extends the `NSKeyValueCoding` informal protocol, adding support for default key values and automatic object wrapping for the additional structure types (`CGPoint`, `CGSize`, `CGRect`, `CGAffineTransform` and `CATransform3D`) and provides access to many of the fields of those structures by key path.

`CALayer` also manages the animations and actions that are associated with a layer. Layers receive action triggers in response to layers being inserted and removed from the layer tree, modifications being made to layer properties, or explicit developer requests. These actions typically result in an animation occurring. See "Animation" (page 39) and "Actions" (page 43) for more information.

Animation and Timing Classes

Many of the visual properties of a layer are implicitly animatable. By simply changing the value of an animatable property the layer will automatically animate from the current value to the new value. For example, setting a layer's hidden property to `YES` triggers an animation that causes the layer to gradually fade away. Most animatable properties have an associated default animation which you can easily customize and replace. A complete list of the animatable properties and their default animations are listed in "Animatable Properties" (page 77).

Animatable properties can also be explicitly animated. To explicitly animate a property you create an instance of one of Core Animation's animation classes and specify the required visual effects. An explicit animation doesn't change the value of the property in the layer, it simply animates it in the display.

Core Animation provides animation classes that can animate the entire contents of a layer or selected attributes using both basic animation and key-frame animation. All Core Animation's animation classes descend from the abstract class `CAAnimation`. `CAAnimation` adopts the `CAMediaTiming` protocol which provides the simple duration, speed, and repeat count for an animation. `CAAnimation` also adopts the `CAAction` protocol. This protocol provides a standardized means for starting an animation in response to an action triggered by a layer.

The animation classes also define a timing function that describes the pacing of the animation as a simple Bezier curve. For example, a linear timing function specifies that the animation's pace is even across its duration, while an ease-in timing function causes an animation to slow down as it nears its duration.

Core Animation provides a number of additional abstract and concrete animation classes:

- `CATransition` provides a transition effect that affects the entire layer's content. It fades, pushes, or reveals layer content when animating. The stock transition effects can be extended by providing your own custom Core Image filters.
- `CAAnimation` allows an array of animation objects to be grouped together and run concurrently.
- `CAPropertyAnimation` is an abstract subclass that provides support for animating a layer property specified by a key path.
- `CABasicAnimation` provides simple interpolation for a layer property.
- `CAKeyframeAnimation` provides support for key frame animation. You specify the key path of the layer property to be animated, an array of values that represent the value at each stage of the animation, as well as arrays of key frame times and timing functions. As the animation runs, each value is set in turn using the specified interpolation.

These animation classes are used by both Core Animation and Cocoa Animation proxies. "Animation" (page 39) describes the classes as they pertain to Core Animation, *Animation Types and Timing Programming Guide* contains a more in-depth exploration of their capabilities.

Layout Manager Classes

Application Kit view classes provide the classic "struts and springs" model of positioning layers relative to their superlayer. While layers support this model, Core Animation on Mac OS X also provides a more flexible layout manager mechanism that allows developers to write their own layout managers.

Core Animation's `CAConstraint` class is a layout manager that arranges sublayers using a set of constraints that you specify. Each constraint (encapsulated by instances of the `CAConstraint` class) describes the relationship of one geometric attribute of a layer (the left, right, top, or bottom edge or the horizontal or vertical center) in relation to a geometric attribute of one of its sibling layers or its superlayer.

Layout managers in general, and the constraint layout manager are discussed in "Laying Out Core Animation Layers" (page 51)

Transaction Management Classes

Every modification to an animatable property of a layer must be part of a transaction. `CATransaction` is the Core Animation class responsible for batching multiple animation operations into atomic updates to the display. Nested transactions are supported.

Core Animation supports two types of transactions: implicit transactions and explicit transactions. Implicit transactions are created automatically when an animatable property of a layer is modified by a thread without an active transaction and are committed automatically when the thread's run-loop next iterates. Explicit transactions occur when the application sends the `CATransaction` class a begin message before modifying the layer, and a commit message afterwards.

Transaction management is discussed in "Transactions" (page 47).

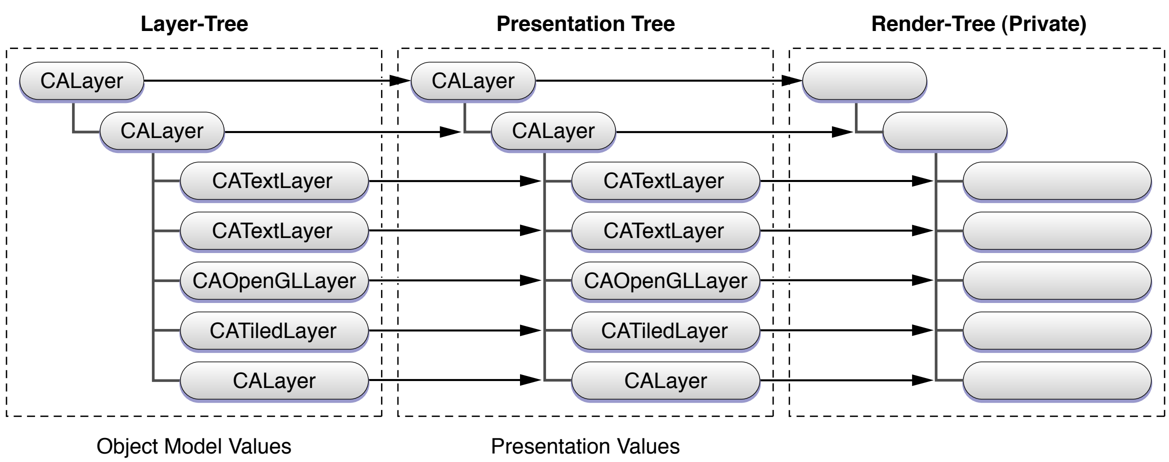
Core Animation Rendering Architecture

While there are obvious similarities between Core Animation layers and Cocoa views the biggest conceptual divergence is that layers do not render directly to the screen.

Where `NSView` and `UIView` are clearly view objects in the model-view-controller design pattern, Core Animation layers are actually model objects. They encapsulate geometry, timing and visual properties, and they provide the content that is displayed, but the actual display is not the layer's responsibility.

Each visible layer tree is backed by two corresponding trees: a presentation tree and a render tree. Figure 1 shows an example layer-tree using the Core Animation layer classes available in Mac OS X.

Figure 1 Core Animation Rendering Architecture



The layer tree contains the object model values for each layer. These are the values you set when you assign a value to a layer property.

The presentation tree contains the values that are currently being presented to the user as an animation takes place. For example, setting a new value for the `backgroundColor` of a layer immediately changes the value in the layer tree. However, the `backgroundColor` value in the corresponding layer in the presentation tree will be updated with the interpolated colors as they are displayed to the user.

The render-tree uses the value in the presentation-tree when rendering the layer. The render-tree is responsible for performing the compositing operations independent of application activity; rendering is done in a separate process or thread so that it has minimal impact on the application's run loop.

You can query an instance of `CALayer` for its corresponding presentation layer while an animation transaction is in process. This is most useful if you intend to change the current animation and want to begin the new animation from the currently displayed state.

Layer Geometry and Transforms

This chapter describes the components of a layer's geometry, how they interrelate, and how transform matrices can produce complex visual effects.

Layer Coordinate System

The layer's location and size are expressed using the same coordinate system that the Quartz graphics environment uses. By default, the graphics environment origin (0.0,0.0) is located in the lower left, and values are specified as floating-point numbers that increase up and to the right in coordinate system units. The coordinate system units, the unit square, is the size of a 1.0 by 1.0 rectangle.

Every layer instance defines and maintains its own coordinate system, and all sublayers are positioned, and drawing is done, relative to this coordinate system. Methods are provided to convert points, rectangles and sizes from one layer coordinate system to another. A layer's coordinate system should be considered the base coordinate system for all the content of the layer, including its sublayers.

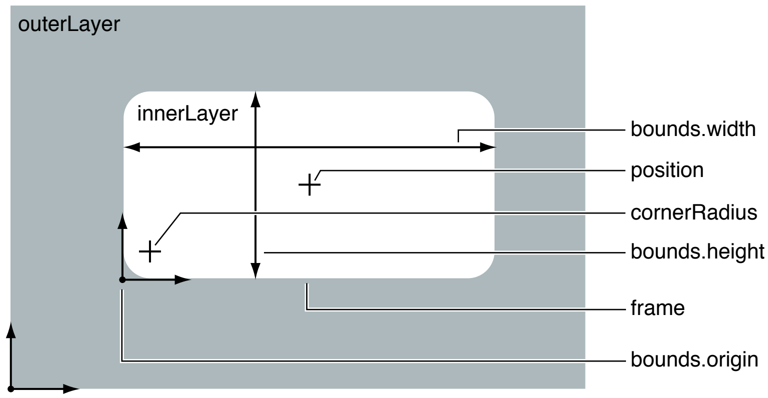
> **iPhone OS Note:** The default root layer of a `UIView` instance uses a flipped coordinate system that matches the default coordinate system of a `UIView` instance–the origin is in the top-left and values increase down and to the right. Layers created by instantiating `CALayer` directly use the standard Core Animation coordinate system.

Specifying a Layer's Geometry

While layers and the layer-tree are analogous to Cocoa views and the view hierarchy in many ways, how a layer's geometry is specified is different, and often simpler, manner. All of a layer's geometric properties, including the layer's transform matrices, can be implicitly and explicitly animated.

Figure 1 shows the properties used to specify a layer's geometry in context.

Figure 1 CALayer geometry properties



The `position` property is a `CGPoint` that specifies the position of the layer relative to its superlayer, and is expressed in the superlayer's coordinate system.

The `bounds` property is a `CGRect` that provides the size of the layer (`bounds.size`) and the origin (`bounds.origin`). The bounds origin is used as the origin of the graphics context when you override a layer's drawing methods.

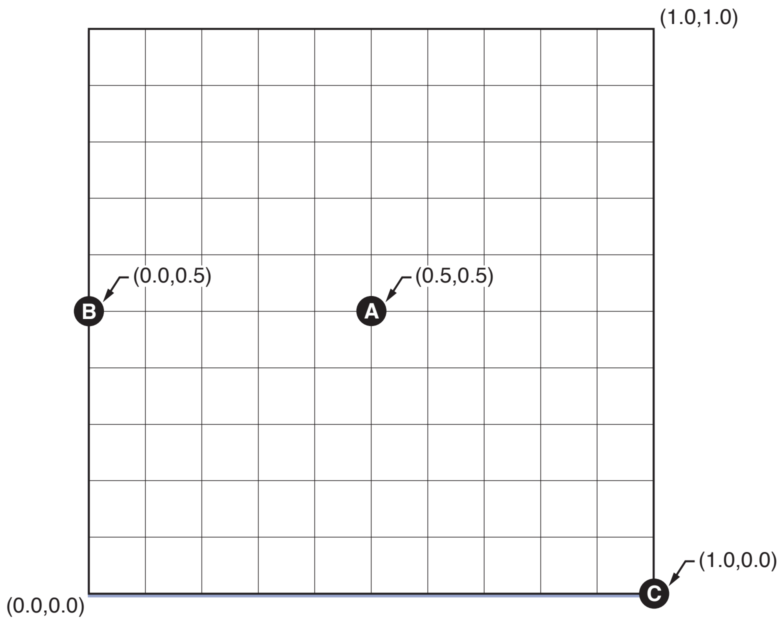
Layers have an implicit `frame` that is a function of the `position`, `bounds`, `anchorPoint`, and `transform` properties. Setting a new frame rectangle changes the layer's `position` and `bounds` properties appropriately, but the frame itself is not stored. When a new frame rectangle is specified the bounds origin is undisturbed, while the bounds size is set to the size of the frame. The layer's position is set to the proper location relative to the anchor point. When you get the `frame` property value, it is calculated relative to the `position`, `bounds`, and `anchorPoint` properties.

The `anchorPoint` property is a `CGPoint` that specifies a location within the bounds of a layer that corresponds with the position coordinate. The anchor point specifies how the bounds are positioned relative to the position property, as well as serving as the point that transforms are applied around. It is expressed in the unit coordinate system-the lower left of the layer bounds is 0.0,0.0, and the upper right is 1.0,1.0.

When you specify the frame of a layer, `position` is set relative to the anchor point. When you specify the position of the layer, `bounds` is set relative to the anchor point.

Figure 2 shows three example values for an anchor point.

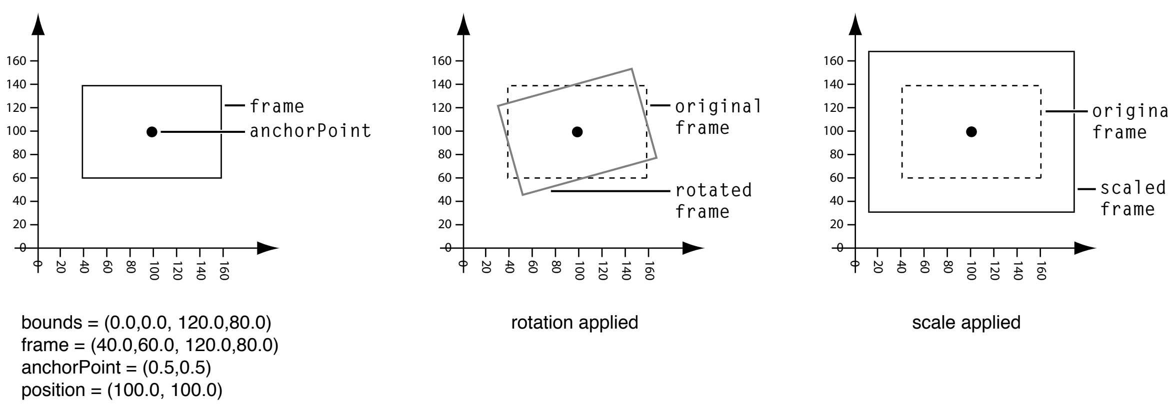
Figure 2 Three anchorPoint values



The default value for `anchorPoint` is (0.5,0.5) which corresponds to the center of the layer's bounds (shown as point A in Figure 2.) Point B shows the position of an anchor point set to (0.0,0.5). Finally, point C (1.0,0.0) causes specifies that the layer's `position` is set to the bottom right corner of the frame.

The relationship of the `frame`, `bounds`, `position`, and `anchorPoint` properties is shown in Figure 3.

Figure 3 Layer Origin of (0.5,0.5)

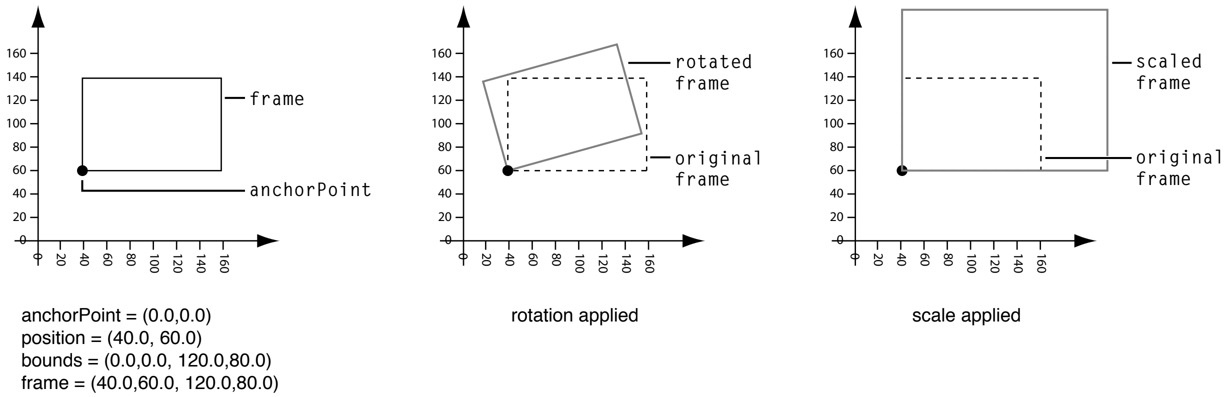


In this example the `anchorPoint` is set to the default value of (0.5,0.5), which corresponds to the center of the layer. The `position` of the layer is set to (100.0,100.0), and the `bounds` is set to the rectangle (0.0, 0.0, 120.0, 80.0). This causes the frame property to be calculated as (40.0, 60.0, 120.0, 80.0).

If you created a new layer, and set only the layer's frame property to (40.0, 60.0, 120.0, 80.0), the `position` property would be automatically set to (100.0,100.0), and the bounds property to (0.0, 0.0, 120.0, 80.0).

Figure 4 shows a layer with the same `frame` rectangle as the layer in Figure 3. However, in this case the `anchorPoint` of the layer is set to (0.0,0.0), which corresponds with the bottom left corner of the layer.

Figure 4 Layer Origin of (0.0,0.0)



With the frame set to (40.0, 60.0, 120.0, 80.0), the value of the `bounds` property is the same, but the value of the `position` property has changed.

Another aspect of layer geometry that differs from Cocoa views is that you can specify a radius that is used to round the corners of the layer. The `cornerRadius` property specifies a radius the layer uses when drawing content, clipping sublayers, and drawing the border and shadow.

The `zPosition` property specifies the z-axis component of the layer's position. The `zPosition` is intended to be used to set the visual position of the layer relative to its sibling layers. It should not be used to specify the order of layer siblings, instead reorder the layer in the sublayer array.

Transforming a Layer's Geometry

Once established, you can transform a layer's geometry using matrix transformations. The `Transform` data structure defines a homogenous three-dimensional transform (a 4 by 4 matrix of `CGFloat` values) that is used to rotate, scale, offset, skew, and apply perspective transformations to a layer.

Two layer properties specify transform matrices: `transform` and `sublayerTransform`. The matrix specified by the `transform` property is applied to the layer and its sublayers relative to the layer's `anchorPoint`. Figure 3 shows how rotation and scaling transforms affect a layer when using an anchorPoint of (0.5,0.5), the default value. Figure 4 shows how the same transform matrices affect a layer when an anchorPoint of (0.0,0.0). The matrix specified by the `sublayerTransform` property is applied only to the layer's sublayers, rather than to the layer itself.

You create and modify `CATransform3D` data structures in one of the following ways:

- using the `CATransform3D` functions
- modifying the data structure members directly
- using key-value coding and key paths.

The constant `CATransform3DIdentity` is the identity matrix, a matrix that has no scale, rotation, skewing, or perspective applied. Applying the identity matrix to a layer causes it to be displayed with its default geometry.

Transform Functions

The transform functions available in Core Animation operate on matrices. You can use these functions (shown in Table 1) to construct a matrix that you later apply to a layer or its sublayers by modifying the transform or sublayerTransform properties respectively. The transform functions either operate on, or return, a `CATransform3D` data structure. This enables you to construct simple or complex transforms that you can readily reuse.

Table 1 CATransform3D transform functions for translation, rotation, and scaling

Function	Use
`CATransform3DMakeTranslation`	Returns a transform that translates by '(tx, ty, tz)'. t' = [1 0 0 0; 0 1 0 0; 0 0 1 0; tx ty tz 1].
`CATransform3DTranslate`	Translate 't' by '(tx, ty, tz)' and return the result: * t' = translate(tx, ty, tz) * t.
`CATransform3DMakeScale`	Returns a transform that scales by `(sx, sy, sz)': * t' = [sx 0 0 0; 0 sy 0 0; 0 0 sz 0; 0 0 0 1].
`CATransform3DScale`	Scale 't' by '(sx, sy, sz)' and return the result: * t' = scale(sx, sy, sz) * t.
`CATransform3DMakeRotation`	Returns a transform that rotates by 'angle' radians about the vector '(x, y, z)'. If the vector has length zero the identity transform is returned.
`CATransform3DRotate`	Rotate 't' by 'angle' radians about the vector '(x, y, z)' and return the result. t' = rotation(angle, x, y, z) * t.

The angles of rotation is specified in radians rather than degrees. The following functions allow you to convert between radians and degrees.

```
CGFloat DegreesToRadians(CGFloat degrees) {return degrees * M_PI / 180;};
CGFloat RadiansToDegrees(CGFloat radians) {return radians * 180 / M_PI;};
```

Core Animation provides a transform function that inverts a matrix, `CATransform3DInvert`. Inversion is generally used to provide reverse transformation of points within transformed objects. Inversion can be useful when you need to recover a value that has been transformed by a matrix: invert the matrix, and multiply the value by the inverted matrix, and the result is the original value.

Functions are also provided that allow you to convert a `CATransform3D` matrix to a `CGAffineTransform` matrix, if the `CATransform3D` matrix can be expressed as such.

Table 2 CATransform3D transform functions for CGAffineTransform conversion

Function	Use
`CATransform3DMakeAffineTransform`	Returns a `CATransform3D` with the same effect as the passed affine transform.
`CATransform3DIsAffine`	Returns `YES` if the passed `CATransform3D` can be exactly represented an affine transform.
`CATransform3DGetAffineTransform`	Returns the affine transform represented by the passed `CATransform3D`.

Functions are provided for comparing transform matrices for equality with the identity matrix, or another transform matrix.

Table 3 CATransform3D transform functions for testing equality

Function	Use
`CATransform3DIsIdentity`	Returns `YES` if the transform is the identity transform.
`CATransform3DEqualToTransform`	Returns `YES` if the two transforms are exactly equal..

Modifying the Transform Data Structure

You can modify the value of any of the `CATransform3D` data structure members as you would any other data structure. Listing 1 contains the definition of the `CATransform3D` data structure, the structure members are shown in their corresponding matrix positions.

Listing 1 CATransform3D structure

```
struct CATransform3D
{
  CGFloat m11, m12, m13, m14;
  CGFloat m21, m22, m23, m24;
  CGFloat m31, m32, m33, m34;
  CGFloat m41, m42, m43, m44;
};

typedef struct CATransform3D CATransform3D;
```

The example in Listing 2 illustrates how to configure a `CATransform3D` as a perspective transform.

Listing 2 Modifying the CATransform3D data structure directly

```
 CATransform3D aTransform = CATransform3DIdentity;
// the value of zDistance affects the sharpness of the transform.
zDistance = 850;
aTransform.m34 = 1.0 / -zDistance;
```

Modifying a Transform Using Key Paths

Core Animation extends the key-value coding protocol to allow getting and setting of the commonly values of a layer's `CATransform3D` matrix through key paths. Table 4 describes the key paths for which a layer's `transform` and `sublayerTransform` properties are key-value coding and observing compliant.

Table 4 CATransform3D key paths

Field Key Path	Description
`rotation.x`	The rotation, in radians, in the x axis.
`rotation.y`	The rotation, in radians, in the y axis.
`rotation.z`	The rotation, in radians, in the z axis.
`rotation`	The rotation, in radians, in the z axis. This is identical to setting the `rotation.z` field.
`scale.x`	Scale factor for the x axis.
`scale.y`	Scale factor for the y axis.
`scale.z`	Scale factor for the z axis.
`scale`	Average of all three scale factors.
`translation.x`	Translate in the x axis.
`translation.y`	Translate in the y axis.
`translation.z`	Translate in the z axis.
`translation`	Translate in the x and y axis. Value is an NSSize or CGSize.

You can not specify a structure field key path using Objective-C 2.0 properties. This will not work:

```
myLayer.transform.rotation.x=0;
```

Instead you must use `setValue:forKeyPath:` or `valueForKeyPath:` as shown below:

```
[myLayer setValue:[NSNumber numberWithInt:0] forKeyPath:@"transform.rotation.x"];
```

Layer-Tree Hierarchy

Along with their own direct responsibilities for providing visual content and managing animations, layers also act as containers for other layers, creating a layer hierarchy.

This chapter describes the layer hierarchy and how you manipulate layers within that hierarchy.

What Is a Layer-Tree Hierarchy?

The layer-tree is the Core Animation equivalent of the Cocoa view hierarchy. Just as an instance of `NSView` or `UIView` has superview and subviews, a Core Animation layer has a superlayer and sublayers. The layer-tree provides many of the same benefits as the view hierarchy:

- Complex interfaces can be assembled using simpler layers, avoiding monolithic and complex subclassing. Layers are well suited to this type of 'stacking' due to their complex compositing capabilities.
- Each layer declares its own coordinate system relative to its superlayer's coordinate system. When a layer is transformed, its sublayers are transformed within it.
- A layer-tree is dynamic. It can be reconfigured as an application runs. Layers can be created, added as a sublayer first of one layer, then of another, and removed from the layer-tree.

Displaying Layers in Views

Core Animation doesn't provide a means for actually displaying layers in a window, they must be hosted by a view. When paired with a view, the view must provide event-handling for the underlying layers, while the layers provide display of the content.

The view system in iPhone OS is built directly on top of Core Animation layers. Every instance of UIView automatically creates an instance of a `CALayer` class and sets it as the value of the view's `layer` property. To display custom layer content in a UIView instance you simply add the layers as sublayers of the view's layer.

On Mac OS X you must configure an `NSView` instance in such a way that it can host a layer. To display the root layer of a layer tree, you set a view's layer and then configure the view to use layers as shown in Table 2.

Listing 1 Inserting a layer into a view

```
// theView is an existing view in a window
// theRootLayer is the root layer of a layer tree

[theView setLayer: theRootLayer];
[theView setWantsLayer:YES];
```

Adding and Removing Layers from a Hierarchy

Simply instantiating a layer instance doesn't insert it into a layer-tree. Instead you add, insert, replace, and remove layers from the layer-tree using the methods described in .Table 1.

Table 1 Layer-tree management methods.

Method	Result
`addSublayer:`	Appends the layer to the receiver's sublayers array.
`insertSublayer: atIndex:`	Inserts the layer as a sublayer of the receiver at the specified index.
`insertSublayer: below:`	Inserts the layer into the receiver's sublayers array, below the specified sublayer.
`insertSublayer: above:`	Inserts the layer into the receiver's sublayers array, above the specified sublayer.
`removeFromSuperlayer`	Removes the receiver from the sublayers array or mask property of the receiver's superlayer.
`replaceSublayer: with:`	Replaces the layer in the receiver's sublayers array with the specified new layer.

You can also set the sublayers of a layer using an array of layers, and setting the intended superlayer's sublayers property. When setting the sublayers property to an array populated with layer objects you must ensure that the layers have had their superlayer set to `nil`.

By default, inserting and removing layers from a visible layer-tree triggers an animation. When a layer is added as a sublayer the animation returned by the parent layer for the action identifier `kCAOnOrderIn` is triggered. When a layer is removed from a layer's sublayers the animation returned by the parent layer for the action identifier `kCAOnOrderOut` is triggered. Replacing a layer in a sublayer causes the animation returned by the parent layer for the action identifier `kCATransition` to be triggered. You can disable animation while manipulating the layer-tree, or alter the animation used for any of the action identifiers.

Repositioning and Resizing Layers

After a layer has been created, you can move and resize it programmatically simply by changing the value of the layer's geometry properties: `frame`, `bounds`, `position`, `anchorPoint`, or `zPosition`.

If a layer's `needsDisplayOnBoundsChange` property is YES, the layer's content is recached when the layer's bounds changes. By default the `needsDisplayOnBoundsChange` property is no.

By default, setting the `frame`, `bounds`, `position`, `anchorPoint`, and `zPosition` properties causes the layer to animate the new values.

Autoresizing Layers

`CALayer` provides a mechanism for automatically moving and resizing sublayers in response to their superlayer being moved or resized. In many cases simply configuring the autoresizing mask for a layer provides the appropriate behavior for an application.

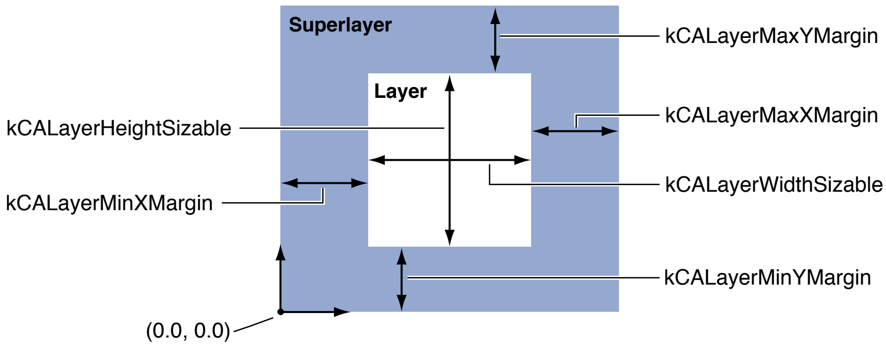
A layer's autoresizing mask is specified by combining the `CAAutoresizingMask` constants using the bitwise `OR` operator and the layer's `autoresizingMask` property to the resulting value. Table 2 shows each mask constant and how it effects the layer's resizing behavior.

Table 2 Autoresizing mask values and descriptions

Autoresizing Mask	Description
`kCALayerHeightSizable`	If set, the layer's height changes proportionally to the change in the superlayer's height. Otherwise, the layer's height does not change relative to the superlayer's height.
`kCALayerWidthSizable`	If set, the layer's width changes proportionally to the change in the superlayer's width. Otherwise, the layer's width does not change relative to the superlayer's width.
`kCALayerMinXMargin`	If set, the layer's left edge is repositioned proportionally to the change in the superlayer's width. Otherwise, the layer's left edge remains in the same position relative to the superlayer's left edge.
`kCALayerMaxXMargin`	If set, the layer's right edge is repositioned proportionally to the change in the superlayer's width. Otherwise, the layer's right edge remains in the same position relative to the superlayer.
`kCALayerMinYMargin`	If set, the layer's top edge is repositioned proportionally to the change in the superlayer's height. Otherwise, the layer's top edge remains in the same position relative to the superlayer.
`kCALayerMaxYMargin`	If set, the layer's bottom edge is repositioned proportional to the change in the superlayer's height. Otherwise, the layer's bottom edge remains in the same position relative to the superlayer.

For example, to keep a layer in the lower-left corner of its superlayer, you use the mask `kCALayerMaxXMargin | kCALayerMaxYMargin`. When more than one aspect along an axis is made flexible, the resize amount is distributed evenly among them. Figure 1 provides a graphical representation of the position of the constant values.

Figure 1 Layer autoresizing mask constants



When one of these constants is omitted, the layer's layout is fixed in that aspect; when a constant is included in the mask the layer's layout is flexible in that aspect.

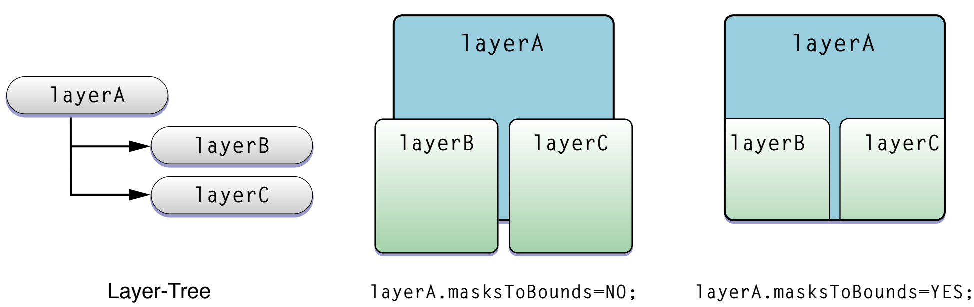
A subclass can override the `CALayer` methods `resizeSublayersWithOldSize:` and `resizeWithOldSuperlayerSize:` to customize the autoresizing behavior for a layer. A layers `resizeSublayersWithOldSize:` method is invoked automatically by a layer whenever bounds property changes, and sends a `resizeWithOldSuperlayerSize:` message to each sublayer. Each sublayer compares the old bounds size to the new size and adjusts its position and size according to its autoresize mask.

Clipping Sublayers

When subviews of a Cocoa view lie outside of the parent view's bounds, the views are clipped to the parent view. Layers remove this limitation, allowing sublayers to be displayed in their entirety, regardless of their position relative to the parent layer.

The value of a layer's `masksToBounds` property determines if sublayers are clipped to the parent. The default value of the `masksToBounds` property is `NO`, which prevents sublayers from being clipped to the parent. Figure 2 shows the results of setting the masksToBounds for `layerA` and how it will affect the display of `layerB` and `layerC`.

Figure 2 Example Values of the masksToBounds property



Providing Layer Content

Providing CALayer Content

When using Cocoa views you must subclass `NSView` or `UIView` and implement `drawRect:` in order to display anything. However `CALayer` instances can often be used directly, without requiring you to create a subclass. Because `CALayer` is a key-value coding compliant container class, that is you can store arbitrary values in any instance, subclassing can often be avoided entirely.

You specify the content of a `CALayer` instance in one of the following ways:

- Explicitly set the `contents` property of a layer instance using a `CGImageRef` that contains the content image.
- Specify a delegate that provides, or draws, the content.
- Subclass `CALayer` and override one of the display methods.

Setting the Contents Property

A layer's content image is specified by `contents` property to a `CGImageRef`. This can be done from another object when the layer is created (as shown in Table 3) or at any other time.

Listing 4 Setting a layer's contents property

```
CALayer *theLayer;

// create the layer and set the bounds and position
theLayer=[CALayer layer];
theLayer.position=CGPointMake(50.0f,50.0f);
theLayer.bounds=CGRectMake(0.0f,0.0f,100.0f,100.0f);

// set the contents property to a CGImageRef
// specified by theImage (loaded elsewhere)
theLayer.contents=theImage;
```

Using a Delegate to Provide Content

You can draw content for your layer, or better encapsulate setting the layer's content image by creating a delegate class that implements one of the following methods: `displayLayer:` or `drawLayer:inContext:`.

Implementing a delegate method to draw the content does not automatically cause the layer to draw using that implementation. Instead, you must explicitly tell a layer instance to re-cache the content, either by sending it a `setNeedsDisplay` or `setNeedsDisplayInRect:` message, or by setting its `needsDisplayOnBoundsChange` property to `YES`.

Delegates that implement the `displayLayer:` method can determine which image should be displayed for the specified layer, and then set that layer's `contents` property accordingly. The example in implementation of `displayLayer:` in "Layer Coordinate System" sets the `contents` property of `theLayer` depending on the value of the state key. Subclassing is not required to store the `state` value, as the `CALayer` instance acts as a key-value coding container.

Listing 5 Example implementation of the delegate method displayLayer:

```
- (void)displayLayer:(CALayer *)theLayer
{
    // check the value of the layer's state key
    if ([[theLayer valueForKey:@"state"] boolValue])
    {
        // display the yes image
        theLayer.contents=[someHelperObject loadStateYesImage];
    }
    else {
        // display the no image
        theLayer.contents=[someHelperObject loadStateNoImage];
    }
}
```

If you must draw the layer's content rather than loading it from an image, you implement the `drawLayer:inContext:` delegate method. The delegate is passed the layer for which content is required and a `CGContextRef` to draw the content in.

The example in implementation of `drawLayer:inContext::` in "Specifying a Layer's Geometry" draws a path in using the `lineWidth` key value returned by `theLayer`.

Listing 6 Example implementation of the delegate method drawLayer:inContext:

```
- (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,
                          [[theLayer valueForKey:@"lineWidth"] floatValue]);
    CGContextStrokePath(theContext);

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

Providing CALayer Content by Subclassing

Although often unnecessary, you can subclass `CALayer` and override the drawing and display methods directly. This is typically done when your layer requires custom behavior that can't be provided though delegation.

A subclass can override the `CALayer` display method and set the layer's contents to the appropriate image. The example in "Transforming a Layer's Geometry"provides the same functionality as the delegate implementation of `displayLayer:` in "Layer Coordinate System." The difference is that the subclass defines `state` as instance property, rather than depending on the key-value coding container ability of `CALayer`.

Listing 7 Example override of the CALayer display method

```
- (void)display
{
    // check the value of the layer's state key
    if (self.state)
    {
        // display the yes image
        self.contents=[someHelperObject loadStateYesImage];
    }
    else {
        // display the no image
        self.contents=[someHelperObject loadStateNoImage];
    }
}
```

`CALayer` subclasses can draw the layer's content into a graphics context by overriding `drawInContext:`. The example in "Modifying the Transform Data Structure" produces the same content image as the delegate implementation in "Specifying a Layer's Geometry." Again, the only difference in the implementation is that `lineWidth` and `lineColor` are now declared as instance properties of the subclass.

Listing 8 Example override of the CALayer drawInContext: method

```
- (void)drawInContext:(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,
                          self.lineWidth);
    CGContextSetStrokeColorWithColor(theContext,
                                     self.lineColor);
    CGContextStrokePath(theContext);
    CFRelease(thePath);
```

```
}
```

Subclassing `CALayer` and implementing one of the drawing methods does not automatically cause drawing to occur. You must explicitly cause the instance to re-cache the content, either by sending it a `setNeedsDisplay` or `setNeedsDisplayInRect:` message, or by setting its `needsDisplayOnBoundsChange` property to `YES`.

Positioning Content Within a Layer

The `CALayer` property `contentsGravity` allows you to position and scale the layer's `contents` image within the layer bounds. By default, the content image fills the layer's bounds entirely, ignoring the natural aspect ratio of the image.

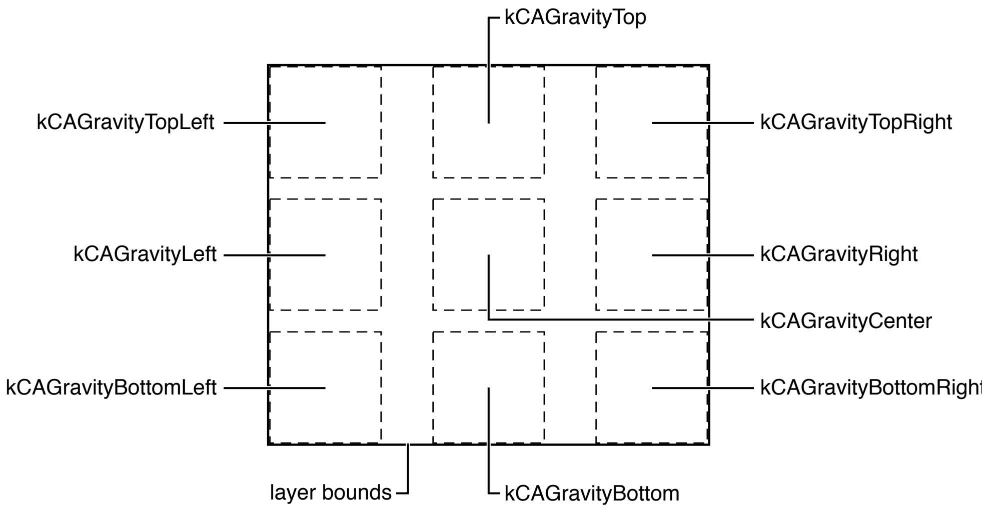
Using the `contentsGravity` positioning constants you can specify that the image is placed along any of the layer's edges, in the layer's corners, or centered within the layer's bounds. "Specifying a Layer's Geometry" lists the positioning constants and their corresponding positions.

Table 7 Positioning constants for a layer's contentsGravity property

Position constant	Description
`kCAGravityTopLeft`	Positions the content image in the top left corner of the layer.
`kCAGravityTop`	Positions the content image horizontally centered along the top edge of the layer.
`kCAGravityTopRight`	Positions the content image in the top right corner of the layer.
`kCAGravityLeft`	Positions the content image vertically centered on the left edge of the layer.
`kCAGravityCenter`	Positions the content image at the center of the layer.
`kCAGravityRight`	Positions the content image vertically centered on the right edge of the layer.
`kCAGravityBottomLeft`	Positions the content image in the bottom left corner of the layer.
`kCAGravityBottom`	Positions the content image centered along the bottom edge of the layer.
`kCAGravityBottomRight`	Positions the content image in the top right corner of the layer.

"Layer Coordinate System" shows the supported content positions and their corresponding constants.

Figure 9 Position constants for a layer's contentsGravity property



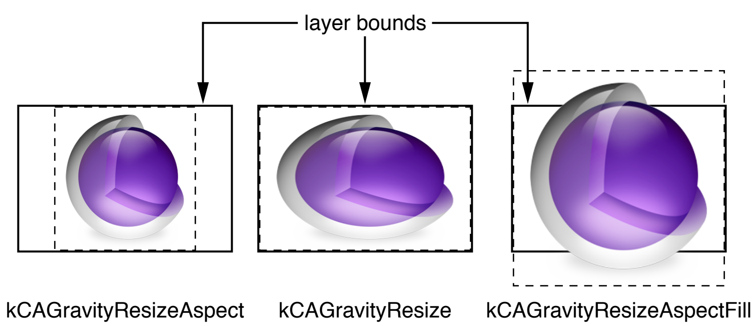
The content image can be scaled up, or down, by setting the `contentsGravity` property to one of the gravity constants listed in "Transform Functions"

Table 8 Scaling constants for a layer's contentsGravity property

Scaling constant	Description
`kCAGravityResize`	Resize the content image to completely fill the layer bounds, potentially ignoring the natural aspect of the content. This is the default.
`kCAGravityResizeAspect`	Resize the content image to scale such that it is displayed as large as possible within the layer bounds, yet still retains its natural aspect.

"Transforming a Layer's Geometry" illustrates how a square image is resized to fit within a rectangular layer bounds using the resizing modes.

Figure 10 Scaling constants for a layer's contentsGravity property



Animation

Animation is a key element of today's user interfaces. When using Core Animation animation is completely automatic. There are no animation loops or timers. Your application is not responsible for frame by frame drawing, or tracking the current state of your animation. The animation occurs automatically in a separate thread, without further interaction with your application.

This chapter provides an overview of the animation classes, and describes how to create both implicit and explicit animations.

Animation Classes and Timing

Core Animation provides an expressive set of animation classes you can use in your application:

- `CABasicAnimation` provides simple interpolation between values for a layer property.
- `CAKeyframeAnimation` provides support for key frame animation. You specify the key path of the layer property to be animated, an array of values that represent the value at each stage of the animation, as well as arrays of key frame times and timing functions. As the animation runs, each value is set in turn using the specified interpolation.
- `CATransition` provides a transition effect that affects the entire layer's content. It fades, pushes, or reveals layer content when animating. The stock transition effects can be extended by providing your own custom Core Image filters.
- `CAAnimationGroup` allows an array of animation objects to be grouped together and run concurrently.

In addition to specifying the type of animation to perform, you must also specify the duration of the animation, the pacing (how the interpolated values are distributed across the duration), if the animation is to repeat and how many times, whether it should automatically reverse when each cycle is completed, and its visual state when the animation is completed. The animation classes and the `CAMediaTiming` protocol provides all this functionality and more.

`CAAnimation` and its subclasses and the timing protocols are shared by both Core Animation and the Cocoa Animation Proxy functionality. The classes are described in detail in *Animation Types and Timing Programming Guide*.

Implicit Animation

Core Animation's implicit animation model assumes that all changes to animatable layer properties should be gradual and asynchronous. Dynamically animated scenes can be achieved without ever explicitly animating layers. Changing the value of an animatable layer property causes the layer to implicitly animate the change from the old value to the new value. While an animation is in-flight, setting a new target value causes the animation transition to the new target value from its current state.

Listing 1 shows how simple it is to trigger an implicit animation that animates a layer from its current position to a new position.

Listing 1 Implicitly animating a layer's position property

```
// assume that the layer is current positioned at (100.0,100.0)
theLayer.position=CGPointMake(500.0,500.0);
```

You can implicitly animate a single layer property at a time, or many. You can also implicitly animate several layers simultaneously. The code in Listing 2 causes four implicit animations to occur simultaneously.

Listing 2 Implicitly animating multiple properties of multiple layers

```
// animate theLayer's opacity to 0 while moving it
// further away in the layer
theLayer.opacity=0.0;
theLayer.zPosition=-100;

// animate anotherLayer's opacity to 1
//  while moving it closer in the layer
anotherLayer.opacity=1.0;
anotherLayer.zPosition=100.0;
```

Implicit animations use the duration specified in the default animation for the property, unless the duration has been overridden in an implicit or explicit transaction. See "Overriding the Duration of Implied Animations" (page 48) for more information.

Explicit Animation

Core Animation also supports an explicit animation model. The explicit animation model requires that you create an animation object, and set start and end values. An explicit animation won't start until you apply the animation to a layer. The code fragment in Listing 3 creates an explicit animation that transitions a layer's opacity from fully opaque to fully transparent, and back over a 3 second duration. The animation doesn't begin until it is added to the layer.

Listing 3 Explicit animation

```
CABasicAnimation *theAnimation;

theAnimation=[CABasicAnimation animationWithKeyPath:@"opacity"];
theAnimation.duration=3.0;
theAnimation.repeatCount=2;
theAnimation.autoreverses=YES;
theAnimation.fromValue=[NSNumber numberWithFloat:1.0];
theAnimation.toValue=[NSNumber numberWithFloat:0.0];
[theLayer addAnimation:theAnimation forKey:@"animateOpacity"];
```

Explicit animations are especially useful when creating animations that run continuously. Listing 4 shows how to create an explicit animation that applies a CoreImage bloom filter to a layer, animating its intensity. This causes the "selection layer" to pulse, drawing the user's attention.

Listing 4 Continuous explicit animation example

```
// The selection layer will pulse continuously.
// This is accomplished by setting a bloom filter on the layer

// create the filter and set its default values
CIFilter *filter = [CIFilter filterWithName:@"CIBloom"];
[filter setDefaults];
[filter setValue:[NSNumber numberWithFloat:5.0] forKey:@"inputRadius"];

// name the filter so we can use the keypath to animate the inputIntensity
// attribute of the filter
[filter setName:@"pulseFilter"];

// set the filter to the selection layer's filters
[selectionLayer setFilters:[NSArray arrayWithObject:filter]];

// create the animation that will handle the pulsing.
CABasicAnimation* pulseAnimation = [CABasicAnimation animation];

// the attribute we want to animate is the inputIntensity
// of the pulseFilter
pulseAnimation.keyPath = @"filters.pulseFilter.inputIntensity";

// we want it to animate from the value 0 to 1
pulseAnimation.fromValue = [NSNumber numberWithFloat: 0.0];
pulseAnimation.toValue = [NSNumber numberWithFloat: 1.5];

// over a one second duration, and run an infinite
// number of times
pulseAnimation.duration = 1.0;
pulseAnimation.repeatCount = 1e100f;

// we want it to fade on, and fade off, so it needs to
// automatically autoreverse.. this causes the intensity
// input to go from 0 to 1 to 0
pulseAnimation.autoreverses = YES;

// use a timing curve of easy in, easy out..
pulseAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
kCAMediaTimingFunctionEaseInEaseOut];

// add the animation to the selection layer. This causes
// it to begin animating. We'll use pulseAnimation as the
// animation key name
[selectionLayer addAnimation:pulseAnimation forKey:@"pulseAnimation"];
```

Starting and Stopping Explicit Animations

You start an explicit animation by sending a `addAnimation:forKey:` message to the target layer, passing the animation and an identifier as parameters. Once added to the target layer the explicit animation will run until the animation completes, or it is removed from the layer. The identifier used to add an animation to a layer is also used to stop it by invoking `removeAnimationForKey:`. You can stop all animations for a layer by sending the layer a `removeAllAnimations` message.

Actions

Layer actions are triggered in response to: layers being inserted and removed from the layer-tree, the value of layer properties being modified, or explicit application requests. Typically, action triggers result in an animation being displayed.

What are Actions?

An action object is an object that responds to an action identifier via the `CAAction` protocol. Action identifiers are named using standard dot-separated key paths. A layer is responsible for mapping action identifiers to the appropriate action object. When the action object for the identifier is located that object is sent the message defined by the `CAAction` protocol.

The `CALayer` class provides default action objects–instances of `CAAnimation`, a `CAAction` protocol compliant class–for all animatable layer properties. `CALayer` also defines the following action triggers that are not linked directly to properties, as well as the action identifiers in Table 1.

Table 1 Action triggers and their corresponding identifiers

Trigger	Action identifier
A layer is inserted into a visible layer-tree, or the `hidden` property is set to `NO`.	The action identifier constant `kCAOnOrderIn`.
A layer is removed from a visible layer-tree, or the `hidden` property is set to `YES`.	The action identifier constant `kCAOnOrderOut`.
A layer replaces an existing layer in a visible layer tree using `replaceSublayer: with:`.	The action identifier constant `kCATransition`.

Action Object Search Pattern

When an action trigger occurs, the layer's `actionForKey:` method is invoked. This method returns an action object that corresponds to the action identifier passed as the parameter, or `nil` if no action object exists.

When the `CALayer` implementation of `actionForKey:` is invoked for an identifier the following search pattern is used:

1. If the layer has a delegate, and it implements the method `actionForLayer:forKey:` it is invoked, passing the layer, and the action identifier as parameters. The delegate's `actionForLayer:forKey:` implementation should respond as follows:

   - Return an action object that corresponds to the action identifier.

- Return `nil` if it doesn't handle the action identifier.
- Return `NSNull` if it doesn't handle the action identifier and the search should be terminated.

2. The layer's `actions` dictionary is searched for an object that corresponds to the action identifier.
3. The layer's `style` property is searched for an `actions` dictionary that contains the identifier.
4. The layer's class is sent a `defaultActionForKey:` message. It will return an action object corresponding to the identifier, or `nil` if not found.

CAAction Protocol

The `CAAction` protocol defines how action objects are invoked. Classes that implement the `CAAction` protocol have a method with the signature `runActionForKey:object:arguments:`.

When the action object receives the `runActionForKey:object:arguments:` message it is passed the action identifier, the layer on which the action should occur, and an optional dictionary of parameters.

Typically, action objects are an instance of a CAAnimation subclass, which implements the `CAAction` protocol. You can, however, return an instance of any class that implements the protocol. When that instance receives the `runActionForKey:object:arguments:` message it should respond by performing its action.

When an instance of `CAAnimation` receives the `runActionForKey:object:arguments:` message it responds by adding itself to the layer's animations, causing the animation to run (see Listing 1 (page 44)).

Listing 1 runActionForKey:object:arguments: implementation that initiates an animation

```
- (void)runActionForKey:(NSString *)key
                object:(id)anObject
             arguments:(NSDictionary *)dict
{
     [(CALayer *)anObject addAnimation:self forKey:key];
}
```

Overriding an Implied Animation

You can provide a different implied animation for an action identifier by inserting an instance of `CAAnimation` into the `actions` dictionary, into an actions dictionary in the `style` dictionary, by implementing the delegate method `actionForLayer:forKey:`, or subclassing a layer class, overriding `defaultActionForKey:` and returning the appropriate action object.

The example in Listing 2 replaces the default implied animation for the `contents` property using delegation.

Listing 2 Implied animation for the contents property

```
- (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;
}
```

The example in Listing 3 (page 45) disables the default animation for the `sublayers` property using the `actions` dictionary pattern.

Listing 3 Implied animation for the sublayers property

```
// get a mutable version of the current actions dictionary
NSMutableDictionary *customActions=[NSMutableDictionary
dictionaryWithDictionary:[theLayer actions]];

// add the new action for sublayers
[customActions setObject:[NSNull null] forKey:@"sublayers"];

// set theLayer actions to the updated dictionary
theLayer.actions=customActions;
```

Temporarily Disabling Actions

You can temporarily disable actions when modifying layer properties by using transactions. See "Temporarily Disabling Layer Actions" (page 47) for more information.

Transactions

Every modification to a layer is part of a transaction. `CATransaction` is the Core Animation class responsible for batching multiple layer-tree modifications into atomic updates to the render tree.

This chapter describes the two types of transactions Core Animation supports: implicit transactions and explicit transactions.

Implicit transactions

Implicit transactions are created automatically when the layer tree is modified by a thread without an active transaction, and are committed automatically when the thread's run-loop next iterates.

The example in Listing 1 modifies a layer's `opacity`, `zPosition`, and `position` properties, relying on the implicit transaction to ensure that the resulting animations occur at the same time.

Listing 1 Animation using an implicit transaction

```
theLayer.opacity=0.0;
theLayer.zPosition=-200;
thelayer.position=CGPointMake(0.0,0.0);
```

Important: When modifying layer properties from threads that don't have a runloop, you must use explicit transactions.

Explicit Transactions

You create an explicit transaction by sending the `CATransaction` class a `begin` message before modifying the layer tree, and a `commit` message afterwards. Explicit transactions are particularly useful when setting the properties of many layers at the same time (for example, while laying out multiple layer), temporarily disabling layer actions, or temporarily changing the duration of resulting implied animations.

Temporarily Disabling Layer Actions

You can temporarily disable layer actions when changing layer property values by setting the value of the transaction's `kCATransactionDisableActions` to true. Any changes made during the scope of that transaction will not resulting in an animation occuring. Listing 2 shows an example that disables the fade animation that occurs when removing `aLayer` from a visible layer-tree.

Listing 2 Temporarily disabling a layer's actions

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

Overriding the Duration of Implied Animations

You can temporarily alter the duration of animations that run in response to changing layer property values by setting the value of the transaction's `kCATransactionAnimationDuration` key to a new duration. Any resulting animations in that transaction scope will use that duration rather than their own. Listing 3 shows an example that causes an animation to occur over 10 seconds rather than the duration specified by the `zPosition` and `opacity` animations..

Listing 3 Overriding the animation duration

```
[CATransaction begin];
[CATransaction setValue:[NSNumber numberWithFloat:10.0f]
                 forKey:kCATransactionAnimationDuration];
theLayer.zPosition=200.0;
theLayer.opacity=0.0;
[CATransaction commit];
```

Although the above example shows the duration bracketed by an explicit transaction `begin` and `commit`, you could omit those and use the implicit transaction instead.

Nesting Transactions

Explicit transactions can be nested, allowing you to disable actions for one part of an animation, or using different durations for the implicit animations of properties that are modified. Only when the outer-most transaction is committed will the animations occur.

Listing 4 shows an example of nesting two transactions. The outer transaction sets the implied animation duration to 2 seconds and sets the layer's `position` property. The inner transaction sets the implied animation duration to 5 seconds and changes the layer's `opacity` and `zPosition`.

Listing 4 Nesting explicit transactions

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

// change the animation duration to 2 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 5 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
```

Laying Out Core Animation Layers

NSView provides the classic "struts and springs" model of repositioning views relative to their superlayer when it resizes. While layers support this model, Core Animation on Mac OS X provides a more general layout manager mechanism that allows developers to write their own layout managers. A custom layout manager (which implements the CALayoutManager protocol) can be specified for a layer, which then assumes responsibility for providing layout of the layer's sublayers.

This chapter describes the constraints layout manager and how to configure a set of constraints.

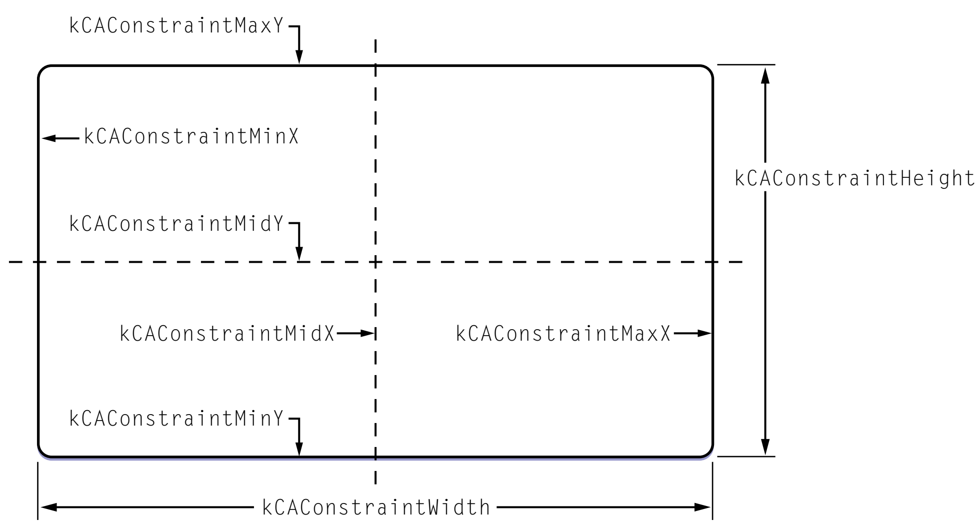
> **iPhone OS Note:** The CALayer class in iPhone OS only supports the "struts and springs" positioning model, it does not provide custom layout managers.

Constraints Layout Manager

Constraint-based layout allows you to specify the position and size of a layer using relationships between itself its sibling layers or its superlayer. The relationships are represented by instances of the CAConstraint class that are stored in an array in the sublayers' constraints property.

Figure 1 shows the layout attributes you can use when specifying relationships.

Figure 1 Constraint layout manager attributes



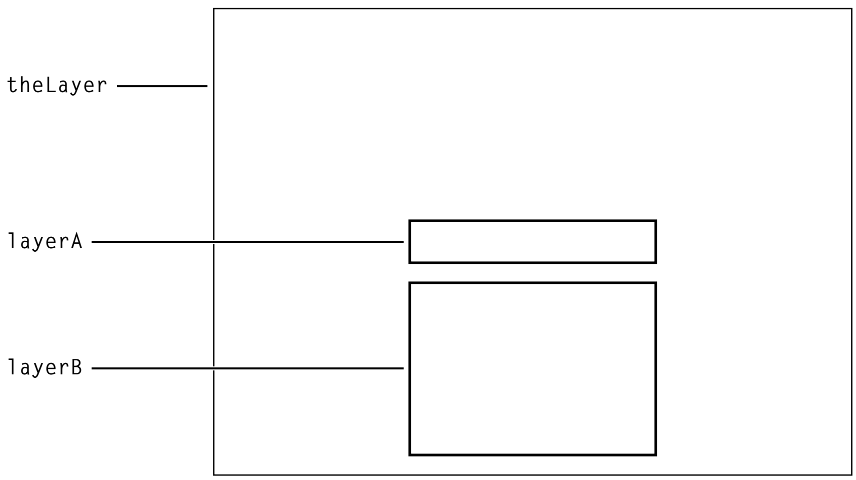
When using constraints layout you first create an instance of `CAConstraintsLayoutManager` and set it as the parent layer's layout manager. You then create constraints for the the sublayers by instantiating `CAConstraint` objects and adding them to the sublayer's constraints using `addConstraint:`. Each `CAConstraint` instance encapsulates one geometry relationship between two layers on the same axis.

Sibling layers are referenced by name, using the `name` property of a layer. The special name `superlayer` is used to refer to the layer's superlayer.

A maximum of two relationships must be specified per axis. If you specify constraints for the left and right edges of a layer, the width will vary. If you specify constraints for the left edge and the width, the right edge of the layer will move relative to the superlayer's frame. Often you'll specify only a single edge constraint, the layer's size in the same axis will be used as the second relationship.

The example code in Listing 1 creates a layer, and then adds sublayers that are positioned using constraints. Figure 2 shows the resulting layout.

Figure 2 Example constraints based layout



Listing 1 Configuring a layer's constraints

```
// create and set a constraint layout manager for theLayer
theLayer.layoutManager=[CAConstraintLayoutManager layoutManager];

CALayer *layerA = [CALayer layer];
layerA.name = @"layerA";

layerA.bounds = CGRectMake(0.0,0.0,100.0,25.0);
layerA.borderWidth = 2.0;

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

[layerA addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                                 relativeTo:@"superlayer"
                                                attribute:kCAConstraintMidX]];

[theLayer addSublayer:layerA];
```

```
CALayer *layerB = [CALayer layer];
layerB.name = @"layerB";
layerB.borderWidth = 2.0;

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintWidth
                                                 relativeTo:@"layerA"
                                                attribute:kCAConstraintWidth]];

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMidX
                                                 relativeTo:@"layerA"
                                                 attribute:kCAConstraintMidX]];


[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMaxY
                                                 relativeTo:@"layerA"
                                                  attribute:kCAConstraintMinY
                                                     offset:-10.0]];

[layerB addConstraint:[CAConstraint constraintWithAttribute:kCAConstraintMinY
                                                 relativeTo:@"superlayer"
                                                  attribute:kCAConstraintMinY
                                                     offset:+10.0]];

[theLayer addSublayer:layerB];
```

Here's what the code does:

1. Creates an instance of `CAConstraintsLayoutManager` and sets it as the `layoutManager` property of `theLayer`.

2. Creates an instance of `CALayer` (`layerA`) and sets the layer's `name` property to "layerA".

3. The bounds of `layerA` is set to a (0.0,0.0,100.0,25.0).

4. Creates a `CAConstraint` object, and adds it as a constraint of `layerA`.

   This constraint aligns the horizontal center of `layerA` with the horizontal center of the superlayer.

5. Creates a second `CAConstraint` object, and adds it as a constraint of `layerA`.

   This constraint aligns the vertical center of `layerA` with the vertical center of the superlayer.

6. Adds `layerA` as a sublayer of `theLayer`.

7. Creates an instance of `CALayer` (`layerB`) and sets the layer's `name` property to "layerB".

8. Creates a `CAConstraint` object, and adds it as a constraint of `layerA`.

   This constraint sets the width of `layerB` to the width of `layerA`.

9. Creates a second `CAConstraint` object, and adds it as a constraint of `layerB`.

   This constraint sets the horizontal center of `layerB` to be the same as the horizontal center of `layerA`.

10. Creates a third `CAConstraint` object, and adds it as a constraint of `layerB`.

    This constraint sets the top edge of `layerB` 10 points below the bottom edge of `layerA`.

11. Creates a fourth `CAConstraint` object, and adds it as a constraint of `layerB`.

    This constraint sets the bottom edge of `layerB` 10 points above the bottom edge of the superlayer.

> **Warning:** It is possible to create constraints that result in circular references to the same attributes. In cases where the layout is unable to be computed, the behavior is undefined.

Core Animation Extensions To Key-Value Coding

The `CAAnimation` and `CALayer` classes extend the `NSKeyValueCoding` protocol adding default values for keys, expanded wrapping conventions, and key path support for `CGPoint`, `CGRect`, `CGSize`, and `CATransform3D`.

Key-Value Coding Compliant Container Classes

Both `CALayer` and `CAAnimation` are key-value coding compliant container classes, allowing you to set values for arbitrary keys. That is, while the key "foo" is not a declared property of the `CALayer` class, however you can still set a value for the key "foo" as follows:

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

You retrieve the value for the key "foo" using the following code:

```
fooValue=[theLayer valueForKey:@"foo"];
```

Mac OS X Note: On Mac OS X, the `CALayer` and `CAAnimation` classes support the `NSCoding` protocol and will automatically archive any additional keys that you set for an instance of those classes.

Default Value Support

Core Animation adds a new convention to key value coding that allows a class to provide a default value that is used when a class has no value set for that key. Both `CALayer` or `CAAnimation` support this convention using the class method `defaultValueForKey:`.

To provide a default value for a key you create a subclass of the class and override `defaultValueForKey:`. The subclass implementation examines the key parameter and then returns the appropriate default value. Listing 1 shows an example implementation of `defaultValueForKey:` that provides a new default value for the layer property `masksToBounds`.

Listing 1 Example implementation of defaultValueForKey:

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

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

Wrapping Conventions

When using the key-value coding methods to access properties whose values are not objects the standard key-value coding wrapping conventions support, the following wrapping conventions are used:

C Type	Class
CGPoint	NSValue
CGSize	NSValue
CGRect	NSValue
CGAffineTransform	NSAffineTransform
CATransform3D	NSValue

Key Path Support for Structure Fields

`CAAnimation` provides support for accessing the fields of selected structures using key paths. This is useful for specifying these structure fields as the key paths for animations, as well as setting and getting values using `setValue:forKeyPath:` and `valueForKeyPath:`.

`CATransform3D` exposes the following fields:

Structure Field	Description
`rotation.x`	The rotation, in radians, in the x axis.
`rotation.y`	The rotation, in radians, in the y axis.
`rotation.z`	The rotation, in radians, in the z axis.
`rotation`	The rotation, in radians, in the z axis. This is identical to setting the `rotation.z` field.
`scale.x`	Scale factor for the x axis.
`scale.y`	Scale factor for the y axis.
`scale.z`	Scale factor for the z axis.
`scale`	Average of all three scale factors.
`translation.x`	Translate in the x axis.
`translation.y`	Translate in the y axis.
`translation.z`	Translate in the z axis.
`translation`	Translate in the x and y axis. Value is an NSSize or CGSize.

`CGPoint` exposes the following fields:

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

`CGSize` exposes the following fields:

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

`CGRect` exposes the following 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.

You can not specify a structure field key path using Objective-C 2.0 properties. This will not work:

```
    myLayer.transform.rotation.x=0;
```

Instead you must use `setValue:forKeyPath:` or `valueForKeyPath:` as shown below:

```
    [myLayer setValue:[NSNumber numberWithInt:0]
forKeyPath:@"transform.rotation.x"];
```

Layer Style Properties

Regardless of the type of media a layer displays, a layer's style properties are applied by the render-tree as it composites layers.

This chapter describes the layer style properties and provides examples of their effect on an example layer.

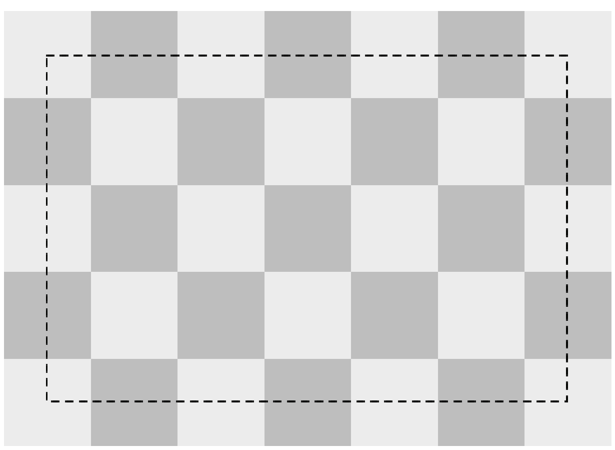
Note: The layer style properties available on Mac OS X and iPhone OS differ and are noted below.

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 (available only on Mac OS X) and a transform that is applied to the layer and its sublayers.

Figure 1 shows the geometry of the example layer.

Figure 1 Layer geometry



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

- `frame`
- `bounds`
- `position`
- `anchorPoint`
- `cornerRadius`

- `transform`
- `zPosition`

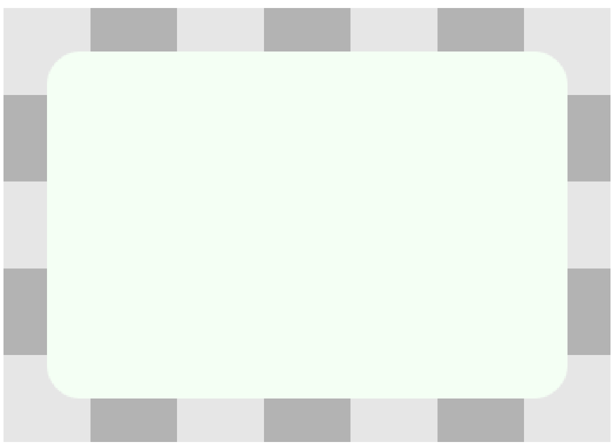
iPhone OS Note: iPhone OS does not support the `cornerRadius` property. To simulate the visual effect of a corner radius you can draw the content using the appropriate clipping regions. You can override the hit testing behavior of a layer and exclude touches as appropriate to emulate a geometry with a corner radius, although this is rarely necessary in a touch-based user interface.

Background Properties

Next, the layer renders its background. You can define a color for the background as well as a Core Image filter.

Figure 2 illustrates the sample layer with its `backgroundColor` set.

Figure 2 Layer with background color



The background filter is applied to the content behind the layer. For example, you may wish to apply a blur filter as a background filter to make the layer content stand out better.

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

- `backgroundColor`
- `backgroundFilters`

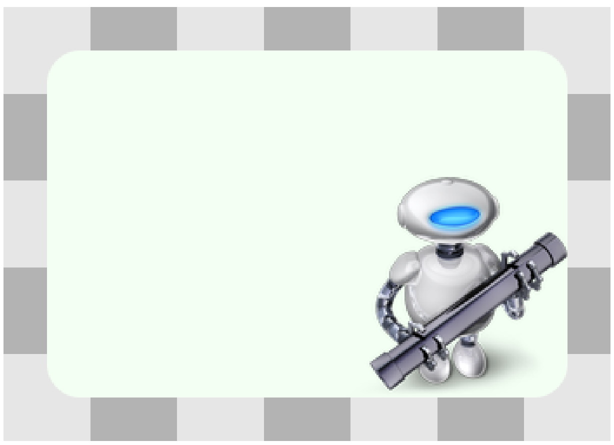
> **iPhone OS Note:** While the `CALayer` class in iPhone OS exposes the `backgroundFilters` property, Core Image is not available. The filters available for this property are currently undefined.

Layer Content

Next, if set, the content of the layer is rendered. The layer content can be created using the Quartz graphics environment, OpenGL, QuickTime, or Quartz Composer.

Figure 4 shows the example layer with its content composited.

Figure 3 Layer displaying a content image



By default, the content of a layer is not clipped to its bounds and corner radius. The `masksToBounds` property can be set to `true` to clip the layer content to those values.

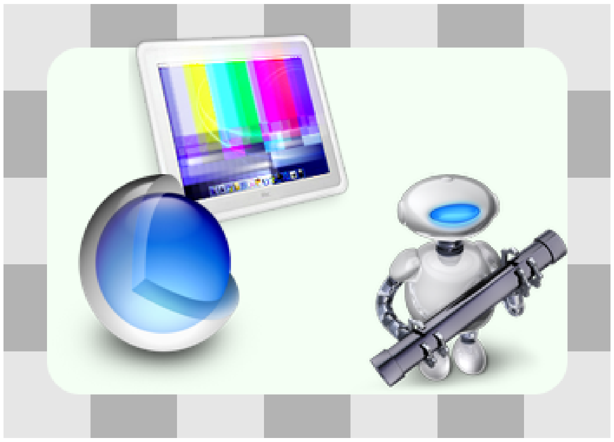
The following `CALayer` properties affect the display of a layer's content:

- `contents`
- `contentsGravity`

Sublayers Content

It is typical that a layer will have a hierarchy of child layers, its sublayers. These sublayers are rendered recursively, relative to the parent layer's geometry. The parent layer's `sublayerTransform` is applied to each sublayer, relative to the parent layer's anchor point.

Figure 4 Layer displaying the sublayers content



By default, a layer's sublayers are not clipped to the layer's bounds and corner radius. The `masksToBounds` property can be set to `true` to clip the layer content to those values. The example layer's `maskToBounds` property is `false`; notice that the sublayer displaying the monitor and test pattern is partially outside of its parent layer's bounds.

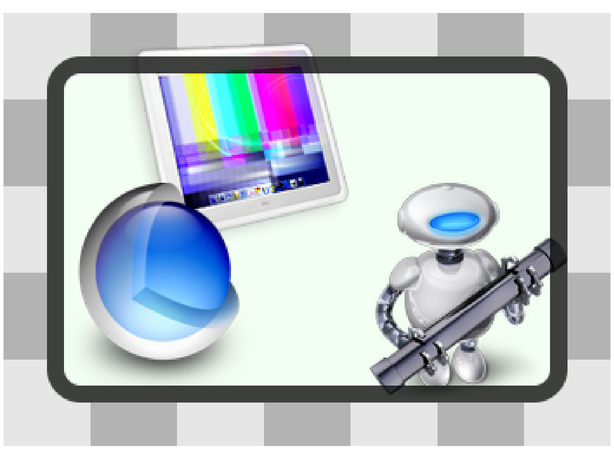
The following `CALayer` properties affect the display of a layer's sublayers:

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

Border Attributes

A layer can display an optional border using a specified color and width. Figure 5 shows the example layer after applying a border.

Figure 5 Layer displaying the border attributes content



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

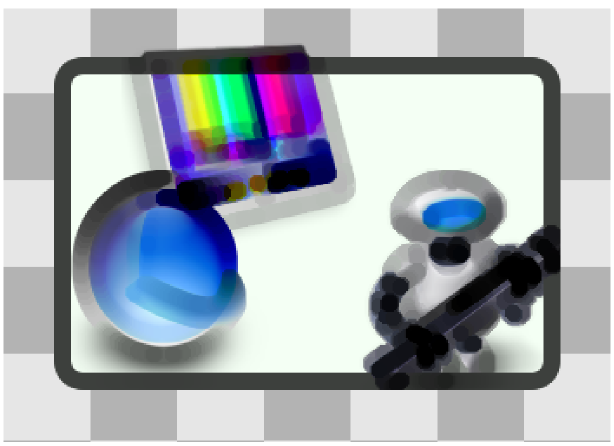
- `borderColor`
- `borderWidth`

iPhone OS Note: As a performance consideration, iPhone OS does not support the `borderColor` and `borderWidth` properties. Drawing a border for layer content is the responsibility of the developer.

Filters Property

An array of Core Image filters can be applied to the layer. These filters affect the layer's border, content, and background. Figure 6 shows the example layer with the Core Image posterize filter applied.

Figure 6 Layer displaying the filters properties



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

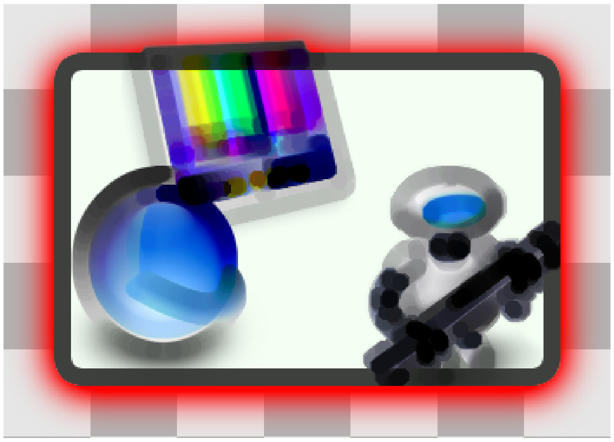
- `filters`

iPhone OS Note: While the `CALayer` class in iPhone OS exposes the `filters` property, Core Image is not available. Currently the filters available for this property are undefined.

Shadow Properties

Optionally, a layer can display a shadow, specifying its opacity, color, offset, and blur radius. Figure 7 shows the example layer with a red shadow applied.

Figure 7 Layer displaying the shadow properties



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

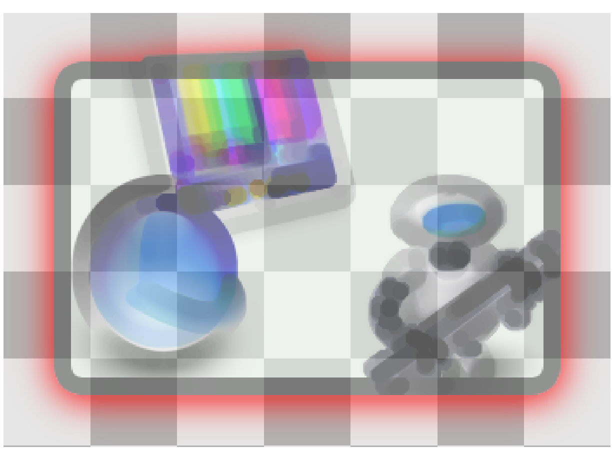
- shadowColor
- shadowOffset
- shadowOpacity
- shadowRadius

> **iPhone OS Note:** As a performance consideration, iPhone OS does not support the shadowColor, shadowOffset, shadowOpacity, and shadowRadius properties.

Opacity Property

By setting the opacity of a layer, you can control the layer's transparency. Figure 8 shows the example layer with an opacity of 0.5.

Figure 8 Layer including the opacity property



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

- `opacity`

Composite Property

A layer's compositing filter is used to combine the layer content with the layers behind it. By default, a layer is composited using source-over. Figure 9 shows the example layer with a compositing filter applied.

Figure 9 Layer composited using the compositingFilter property

The following `CALayer` property specifies the composting filter for a layer:

- `compositingFilter`

> **iPhone OS Note:** While the `CALayer` class in iPhone OS exposes the `compositingFilter` property, Core Image is not available. Currently the filters available for this property are undefined.

Mask Properties

Finally, you can specify a layer that will serve as a mask, further modifying how the rendered layer appears. The opacity of the mask layer determines masking when the layer is composited. Figure 10 shows the example layer composited with a mask layer.

Figure 10 Layer composited with the mask property

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

- `mask`

iPhone OS Note: As a performance consideration, iPhone OS does not support the `mask` property.

Example: Core Animation Menu Application

The Core Animation Menu example displays a simple selection example using Core Animation layers to generate and animate the user interface. In less than 100 lines of code, it demonstrates the following capabilities and design patterns:

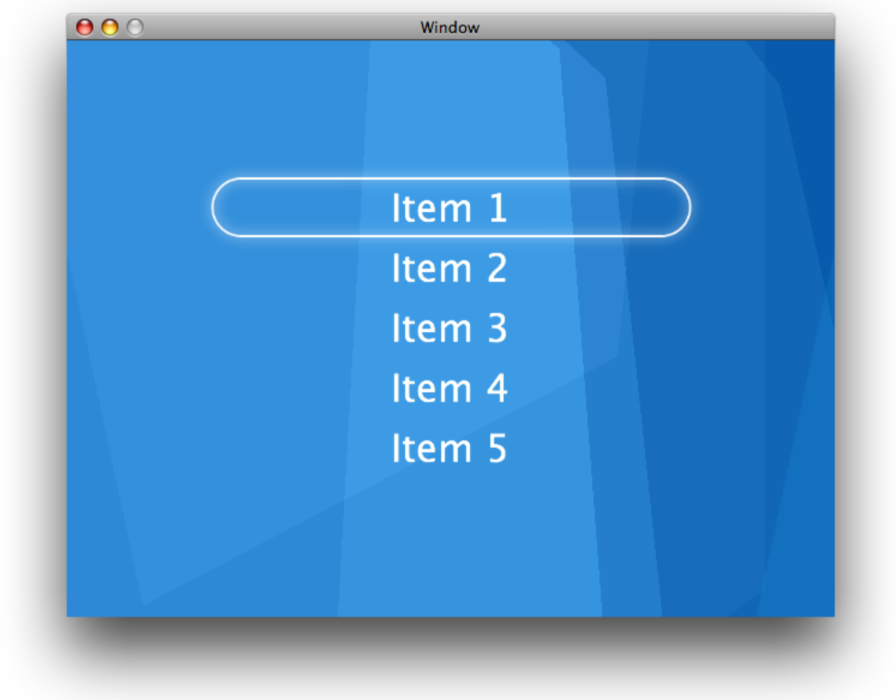
- Hosting the root-layer of a layer hierarchy in a view.
- Creating and inserting layers into a layer hierarchy.
- Using a `QCCompositionLayer` to display Quartz Composer compositions as layer content.
- Using an explicit animation that runs continuously.
- Animating Core Image Filter inputs.
- Implicitly animating the position of the selection item.
- Handling key events through the MenuView instance that hosts the view.

This application makes heavy use of Core Image filters and Quartz Composer compositions and, as a result, runs only on Mac OS X. The techniques illustrated for managing the layer hierarchy, implicit and explicit animation, and event handling are common to both platforms.

The User Interface

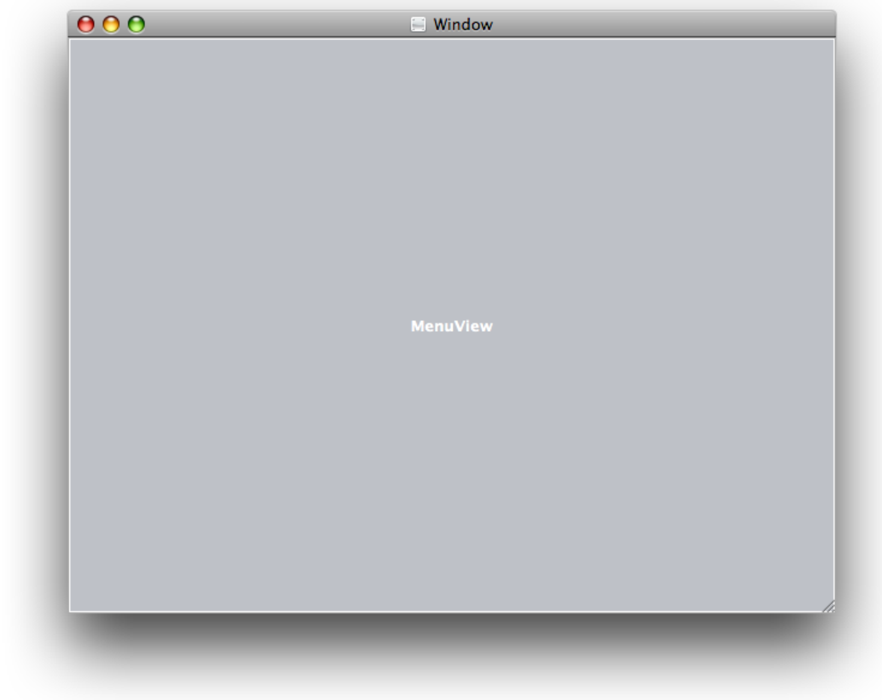
The Core Animation Menu application provides a very basic user interface; the user can select a single item in a menu. The user navigates the menu using the up and down arrows on the keyboard. As the selection changes the selection indicator (the rounded white rectangle) animates smoothly to its new location. A continuously animating bloom filter is set for the selection indicator causing it to subtly catch your attention. The background is a Quartz Composer animation that runs continuously. Figure 1 shows the application's interface.

Figure 1 Core Animation Menu Interface



Examining the Nib File

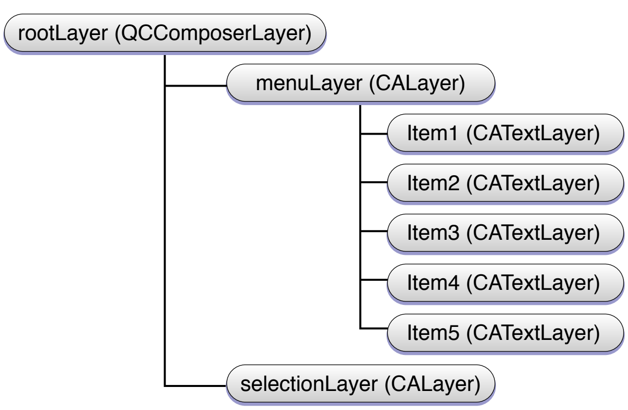
`Menu.nib` is very straightforward. An instance of `CustomView` is dragged from the Interface Builder palette and positioned in the window. It is resized such that it fills the entire window. The `MenuView.h` file is imported into Interface Builder by dragging it to the Menu.nib window. The `CustomView` is then selected, and the object type is changed to `MenuView`.



No other connections need to be made. When the nib file is loaded the window is unarchived and the `MenuView` is as well. The `MenuView` class gets an `awakeFromNib` message and the layers are configured there.

The Layer Hierarchy

The layer hierarchy, also referred to as the layer tree, of the Menu application is shown below.



The `rootLayer` is an instance of `QCComposerLayer`. As the root-layer this layer is the same size as the `MenuView` instance, and remains that way as the window is resized.

The `menusLayer` is a sublayer of the `rootLayer`. It is an empty layer; it does not have anything set as its `contents` property and none of its style properties are set. The `menusLayer` is simply used as a container for the menu item layers. This approach allows the application to easily access a menu item sublayer by its position in the `menusLayers.sublayers` array. The `menusLayer` is the same size as, and overlaps, the `rootLayer`. This was done intentionally so that there was no need to convert between coordinate systems when positioning the `selectionLayer` relative to the current menu item.

The Code

Having looked at the application's nib file and the overall design, you can now begin examining the implementation of the `MenuView` class..

Examining MenuView.h

The `MenuView` class is a subclass of `NSView` and it declares four instance variables:

- `NSIndex selectedIndex` — tracks the index that is currently selected.
- `CALayer *menusLayer` —the Core Animation layer that contains the menus items as its sublayers.
- `CALayer *selectionLayer` — the Core Animation layer that displays the selection indicator
- `NSArray *name` — an array of names displayed as menu items

Note: Notice that `Quartz/CoreAnimation.h` is imported. The `QuartzCore.framework` must be added to any project that uses Core Animation. Because this example uses Quartz Composer the `MenuView` implementation also imports `Quartz/Quartz.h`, and the `Quartz.framework` is added to the project.

Listing 1 MenuView.h listing

```
#import <Cocoa/Cocoa.h>
#import <QuartzCore/CoreAnimation.h>


// the MenuView class is the view subclass that is inserted into
// the window.  It hosts the rootLayer, and responds to events
@interface MenuView : NSView {

    // contains the selected menu item index
     NSInteger selectedIndex;

    // the layer that contains the menu item layers
    CALayer *menusLayer;

    // the layer that is used for the selection display
    CALayer *selectionLayer;

    // the array of menu item names
    NSArray *names;

}
```

```
-(void)awakeFromNib;
-(void)setupLayers;
-(void)changeSelectedIndex:(NSInteger)theSelectedIndex;
-(void)moveUp:(id)sender;
-(void)moveDown:(id)sender;
-(void)dealloc;
```

Examining MenuView.m

The `MenuView` class is the workhorse of this application. It responds when the view is loaded by the nib, sets up the layers to be displayed, creates the animations, and handles the keys that move the selection.

The examination of the `MenuView.m` is split as follows:

- Setting Up the `MenuView`
- Setting Up the Layers
- Animating the Selection Layer Movement
- Responding to Key Events
- Cleaning Up

Setting Up the MenuView

The `awakeFromNib` method is called when `Menu.nib` is loaded and unarchived. The view is expected to complete its setup in `awakeFromNib`.

The `MenuView` implementation of `awakeFromNib` creates an array of strings, names, that are used to display the menu items. It then calls the `setupLayers` method to setup the layers for the view.

```
- (void)awakeFromNib
{
    names=[[NSArray arrayWithObjects:@"Item 1",@"Item 2",
                                     @"Item 3",@"Item 4",@"Item 5",
                                     nil] retain];

    [self setupLayers];
}
```

Setting Up the Layers

The majority of the code in the Menu example resides in the `setupLayers` method. This method is responsible for the following:

- Creating and initializing `rootLayer`
- Setting `rootLayer` as the hosted layer of the view
- Creating and initializing the `menusLayer`
- Creating and initializing the menu item layers
- Adding the menu item positioning constraints

- Layout the menusLayer
- Creating the selectionLayer
- Configuring the continuous animation of selectionLayer
- Adding it to the layer tree of rootLayer
- Setting the initial value of selectedIndex

First, the constants used to position and space the layers are defined.

```
-(void)setupLayers;
{
    CGFloat width=400.0;
    CGFloat height=50.0;
    CGFloat spacing=20.0;
    CGFloat fontSize=32.0;
    CGFloat initialOffset=100.0;
```

The view must be set as the first responder to allow it to initially handle the up and down arrow events.

```
[[self window] makeFirstResponder:self];
```

Create the rootLayer, The rootlayer is an instance of QCCompositionLayer that displays the Background.qtz file which is located within the application bundle.

```
QCCompositionLayer* rootLayer;
rootLayer=[QCCompositionLayer compositionLayerWithFile:
           [[NSBundle mainBundle] pathForResource:@"Background"
                                           ofType:@"qtz"]];
```

The instance of MenuView is set as the layer-hosting view of rootLayer. The order of these two calls is important. By first setting the layer to rootLayer and then setting setWantsLayer: to YES our layer is used rather than the one that the view would create. This is the key difference between layer-hosting views and layer-backed views.

```
[self setLayer:rootLayer];
[self setWantsLayer:YES];
```

Create the menusLayer, and set its bounds to those of rootLayer. Again, this is done to allow us to use the same coordinate system for both the menusLayer sublayers and the selectedLayer. The menusLayer is also retained, MenuView requires it when positioning the selectedLayer.

```
menusLayer=[[CALayer layer] retain];
menusLayer.frame=rootLayer.frame;
```

Specify that the sublayers of menusLayer will be laid out using the CAConstraintLayoutManager. Constraints layout allows you to specify the location and size of layers relative to their sibling layers and superlayer. The superlayer is configured to use the constraints manager, and individual CAContraint instances are created and attached to each of the sublayers.

```
menusLayer.layoutManager=[CAConstraintLayoutManager layoutManager];
```

Add the menusLayer as a sublayer of the rootLayer.

```
[rootLayer addSublayer:menusLayer];
```

The next code fragment iterates over the items in the names array, creating a new `CATextLayer` for each name and defines its position using constraints.

```
NSInteger i;
for (i=0;i<[names count];i++) {
```

Get the name at the index of the current iteration.

```
NSString *name=[names objectAtIndex:i];
```

Create a new `CATextLayer` instance called `menuItemLayer`. Set its string to the name of the menu item, and specify that it should be displayed in white 32 point Lucida-Grande.

```
CATextLayer *menuItemLayer=[CATextLayer layer];
menuItemLayer.string=name;
menuItemLayer.font=@"Lucida-Grande";
menuItemLayer.fontSize=fontSize;
menuItemLayer.foregroundColor=CGColorCreateGenericRGB(1.0,1.0,1.0,1.0);
```

Note that the bounds of the `menuItemLayer` is never specified. When using `CATextLayer` instances the constraints manager takes responsibility for setting the bounds and height of the layer.

The next step is to specify the constraints for the layout. First the vertical constraint is set relative to the top edge of the superlayer. The top edge of `menuItemLayer` is offset by the `initialOffset` (defined earlier) and by the spacing between items (also specified earlier) and the height (again specified earlier) is multiplied by the index of the name. The final value is inverted because the layer coordinate system uses the bottom left as its origin.

```
[menuItemLayer addConstraint:[CAConstraint
              constraintWithAttribute:kCAConstraintMaxY
                           relativeTo:@"superlayer"
                            attribute:kCAConstraintMaxY
                               offset:-(i*height+spacing+initialOffset)]];
```

The second constraint simply causes the `menuItemLayer` object to be centered horizontally, relative to the center of its superlayer.

```
[menuItemLayer addConstraint:[CAConstraint
                              constraintWithAttribute:kCAConstraintMidX
                                           relativeTo:@"superlayer"
                                            attribute:kCAConstraintMidX]];
```

Each `menuItemLayer` is added to the menusLayer layer as a sublayer.

```
[menusLayer addSublayer:menuItemLayer];
} // end of for loop
```

Having configured all the menu item layers you must now force them to be laid out immediately. This is necessary to ensure that the first placement of the `selectionLayer` is correct.

```
[menusLayer layoutIfNeeded];
```

Now the `CALayer` that is used as the `selectionlayer` is created and configured. The `bounds` is set to be the width and height defined earlier. The layer is retained because we rely on it being available to `MenuView` after the layer is added to the layer tree.

```
selectionLayer=[[CALayer layer] retain];
selectionLayer.bounds=CGRectMake(0.0,0.0,width,height);
```

The `selectionLayer` depends on the `borderWidth`, `borderColor`, and `cornerRadius` style properties to provide its visual components. They are set to 2 points wide, a color of white, and a corner radius that ensures that the ends of the selectionLayer are rounded completely.

```
selectionLayer.borderWidth=2.0;
selectionLayer.borderColor=CGColorCreateGenericRGB(1.0f,1.0f,1.0f,1.0f);
selectionLayer.cornerRadius=height/2;
```

As the `selectionLayer` is displayed it softly pulses every second. This is done using a CIBloom filter and animating its inputIntensity between 0 (no intensity) and 1.5 (somewhat intense).

Create the filter, set its default values, and then specify the inputRadius is 5.0.

```
CIFilter *filter = [CIFilter filterWithName:@"CIBloom"];
[filter setDefaults];
[filter setValue:[NSNumber numberWithFloat:5.0] forKey:@"inputRadius"];
```

Core Animation extends the `CIFilter` class by adding the `name` property. The `name` property allows the inputs of filters in the layer's filters array to be animated using a key path.

```
[filter setName:@"pulseFilter"];
```

Set the `selectionLayer` filters array so that it contains filter.

```
[selectionLayer setFilters:[NSArray arrayWithObject:filter]];
```

The pulse animation is an explicit animation that runs continuously. It is a subclass of `CABasicAnimation` and, as such, must specify values for a `keyPath`, `toValue`, and `fromValue`.

```
CABasicAnimation* pulseAnimation = [CABasicAnimation animation];
```

Set the key path to be animated to `filters.pulseFilter.inputIntensity`. This is where the filter's name property is used.

```
pulseAnimation.keyPath = @"filters.pulseFilter.inputIntensity";
```

Set the `fromValue` and `toValue` to 0 and 1.0 respectively. This gives a nice pulse effect.

```
pulseAnimation.fromValue = [NSNumber numberWithFloat: 0.0];
pulseAnimation.toValue = [NSNumber numberWithFloat: 1.0];
```

The animation is 1 second long, and it repeats indefinitely. When the animation reaches 1.5, it cycles back to 0, and so on. The following code sets that up.

```
pulseAnimation.duration = 1.0;
pulseAnimation.repeatCount = 1e100f;
pulseAnimation.autoreverses = YES;
```

The `timingFunction` of an animation controls how the animation values are distributed over the course of the animation duration. In this case we'll use an easeIn-easeOut animation. This causes the animation to begin slowly, ramp up to speed, and then slow again before completing.

```
pulseAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
                                    kCAMediaTimingFunctionEaseInEaseOut];
```

For an explicit animation to begin you must add it to the layer's animation collection. This is done using `addAnimation:forKey:`. The key itself is used as an identifier for removing the animation later, if necessary.

```
[selectionLayer addAnimation:pulseAnimation forKey:@"pulseAnimation"];
```

Finally, now that setup is complete add the `selectionLayer` to the `rootLayer`.

```
[rootLayer addSublayer:selectionLayer];
```

Set the initial position of the `selectionLayer` and the initial `selectedIndex` to 0.

```
[self changeSelectedIndex:0];
// end of setupLayers
```

The `setupLayers` method is by far the longest and most complex in this application. However, by breaking it down into the setup for each layer, it becomes much easier to understand.

Animating the Selection Layer Movement

The method `changeSelectedIndex:` is responsible for: setting selectedIndex to the new value, ensuring that the new value of selectedIndex is within the range of the number of items in the menu items, and positioning the selection layer relative to the `menusLayer` sublayer at the `selectedIndex`. This causes the selection layer to animate to show that the new item is selected.

```
-(void)changeSelectedIndex:(NSInteger)theSelectedIndex
{
    selectedIndex=theSelectedIndex;

    if (selectedIndex == [names count]) selectedIndex=[names count]-1;
    if (selectedIndex < 0) selectedIndex=0;

    CALayer *theSelectedLayer=[[menusLayer sublayers]
objectAtIndex:selectedIndex];
    selectionLayer.position=theSelectedLayer.position;
};
```

Notice that all that is required to animate the `selectionLayer` is to simply assign a new value to its `position` property. This is an example of implicit animation

Responding to Key Events

Because layers do not take part in the responder chain, or accept events, the MenuView that acts as the layer-host for the layer tree must assume that role. The `moveUp:` and `moveDown:` messages are provided by `NSResponder`, of which `MenuView` is a descendent. The `moveUp:` and `moveDown:` messages are invoked when the up arrow and down arrows are pressed respectively. Using these methods allows the application to respect any remapped arrow key functionally specified by the user. (And it's easier than implementing `keyDown:`).

When the up arrow is pressed the `selectedIndex` value is de-incremented and updated by calling `changeSelectedIndex:`.

```
-(void)moveUp:(id)sender
{
    [self changeSelectedIndex:selectedIndex-1];
}
```

When the down arrow is pressed the `selectedIndex` value is incremented and updated by calling `changeSelectedIndex:`.

```
-(void)moveDown:(id)sender
{
    [self changeSelectedIndex:selectedIndex+1];

}
```

Cleaning Up

When the `MenuView` is released, we are responsible for cleaning up our instance variables. The `menusLayer`, `selectionLayer`, and `names` are autoreleased in the `dealloc` implementation.

```
-(void)dealloc
{
    [menusLayer autorelease];
    [selectionLayer autorelease];
    [names autorelease];
    [super dealloc];
}
```

Animatable Properties

CALayer Animatable Properties

The following CALayer class properties can be animated by Core Animation. See CALayer for more information.

- anchorPoint

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- backgroundColor

  Uses the default implied CABasicAnimation described in Table 10 (page 79). (subproperties are animated using a basic animation)

- backgroundFilters

  Uses the default implied CATransitionAnimation described in Table 11 (page 79). Sub-properties of the filters are animated using the default implied CABasicAnimation described in Table 10 (page 79).

- borderColor

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- borderWidth

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- bounds

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- compositingFilter

  Uses the default implied CATransitionAnimation described in Table 11 (page 79). Sub-properties of the filters are animated using the default implied CABasicAnimation described in Table 10 (page 79).

- contents
- contentsRect

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- cornerRadius

  Uses the default implied CABasicAnimation described in Table 10 (page 79).

- doubleSided

  No default implied animation is set.

- filters

  Uses the default implied CABasicAnimation described in Table 10 (page 79). Sub-properties of the filters are animated using the default implied CABasicAnimation described in Table 10 (page 79).

- `frame`

  The frame property itself is not animatable. You can achieve the same results by modifying the `bounds` and `position` properties instead.

- `hidden`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `mask`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79). This property is available only on Mac OS X.

- `masksToBounds`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `opacity`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `position`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `shadowColor`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79). This property is available only on Mac OS X.

- `shadowOffset`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79). This property is available only on Mac OS X.

- `shadowOpacity`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79). This property is available only on Mac OS X.

- `shadowRadius`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79). This property is available only on Mac OS X.

- `sublayers`

  Uses the default implied `CATransitionAnimation` described in Table 11 (page 79).

- `sublayerTransform`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `transform`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

- `zPosition`

  Uses the default implied `CABasicAnimation` described in Table 10 (page 79).

Table 10 Default Implied Basic Animation

Description	Value
Class	CABasicAnimation
`duration`	.25 seconds, or the duration of the current transaction
keyPath	Dependent on layer property type

Table 11 Default Implied Transition

Description	Value
Class	CATransition
duration	.25 seconds, or the duration of the current transaction
type	Fade (`kCATransitionFade`)
startProgress	0.0
endProgress	1.0

CIFilter Animatable Properties

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

- `name`
- `enabled`

Document Revision History

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

Date	Notes
2008-11-13	Introduces iPhone SDK content to Mac OS X content. Corrects frame animation capabilities.
2008-09-09	Corrected typos.
2008-06-18	Updated for iPhone OS.
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.
	New document that introduces the main components and services of Core Animation.
	Added “Key-Value Coding Additions” chapter.
	Updated class names to reflect new Core Animation API prefix.