Animation Types and Timing Programming Guide

Developer

Contents

Figures, Tables, and Listings

Introduction to Animation Types and Timing Programming Guide

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

This document describes the fundamental concepts involving the timing and animation classes used with Core Animation. Core Animation is an Objective-C framework that combines a high-performance compositing engine with a simple to use animation programming interface.

Note: Animation is an inherently visual medium. The HTML version of *Animation Types and Timing Programming Guide* contains QuickTime movies (along with static images) that show example animations to help illustrate concepts. The PDF version contains only the static images.

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.

Organization of This Document

Animation Types and Timing consists of the following articles:

- Animation Class Roadmap (page 6) provides an overview of the animation classes and timing protocol.
- Timing, Timespaces, and CAAnimation (page 8) describes in detail the timing model for Core Animation and the abstract `CAAnimation` class.
- Property-Based Animations (page 12) describes the property-based animations: `CABasicAnimation` and `CAKeyframeAnimation`.
- Transition Animation (page 18) describes the transition animation class, `CATransition`.

See Also

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

- *Animation Overview* describes the animation technologies available on OS X.
- *Core Animation Programming Guide* contains code fragments that demonstrate common Core Animation tasks.
- *Animation Programming Guide for Cocoa* describes the animation capabilities available to Cocoa Applications.

Animation Class Roadmap

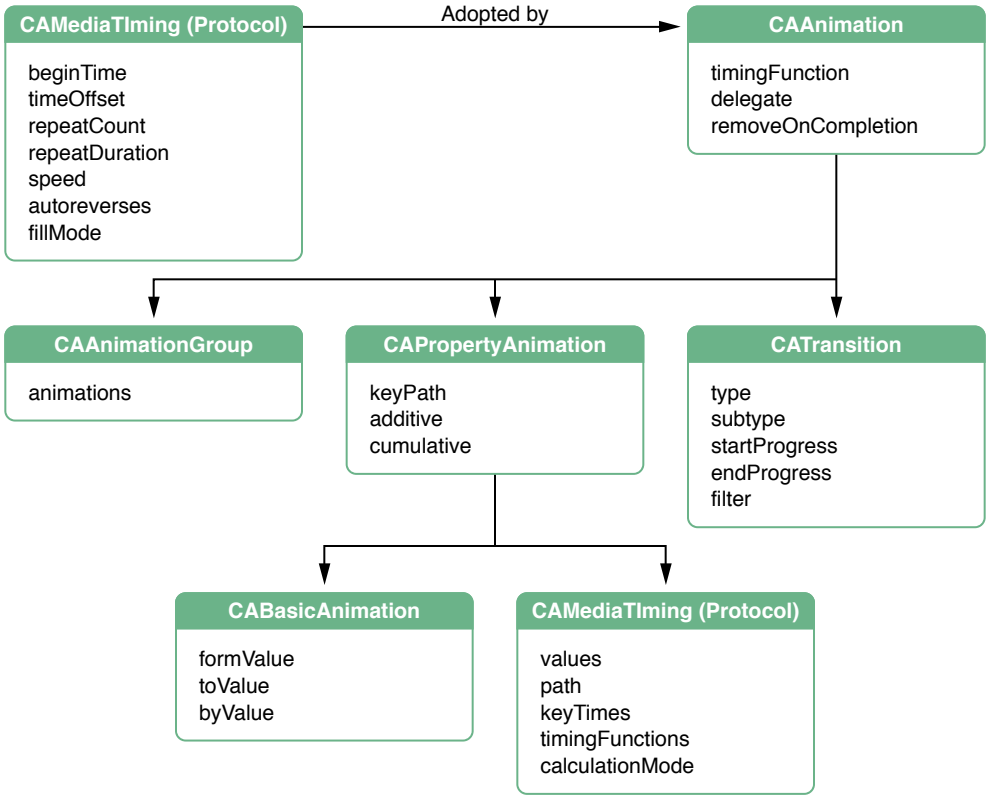
Core Animation provides an expressive set of animation classes you can use in your application:

- `CAAnimation` is the abstract class that all animations subclass. `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 `CAAnimation` class also defines an animation's timing as an instance of `CAMediaTimingFunction`. The timing function describes the pacing of the animation as a simple Bezier curve. 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 speed up as it nears its duration.
- `CAPropertyAnimation` is an abstract subclass of `CAAnimation` that provides support for animating a layer property specified by a key path.
- `CABasicAnimation` is a subclass of `CAPropertyAnimation` that provides simple interpolation for a layer property.
- `CAKeyframeAnimation` (a subclass of `CAPropertyAnimation`) 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. On OS X, 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.

Figure 1 shows the animation class hierarchy, and also summarizes the properties available through inheritance.

Figure 1 Core Animation classes and protocol



Timing, Timespaces, and CAAnimation

When broken down to its simplest definition an animation is simply the varying of a value over a time. Core Animation provides base timing functionality for both animations and layers, providing a powerful timeline capability.

This chapter discusses the timing protocol and the basic animation support common to all animation subclasses.

Media Timing Protocol

The Core Animation timing model is described by the `CAMediaTiming` protocol and adopted by the `CAAnimation` class and its subclasses. The timing model specifies the time offset, duration, speed, and repeating behavior of an animation.

The `CAMediaTiming` protocol is also adopted by the `CALayer` class, allowing a layer to define a timespace relative to its superlayer; similar manner to describing a relative coordinate space. This concept of a layer-tree timespace provides a scalable timeline that starts at the root layer, through its descendants. Since an animation must be associated with a layer to be displayed, the animation's timing is scaled to the timespace defined by the layer.

The `speed` property of an animation or layer specifies this scaling factor. For example, a 10 second animation that is attached to a layer with a timespace that has a speed value of 2 will take 5 seconds to display (twice the speed). If a sublayer of that layer also defines a speed factor of 2, then its animations will display in 1/4 the time (the speed of the superlayer * the speed of the sublayer).

Similarly, a layer's timespace can also affect the playback of dynamic layer media such as Quartz Composer compositions. Doubling the speed of a `QCCompositionLayer` causes the composition to play twice as fast, as well as doubling the speed of any animations attached to that layer. Again, this effect is hierarchical, so any sublayers of the `QCCompositionLayers` will also display their content using the increased speed.

The `duration` property of the `CAMediaTiming` protocol is used by animations to define how long, in seconds, a single iteration of an animation will take to display. The default duration for subclasses of `CAAnimation` is 0 seconds, which indicates that the animation should use the duration specified by the transaction in which it is run, or .25 seconds if no transaction duration is specified.

The timing protocol provides the means of starting an animation a certain number of seconds into its duration using two properties: `beginTime` and `timeOffset`. The `beginTime` specifies the number of seconds into the duration the animation should start and is scaled to the timespace of the animation's layer. The `timeOffset` specifies an additional offset, but is stated in the local active time. Both values are combined to determine the final starting offset.

Repeating Animations

The repetition behavior of an animation is also determined by the `CAMediaTiming` protocol by the `repeatCount` and `repeatDuration` properties. The `repeatCount` specifies the number of times the animation should repeat and can be a fractional number. Setting the `repeatCount` to a value of 2.5 for a 10 second animation would cause the animation to run for a total of 25 seconds, ending half way through its third iteration. Setting the repeatCount to `1e100f` will cause the animation to repeat until it is removed from the layer.

The `repeatDuration` is similar to the `repeatCount`, although it is specified in seconds rather than iterations. The `repeatDuration` can also be a fractional value.

The `autoreverses` property of an animation determines whether the animation plays backwards after it finishes playing forwards; assuming that multiple repetitions are specified.

Fill Mode

The `fillMode` property of the timing protocol defines how an animation will be displayed outside of its active duration. The animation can be frozen at its starting position, at its ending position, both, or removed entirely from display. The default behavior is to remove the animation from display when it has completed.

Animation Pacing

The pacing of an animation determines how the interpolated values are distributed over the duration of the animation. Using the appropriate pacing for a particular visual effect can greatly enhance its affect on the user.

The pacing of an animation is represented by a timing function that is expressed as a cubic Bezier curve. This function maps the duration of a single cycle of the animation (normalized to the range [0.0,1.0]) to the output time (also normalized to that range).

The timingFunction property of the `CAAnimation` class specifies an instance of the `CAMediaTimingFunction`, the class responsible for encapsulating the timing functionality.

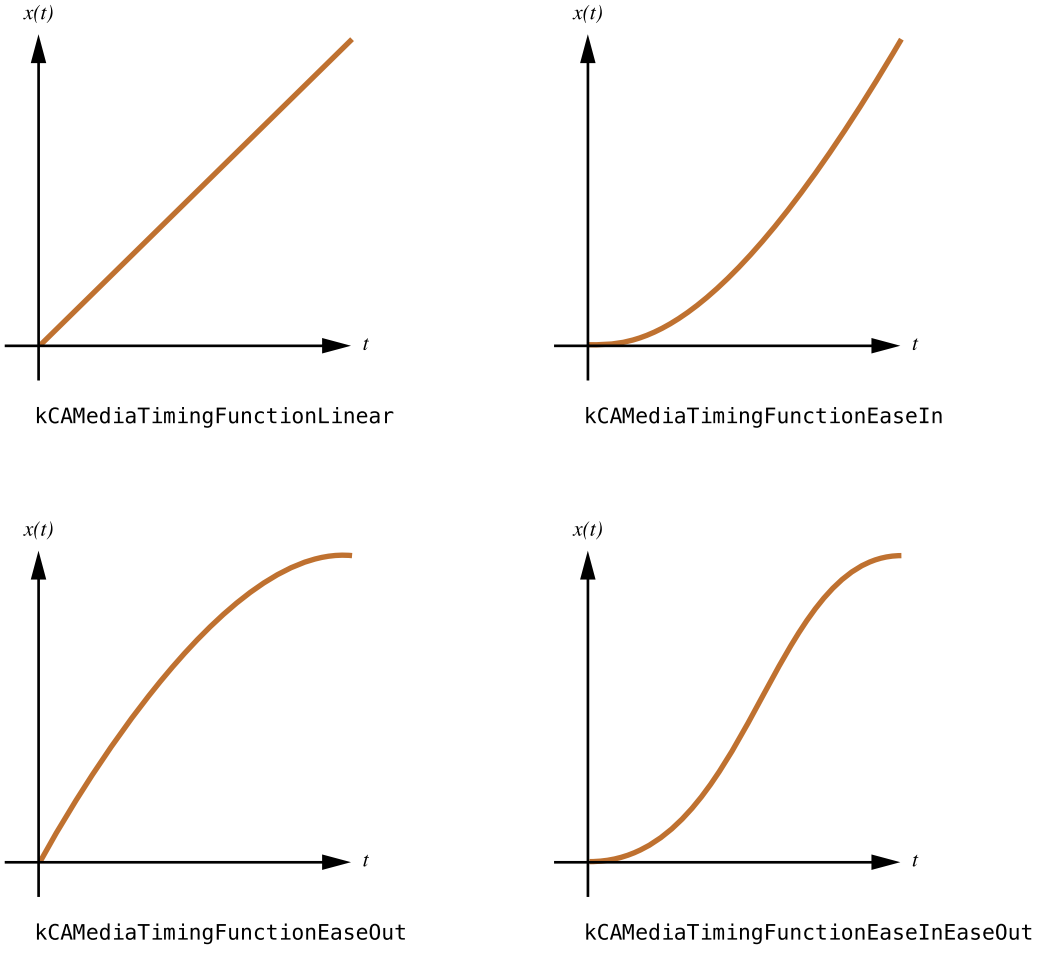
`CAMediaTimingFunction` provides two options for specifying the mapping function: constants for the common pacing curves, and methods for creating custom functions by specifying two control points.

The predefined timing functions are returned by specifying one of the following constants to the `CAMediaTimingFunction` class method `functionWithName:`:

- `kCAMediaTimingFunctionLinear` specifies linear pacing. A linear pacing causes an animation to occur evenly over its duration.
- `kCAMediaTimingFunctionEaseIn` specifies ease-in pacing. Ease-in pacing causes the animation to begin slowly, and then speed up as it progresses.
- `kCAMediaTimingFunctionEaseOut` specifies ease-out pacing. An ease-out pacing causes the animation to begin quickly, and then slow as it completes.
- `kCAMediaTimingFunctionEaseInEaseOut` specifies ease-in ease-out pacing. An ease-in ease-out animation begins slowly, accelerates through the middle of its duration, and then slows again before completing.

Figure 1 shows the predefined timing functions as their cubic Bezier timing curves.

Figure 1 Cubic Bezier curve representations of the predefined timing functions



Custom timing functions are created using the `functionWithControlPoints::::` class method or the `initWithControlPoints::::` instance method. The end points of the Bezier curve are automatically set to (0.0,0.0) and (1.0,1.0). and the creation methods expect the `c1x`, `c1y`, `c2x`, and `c2y` as the parameters. The resulting control points defining the bezier curve are: `[(0.0,0.0), (c1x,c1y), (c2x,c2y), (1.0,1.0)]`.

Listing 1 shows an example code fragment that creates a custom timing function using the points `[(0.0,0.0), (0.25,0.1), (0.25,0.1), (1.0,1.0)]`.

Listing 1 Custom CAMediaTimingFunction code fragment

```
CAMediaTimingFunction *customTimingFunction;

customTimingFunction=[CAMediaTimingFunction functionWithControlPoints:0.25f :0.1f
 :0.25f :1.0f];
```

> **Note:** Keyframe animation requires a more nuanced pacing and timing model than can be provided by a single instance of `CAMediaTimingFunction`. See Keyframe Timing and Pacing Extensions (page 16) for more information.

Animation Delegates

The `CAAnimation` class provides the means to notify a delegate object when an animation starts and stops.

If an animation has a delegate specified it will receive `animationDidStart:` message, passing the animation instance that began. When an animation stops the delegate receives an `animationDidStop:finished:` message, passing the animation instance that stopped and a Boolean indicating whether the animation completed its duration successfully or was stopped manually.

> **Important:** The `CAAnimation` delegate object is retained by the receiver. This is a rare exception to the memory management rules described in *Advanced Memory Management Programming Guide*.

Property-Based Animations

Property-based animations are animations that interpolate values of a single attribute of a layer, for example, the position, background color, bounds, etc.

This chapter discusses how Core Animation abstracts property animation and the classes it provides to support basic and multiple keyframe animation of layer properties.

Property-Based Abstraction

The `CAPropertyAnimation` class is the abstract subclass of `CAAnimation` that provides the base functionality for animating a specific property of a layer. Property-based animations are supported for all value types that can be mathematically interpolated, including:

- integers and doubles
- `CGRect`, `CGPoint`, `CGSize`, and `CGAffineTransform` structures
- `CATransform3D` data structures
- `CGColor` and `CGImage` references

As with all animations, a property animation must be associated with a layer. The property that is to be animated is specified using a key-value coding key path relative to the layer. For example, to animate the x component of the `position` property of "layerA" you'd create an animation using the key path "position.x" and add that animation to "layerA".

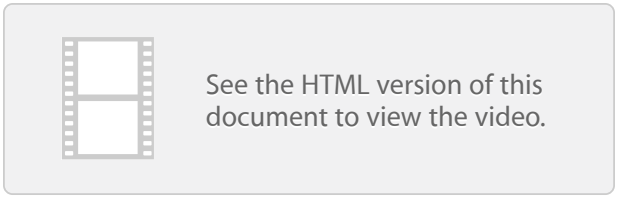
You will never need to instantiate an instance of `CAPropertyAnimation` directly. Instead you would create an instance of one of its subclasses, `CABasicAnimation` or `CAKeyframeAnimation`. Likewise, you would never subclass `CAPropertyAnimation`, instead you would subclass `CABasicAnimation` or `CAKeyframeAnimation` to add functionality or store additional properties.

Basic Animations

The `CABasicAnimation` class provides basic, single-keyframe animation capabilities for a layer property. You create an instance of `CABasicAnimation` using the inherited `animationWithKeyPath:` method, specifying the key path of the layer property to be animated. Animatable Properties in *Core Animation Programming Guide* summarize the animatable properties for `CALayer` and its filter properties.

Figure 1 shows a 3-second animation of a layer's position property from (74.0,74.0) to a final position of (566.0,406.0). The parent layer is assumed to have a bounds of (0.0,0.0,640.0,480.0).

Figure 1 3 second basic animation of a layer's position property



Configuring a Basic Animation's Interpolation Values

The `fromValue`, `byValue` and `toValue` properties of the `CABasicAnimation` class define the values being interpolated between. All are optional, and no more than two should be non-`nil`. The object type that the property is set to should match the type of the property being animated.

The interpolation values are used as follows:

- Both `fromValue` and `toValue` are non-`nil`. Interpolates between `fromValue` and `toValue`.
- `fromValue` and `byValue` are non-`nil`. Interpolates between `fromValue` and (`fromValue` + `byValue`).
- `byValue` and `toValue` are non-`nil`. Interpolates between (`toValue` - `byValue`) and `toValue`.
- `fromValue` is non-`nil`. Interpolates between `fromValue` and the current presentation value of the property.
- `toValue` is non-`nil`. Interpolates between the current value of `keyPath` in the target layer's presentation layer and `toValue`.
- `byValue` is non-`nil`. Interpolates between the current value of `keyPath` in the target layer's presentation layer and that value plus `byValue`.
- All properties are `nil`. Interpolates between the previous value of `keyPath` in the target layer's presentation layer and the current value of `keyPath` in the target layer's presentation layer.

Note: On OS X the values passed to these properties are `NSPoint` structures. However, on iOS the types are `CGPoints`. Aside from that small difference, their use is identical.

An Example Basic Animation

Listing 1 shows a code fragment that implements an explicit animation equivalent to the animation in Figure 1.

Listing 1 CABasicAnimation code fragment

```
CABasicAnimation *theAnimation;

// create the animation object, specifying the position property as the key path
// the key path is relative to the target animation object (in this case a CALayer)
theAnimation=[CABasicAnimation animationWithKeyPath:@"position"];

// set the fromValue and toValue to the appropriate points
theAnimation.fromValue=[NSValue valueWithPoint:NSMakePoint(74.0,74.0)];
theAnimation.toValue=[NSValue valueWithPoint:NSMakePoint(566.0,406.0)];

// set the duration to 3.0 seconds
theAnimation.duration=3.0;

// set a custom timing function
theAnimation.timingFunction=[CAMediaTimingFunction functionWithControlPoints:0.25f
 :0.1f :0.25f :1.0f];
```

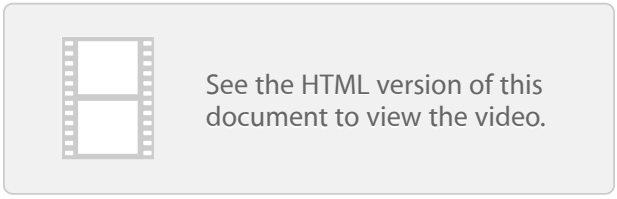
Note: This example is for OS X. When compiling under iOS a small change is required. The `NSMakePoint` function is not available, instead use the `CGPointMake` function. Aside from that direct substitution the code is identical.

Keyframe Animations

Keyframe animation, supported in Core Animation by the `CAKeyframeAnimation` class, is similar to basic animation; however it allows you to specify an array of target values. Each of these target values is interpolated, in turn, over the duration of the animation.

Figure 2 shows a 5-second animation of a layer's position property using a CGPathRef for the keyframe values.

Figure 2 5 second keyframe animation of a layer's position property



Providing Keyframe Values

Keyframe values are specified using one of two properties: a Core Graphics path (the `path` property) or an array of objects (the `values` property).

A Core Graphics path is suitable for animating a layer's `anchorPoint` or `position` properties, that is, properties that are `CGPoints`. Each point in the path, except for `moveto` points, defines a single keyframe segment for the purpose of timing and interpolation. If the `path` property is specified, the `values` property is ignored.

By default, as a layer is animated along a CGPath it maintains the rotation to which it has been set. Setting the `rotationMode` property to `kCAAnimationRotateAuto` or `kCAAnimationRotateAutoReverse` causes the layer to rotate to match the path tangent.

Providing an array of objects to the `values` property allows you to animate any type of layer property. For example:

- Provide an array of `CGImage` objects and set the animation key path to the `content` property of a layer. This causes the content of the layer to animate through the provided images.
- Provide an array of `CGRects` (wrapped as objects) and set the animation key path to the `frame` property of a layer. This causes the frame of the layer to iterate through the provided rectangles.

- Provide an array of `CATransform3D` matrices (again, wrapped as objects) and set the `animation` key path to the `transform` property. This causes each transform matrix to be applied to the layer's `transform` property in turn.

Keyframe Timing and Pacing Extensions

Keyframe animation requires a more complex timing and pacing model than that of a basic animation.

The inherited `timingFunction` property is ignored. Instead you can pass an optional array of `CAMediaTimingFunction` instances in the `timingFunctions` property. Each timing function describes the pacing of one keyframe to keyframe segment.

While the inherited duration property is valid for `CAKeyframeAnimation`, you can attain more subtle control of timing by using the `keyTimes` property.

The `keyTimes` property specifies an array of `NSNumber` objects that define the duration of each keyframe segment. Each value in the array is a floating point number between 0.0 and 1.0 and corresponds to one element in the `values` array. Each element in the `keyTimes` array defines the duration of the corresponding keyframe value as a fraction of the total duration of the animation. Each element value must be greater than, or equal to, the previous value.

The appropriate values for the `keyTimes` array are dependent on the `calculationMode` property.

- If the `calculationMode` is set to `kCAAnimationLinear`, the first value in the array must be 0.0 and the last value must be 1.0. Values are interpolated between the specified keytimes.
- If the `calculationMode` is set to `kCAAnimationDiscrete`, the first value in the array must be 0.0.
- If the calculationMode is set to kCAAnimationPaced, the keyTimes array is ignored.

An Example Keyframe Animation

Listing 2 shows a code fragment that implements an explicit animation equivalent to the animation in Figure 2.

Listing 2 CAKeyframeAnimation code fragment

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

```
                                    320.0,74.0);
CGPathAddCurveToPoint(thePath,NULL,320.0,500.0,
                                    566.0,500.0,
                                    566.0,74.0);

CAKeyframeAnimation * theAnimation;

// create the animation object, specifying the position property as the key path
// the key path is relative to the target animation object (in this case a CALayer)
theAnimation=[CAKeyframeAnimation animationWithKeyPath:@"position"];
theAnimation.path=thePath;

// set the duration to 5.0 seconds
theAnimation.duration=5.0;



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

Transition Animation

Transition animation is used when it is impossible to mathematically interpolate the effect of changing the value of a layer property, or the state of a layer in the layer tree.

This chapter discusses the transition animation functionality provided by Core Animation.

Creating a Transition Animation

The `CATransition` class provides transition functionality to Core Animation. It is a direct subclass of `CAAnimation` as it affects an entire layer, rather than a specific property of a layer.

A new instance of `CATransition` is created using the inherited class method `animation`. This will create a transition animation with the default values shown in Table 1:

Table 1 Default `CATransition` property values

Transition Property	Value
`type`	Uses a fade transition. The value is `kCATransitionFade`.
`subType`	Not applicable.
`duration`	Uses the duration of the current transaction or 0.25 seconds if the duration has not been set for a transaction. The value is 0.0
`timingFunction`	Uses linear pacing. The value is `nil`.
`startProgress`	0.0
`endProgress`	1.0

Configuring a Transition

Once created, you can configure the transition animation using one of the predefined transition types or, on OS X, create a custom transition using a Core Image filter.

The predefined transitions are used by setting the `type` property to one of the constants in Table 2.

Table 2 Common transition types

Transition Type	Description
`kCATransitionFade`	The layer fades as it becomes visible or hidden.
`kCATransitionMoveIn`	The layer slides into place over any existing content.
`kCATransitionPush`	The layer pushes any existing content as it slides into place
`kCATransitionReveal`	The layer is gradually revealed in the direction specified by the transition subtype.

With the exception of `kCATransitionFade`, the predefined transition types also allow you to specify a direction for the transition by setting the `subType` property to one of the constants in Table 3.

Table 3 Common transition subtypes

Transition Subtype Constant	Description
`kCATransitionFromRight`	The transition begins at the right side of the layer.
`kCATransitionFromLeft`	The transition begins at the left side of the layer.
`kCATransitionFromTop`	The transition begins at the top of the layer.
`kCATransitionFromBottom`	The transition begins at the bottom of the layer.

The `startProgress` property allows you to change the start point of the transition by setting a value that represents a fraction of the entire animation. For example, to start a transition half way through its progress the `startProgress` value would be set to 0.5. Similarly, you can specify the `endProgress` value for the transition. The `endProgress` is the fraction of the entire transition that the transition should stop at. The default values are 0.0 and 1.0, respectively.

If the predefined transitions don't provide the desired visual effect and your application is targeted to OS X rather than iOS, you can specify a custom Core Image filter object that is used to display the transition. A custom filter must support both the `kCIInputImageKey` and the `kCIInputTargetImageKey` input keys, and the `kCIOutputImageKey` output key. The filter may optionally support the `kCIInputExtentKey` input key, which is set to a rectangle describing the region in which the transition should run.

Document Revision History

This table describes the changes to *Animation Types and Timing Programming Guide*.

Date	Notes
2010-05-18	Called out iOS difference for NSMakePoint usage in example.
2010-03-24	Updated movies.
2008-09-09	Corrected typos.
2008-07-08	Updated for iOS.
2008-04-08	Corrected typos.
2008-02-08	Corrected timing function image.
2007-10-31	New document that describes the animation and timing classes used by both Core Animation and Cocoa Animation proxies.

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