Animation Programming Guide for Cocoa

Graphics & Animation: Animation

2006-05-23

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction to Animation Programming Guide for Cocoa

Cocoa provides facilities for animating certain types of operations over a finite or indefinite amount of time. The basic animation support provided by the `NSAnimation` class focuses on providing you with a source for animation timing and management. Although the word "animation" may make you think of cartoons or other forms of movies, animation objects are more designed for animating portions of your program's user interface. For example, you can use the `NSViewAnimation` class (a subclass of `NSAnimation`) to create smooth transitions in the size, position, or opacity of a view or window. This animated appearance lets you create a user interface with a more fluid appearance.

This document describes the fundamental concepts involved in using Cocoa animation objects and also provides examples of how to use them in your own applications.

Organization of This Document

This document contains the following articles:

- "Using an NSAnimation Object" (page 9) describes the basic features of animation objects and how you customize them.
- "Animating Views and Windows" (page 15) describes the use of view animation objects, which provide a high-level interface for smoothly resizing, repositioning, and changing the opacity of view and window objects.

See Also

Sample code is available that provides examples for using the Cocoa animation classes:

- *Reducer* implements a reusable collapsible view (using `NSViewAnimation`) and an animated tab view class subclass of `NSAnimation`.
- *iSpend* implements an expanding view using `NSViewAnimation`.

Using an NSAnimation Object

The `NSAnimation` class provides sophisticated behavior for animations that occur over a finite length of time. Mac OS X uses animation objects to implement transition animations for user interface elements. You can define custom animation objects to implement animations for your own code. Unlike `NSTimer`, animation notifications can occur at irregular intervals, allowing you to create animations that appear to speed up or slow down.

The sections that follow cover the basic steps for creating a custom `NSAnimation` object and using it to manage your animated content. If you want to animate your views and windows, you should see if the `NSViewAnimation` class (which is a subclass of `NSAnimation`) offer the behavior you need. View animation objects provide sophisticated behavior for resizing and moving views over time and are described in "Animating Views and Windows" (page 15).

Note: Animation objects are available in Mac OS X v10.4 and later.

Creating and Configuring an Animation Timer

An `NSAnimation` object has several important attributes:

- Current progress—A value between 0.0 and 1.0 that indicates the percentage of the animation completed.
- Frame rate—The number of updates per second.
- Duration—The period (in seconds) over which the animation occurs.
- Animation curve—The relative speed of the animation over its course; for example, the animation could slowly speed up at the beginning, gradually slow down near its end, or remain the same speed throughout.
- Blocking mode—The mode in which the animation runs in terms of the application's responsiveness to user actions.

When you configure a new `NSAnimation` object, you must, at a minimum, set its duration, animation curve, frame rate, and blocking mode attributes. You should also assign a delegate to monitor the progress of the animation. When the animation begins, ends, is explicitly stopped, or reaches a progress mark, the animation object sends a message to the current delegate. (See "Setting and Handling Progress Marks" (page 10) for information about progress marks). If you do not want to use a delegate, you must subclass `NSAnimation` to receive progress information; see "Subclassing NSAnimation" (page 11).

Listing 1 shows a sample method that creates and configures a standard `NSAnimation` object. The object that created the animation acts as the delegate and handles any progress messages.

Listing 1 Initializing an NSAnimation object

```
- (id)init
{
```

```
    self = [super init];
    if (self)
    {
        // theAnim is an NSAnimation instance variable.
        theAnim = [[NSAnimation alloc] initWithDuration:10.0
                                         animationCurve:NSAnimationEaseIn];
        [theAnim setFrameRate:20.0];
        [theAnim setAnimationBlockingMode:NSAnimationNonblocking];
        [theAnim setDelegate:self];
    }
    return self;
}
```

The `initWithDuration:animationCurve:` method is the designated initializer for the `NSAnimation` class. This method lets you set two of the animation attributes. For the other attributes, you can use the default values or set the attribute value explicitly using the appropriate accessor methods. The default attributes are as follows:

- The default animation curve is `NSAnimationEaseInOut`.
- The default blocking mode is `NSAnimationBlocking`.
- The default frame rate is a reasonable value. This frame rate is usually 60 Hz, but the exact value should not be relied upon.

Once you have prepared an `NSAnimation` object for use, you can run it by sending it a `startAnimation` message. If you need to stop it before the animation completes its scheduled duration, send the object a `stopAnimation` message. The delegate of the `NSAnimation` object (if one exists) receives messages informing it of both of these events, as well as a message that tells it if the animation completed as scheduled.

Setting and Handling Progress Marks

`NSAnimation` has the notion of **progress marks**—floating-point values (of type `NSAnimationProgress`) that indicate the percentage amount of the animation that is complete. When you start an animation and it reaches a progress mark (specifically, its current progress is equal to the progress mark), the animation object sends a message to its delegate. The delegate can then update a custom progress indicator, play a sound, or accomplish some other effect appropriate to that point of the animation.

Important: Although you can use progress marks to "time-slice" the animation of an object, it is not an ideal way to achieve a smooth animation. A recommended alternative is to subclass `NSAnimation` and redraw an object at each change of frame; see "Smooth Animations" (page 12) for more information.

Usually you set the progress marks for an animation object when you first create and initialize the object. Listing 2 shows one approach that sets 20 equally spaced progress marks.

Listing 2 Setting the progress marks of an NSAnimation object

```
- (void)awakeFromNib
{
     NSAnimationProgress progMarks[] = {
            0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5,
            0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0  };
```

```
    int i, count = 20;
    // theAnim is an NSAnimation instance variable
    theAnim = [[NSAnimation alloc] initWithDuration:10.0
                                   animationCurve:NSAnimationEaseInOut];
    [theAnim setFrameRate:20.0];
    [theAnim setDelegate:self];

    for (i=0; i<count; i++)
        [theAnim addProgressMark:progMarks[i]];
}
```

Instead of adding progress-mark values in a loop, as in this example, you can set them in one invocation by using the `setProgressMarks:` method, which takes an array of `NSNumber` objects encapsulating `float` values.

When a running animation object reaches a progress mark, it sends an `animation:didReachProgressMark:` message to its delegate. The delegate should handle this message in a way appropriate to the progress mark passed in. Listing 3 illustrates how the delegate implements this method to play a train sound at regular intervals.

Listing 3 Delegate implementation of animation:didReachProgressMark:

```
- (void)animation:(NSAnimation *)animation
            didReachProgressMark:(NSAnimationProgress)progress
{
    if (animation == theAnim)
        [[NSSound soundNamed:@"chug"] play];
}
```

Subclassing NSAnimation

Although you can use an `NSAnimation` object as-is for many purposes, subclassing it is a more common scenario. There are three major reasons to subclass `NSAnimation`:

- To achieve smooth animations by redrawing at per-frame intervals
- To specify valid run-loop modes when running an animation on the main thread in nonblocking mode
- To return custom curve values without the overhead of a delegate that responds to `animation:valueForProgress:`

The procedures for accomplishing the first two of these objectives are described in the following sections. To return custom curve values without implementing the delegate method, you must override the `currentValue` method. See the `NSAnimation` class documentation for further information.

Smooth Animations

As mentioned in "Setting and Handling Progress Marks" (page 10), you can attach a series of progress marks to an `NSAnimation` object and have the delegate implement the `animation:didReachProgressMark:` method to redraw an object at each progress mark. However, this is not the best way to animate an object. Unless you set a large number of progress marks (30 per second or more), the animation is probably going to appear jerky.

A better approach is to subclass `NSAnimation` and override the `setCurrentProgress:` method, as illustrated in Listing 4. The `NSAnimation` object invokes this method after each frame to change the progress value. By intercepting this message, you can perform any redrawing or updating you need for that frame. If you do override this method, be sure to invoke the implementation of `super` so that it can update the current progress.

Listing 4 Overriding the setCurrentProgress: method

```
- (void)setCurrentProgress:(NSAnimationProgress)progress
{
    // Call super to update the progress value.
    [super setCurrentProgress:progress];

    // Update the window position.
    NSRect theWinFrame = [[NSApp mainWindow] frame];
    NSRect theScreenFrame = [[NSScreen mainScreen] visibleFrame];
    theWinFrame.origin.x = progress *
            (theScreenFrame.size.width - theWinFrame.size.width);
    [[NSApp mainWindow] setFrame:theWinFrame display:YES animate:YES];
}
```

Custom Run-Loop Mode Sets

An `NSAnimation` object with a blocking mode of `NSAnimationNonblocking` runs in the main thread of the process in a run-loop mode that accepts user input. Before it runs the animation, the animation object sends itself a `runLoopModesForAnimation` message to get the currently valid run-loop modes. By default, this method returns `nil`, which tells `NSAnimation` to use the default mode (`NSDefaultRunLoopMode`), modal panel mode (`NSModalPanelRunLoopMode`), and event tracking run-loop mode (`NSEventTrackingRunLoopMode`).

You can override this method to return a different set of run loop modes, which can include custom modes. Listing 5 shows an implementation that returns the default array of modes minus the event-tracking mode (`NSEventTrackingRunLoopMode`).

Listing 5 Returning run-loop modes from runLoopModesForAnimating

```
- (NSArray *)runLoopModesForAnimating
{
    return [NSArray arrayWithObjects: NSDefaultRunLoopMode,
                          NSModalPanelRunLoopMode, nil];
}
```

Linking Animations

You can link two animation objects so that one of them starts running (or stops running) when the other reaches a specified animation mark. This feature of `NSAnimation` is useful for coordinating different effects. Listing 6 illustrates how the `startWhenAnimation:reachesProgress:` method is used to start an animation when another animation reaches the midway point.

Listing 6 Linking two animations

```
- (IBAction)startAnim:(id)sender
{
    // theAnim and theOtherAnim are variables of type NSAnimation.
    [theOtherAnim startWhenAnimation:theAnim reachesProgress:0.5];
    [theAnim startAnimation];
}
```

If you want instead to stop an animation when another animation reaches a progress mark, use the `stopWhenAnimation:reachesProgess:` method. You can link animations indefinitely, one after another. However, there can be only one “start” and one “stop” animation at any given time.

If you have a delegate that is responding to `animation:didReachProgressMark:` messages, it has to distinguish among the multiple animations, as in Listing 7.

Listing 7 Handling progress marks of simultaneously running animations

```
- (void)animation:(NSAnimation *)animation
            didReachProgressMark:(NSAnimationProgress)progress
{
    if (animation == theOtherAnim)
    {
        // Do an effect appropriate to progress mark.
    }
    else if (animation == theAnim)
    {
        // Do an effect appropriate to progress mark.
    }
}
```

Animating Views and Windows

The `NSViewAnimation` class is a subclass of `NSAnimation` that provides a convenient way to animate aspects of your view and window objects, including the following:

- Change the position of the frame.
- Change the size of the frame
- Change the opacity of the object and fade it in or out.

The View Animation Process

You use view animation objects in a slightly different way than you do regular `NSAnimation` objects. A single view animation object can manage the animation process for multiple views and windows simultaneously. Rather than setting the attributes using methods of the animation object, you instead create a dictionary of animation attributes for each view or window you want to modify. Each dictionary specifies the target of the action (the view or window), and the effects you want to apply to that target. To set other factors, such as the duration and timing curve of the animation, you continue to use the methods of `NSAnimation`.

An animation attributes dictionary has only one required value: the target object. You add this object to the dictionary using the `NSViewAnimationTargetKey` key. The presence of this key alone, though, does not change the view or window. To make changes, you must include one or more additional keys to specify the desired behavior.

You can perform multiple actions on a single target object simultaneously, if you choose. For example, you can resize a view, change its position on the screen, and fade it in or out all at once. The following sections show you how to perform each of these actions separately, for simplicity. To perform them both, simply add all of the relevant keys to the attributes dictionary.

Changing the Frame Rectangle

Changing the frame rectangle of a view or window lets you resize and reposition that object relative to its parent. In the case of views, this means changing the position and size of the view in its superview. In the case of windows, it means changing the position and size of the window on the desktop. Table 1 lists the keys and values you would put into the attributes dictionary to change the frame rectangle.

Table 1 Keys for resizing or repositioning a view or window.

Key	Value	Description
`NSViewAnimation-TargetKey`	`id`	Identifies the `NSView` or `NSWindow` object to resize or reposition. This key is required.

Key	Value	Description
`NSViewAnimation-StartFrameKey`	`NSValue`	Contains the starting frame rectangle of the target object. The `NSValue` object should contain an encoded `NSRect` data type. This value is typically equal to the current frame of the view or window. This key is optional and defaults to the current frame rectangle of the target object.
`NSViewAnimation-EndFrameKey`	`NSValue`	Contains the ending frame rectangle of the target object. The `NSValue` object should contain an encoded `NSRect` data type. This key is recommended; if not present, it defaults to the current frame rectangle of the target object.

Fading Objects In and Out

If you want to hide a view or window, rather than have the object suddenly disappear, you can use a view animation to make that object gradually fade away. Similarly, you can use a similar type of view animation to make the object visible again. When fading a view back in, the size of the ending frame rectangle must be non-zero; if it is not, the view remains hidden. Table 2 lists the keys and values you would put into the attributes dictionary to fade an object in or out.

Table 2 Keys for fading a view or window.

Key	Value	Description
`NSViewAnimation-TargetKey`	`id`	Identifies the `NSView` or `NSWindow` object to modify. This key is required.
`NSViewAnimation-EffectKey`	`NSString`	Contains one of the following string constants: `NSViewAnimationFadeInEffect` makes the object visible and `NSViewAnimationFadeOutEffect` hides it. These effects change the opacity of the object over the course of the animation.

A View Animation Example

Listing 1 illustrates the basic use of a view animation object. The action method sets up attribute dictionaries for two different view objects and then runs the animation whenever the action occurs. For the first view object, the animation object shifts the origin of the view by 50 units along each axis. For the second view, the animation object shrinks the frame size to zero while simultaneously fading the view out until it is completely hidden. The animation uses a custom timing curve and duration but uses the default blocking mode, which blocks user input on the main thread until the animation is complete.

Listing 1 Animating two NSView objects

```
- (IBAction)startAnimations:(id)sender
{
    // firstView, secondView are outlets
    NSViewAnimation *theAnim;
    NSRect firstViewFrame;
```

```
    NSRect newViewFrame;
    NSMutableDictionary* firstViewDict;
    NSMutableDictionary* secondViewDict;

    {
        // Create the attributes dictionary for the first view.
        firstViewDict = [NSMutableDictionary dictionaryWithCapacity:3];
        firstViewFrame = [firstView frame];

        // Specify which view to modify.
        [firstViewDict setObject:firstView forKey:NSViewAnimationTargetKey];

        // Specify the starting position of the view.
        [firstViewDict setObject:[NSValue valueWithRect:firstViewFrame]
                 forKey:NSViewAnimationStartFrameKey];

        // Change the ending position of the view.
        newViewFrame = firstViewFrame;
        newViewFrame.origin.x += 50;
        newViewFrame.origin.y += 50;
        [firstViewDict setObject:[NSValue valueWithRect:newViewFrame]
                 forKey:NSViewAnimationEndFrameKey];
    }

    {
        // Create the attributes dictionary for the second view.
        secondViewDict = [NSMutableDictionary dictionaryWithCapacity:3];

        // Set the target object to the second view.
        [secondViewDict setObject:secondView forKey:NSViewAnimationTargetKey];

        // Shrink the view from its current size to nothing.
        NSRect viewZeroSize = [secondView frame];
        viewZeroSize.size.width = 0;
        viewZeroSize.size.height = 0;
        [secondViewDict setObject:[NSValue valueWithRect:viewZeroSize]
                 forKey:NSViewAnimationEndFrameKey];

        // Set this view to fade out
        [secondViewDict setObject:NSViewAnimationFadeOutEffect
                forKey:NSViewAnimationEffectKey];
    }

    // Create the view animation object.
    theAnim = [[NSViewAnimation alloc] initWithViewAnimations:[NSArray
                arrayWithObjects:firstViewDict, secondViewDict, nil]];

    // Set some additional attributes for the animation.
    [theAnim setDuration:1.5];    // One and a half seconds.
    [theAnim setAnimationCurve:NSAnimationEaseIn];

    // Run the animation.
    [theAnim startAnimation];

    // The animation has finished, so go ahead and release it.
    [theAnim release];
}
```

Document Revision History

This table describes the changes to *Animation Programming Guide for Cocoa*.

Date	Notes
2006-05-23	Corrected source code in "Animating Views and Windows."
2006-04-04	New document that describes the use of Cocoa animation objects.