Extracted from:

Core Animation for Mac OS X and the iPhone

Creating Compelling Dynamic User Interfaces

This PDF file contains pages extracted from Core Animation for Mac OS X and the iPhone, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2008 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The aim, if reached or not, makes great the life: Try to be Shakespeare, leave the rest to fate!

► Robert Browning

Chapter 3

Animation Types

In the previous chapter, we saw how easy it is to add a default animation to our applications. Even this type of animation looks great, but we don't have much control. All we can do is set a new value and watch what happens. In this chapter, we are going to dig into part of what is going on behind the scenes so that we can get precisely the animation we want.

Up to this point, we have seen only AppKit APIs, but in this chapter we are going to start to mix in the animation classes from Core Animation. The animation classes are tightly integrated into AppKit, which makes it possible for us to use much of Core Animation without having to get into the details of the whole framework. The animation classes give us a great deal of control over what our effects look like and how they play out onscreen.

3.1 Basic Animation

In the example in Section 2.3, *Finding Animations*, on page 25, we saw the CABasicAnimation class in action. This basic animation was responsible for the smooth transition from the left to the right of the screen. The basic animation is just that: basic. We can use it for all the simple stuff that you don't want to think too much about. When we are simply moving a view from one side of the screen to the other or scaling the view (we did both in the previous example, if you recall), we can leave everything to the defaults to get the basic animation.

For many of the things you want to animate, this animation fits the bill because it's easy to use since every property has reasonable defaults. However, easy does not always lead to the effect you want. Several other

types of animations have more advanced capabilities that we can exploit to make some truly fantastic-looking (and functional) user interfaces. Let's dig into the other animations types now.

3.2 Keyframe Animations

Placing an animated item precisely where we want it when we want it to be there takes more than a basic animation. That kind of functionality is accomplished by using the keyframe animation. With it we can specify exactly what value we want our animated property to have and exactly the length of time we want the property to take to reach that value.

The term *keyframe* comes from the animation and motion graphics world (and should not be confused with the *key* term used with key-value coding). If you are familiar with Apple's Final Cut Studio suite, which includes Motion and Final Cut Pro, you have probably played with keyframes. If not, a keyframe is basically a reference point around which interpolation happens. The keyframe specifies a point in the animation that is precise and does not depend on an interpolation function. In a program like Motion, you create a keyframe by specifying a location and a time, and Motion takes care of interpolating the "before" and "after" sequence. If you specify multiple keyframes, then Motion will interpolate between each of them, making sure to hit your specific locations at the specified time. In Core Animation, we do basically the same thing; we specify a value at a time, and Core Animation will interpolate between the keyframes.

Let's say we want the opacity of an image to fade from zero to 75% and then back down to zero over the course of an animation (in other words, the image fades in and then fades back out). In addition, we also want the opacity to remain zero until 25% of the time has passed and then be back at zero when 75% of the time has passed. The only way to make this happen is with a keyframe animation.

The curve is shown in Figure 3.1, on the following page. The horizontal axis is the time span, and the vertical axis is the opacity. The initial diamond is the point in time where the image starts to fade in, and then until the next diamond, the opacity is smoothly increasing. Then as the animation hits that second point, the opacity begins to decrease until the animation finishes at the final diamond back at zero opacity.

CLICK HERE to purchase this book now.

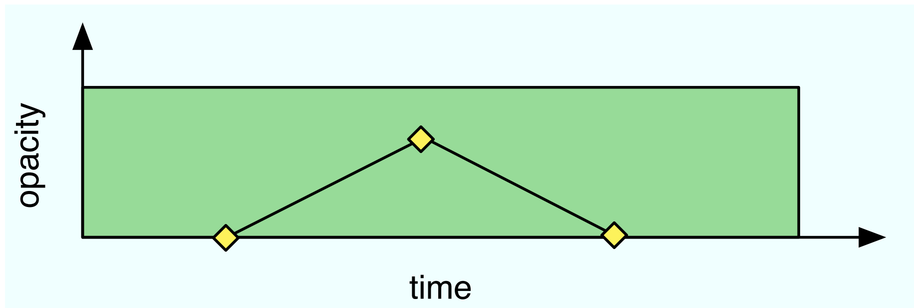

Figure 3.1: Opacity keyframe example

Keyframes are specified by providing an array of values, one value for each specific keyframe we want to set during the animation. For example, each of the diamonds in Figure 3.1 is a keyframe for opacity, with the opacity values being (0.0, 0.75, 0.0).

Another important thing to keep in mind about keyframe animations is that they work in terms of "normalized" time. The total duration of the keyframe animation is specified in seconds, but the keyframe begin and end points are specified as percentages of the total duration. The beginning of time for the animation is 0, and the end of time is 1. So, we can think of points in time as a percentage of completion. In other words, 0.5 is half of the time frame for the animation, regardless of how long it really runs.

Let's consider further changing the opacity as in the graph in Figure 3.1. The array of values, as we said, are (0.0, 0.75, 0.0), and the time values are (0.25, 0.50, 0.75).

At the beginning of the animation, the alpha value is 0.0, and the value remains at zero until the animation arrives at 25% of its duration. Between 25% of the duration and 50% of the duration, the opacity will continue to rise smoothly until the duration reaches 50%. At that point, the opacity will be 75%, and 50% of the duration will have elapsed. As the animation proceeds to 75% of its duration, the opacity will fade back to 0% where it will remain through the final 25% of the duration.

CLICK HERE to purchase this book now.

Keyframes in Keynote

One of the new features in Keynote 4 (for iWork '08) is to animate objects along a path. One demo during the intro of the new version shows an airplane moving along a curved bezier path. The CAKeyframeAnimation class is intended to allow us to build that very same kind of animation into our applications.

The code to create the keyframe animation would look like this:

```
- (CAKeyframeAnimation *)opacityAnimation {
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
    animation.values = [NSArray arrayWithObjects:
                       [NSNumber numberWithFloat:0.0],
                       [NSNumber numberWithFloat:0.75],
                       [NSNumber numberWithFloat:0.0], nil];
    animation.keyTimes = [NSArray arrayWithObjects:
                         [NSNumber numberWithFloat:0.25],
                         [NSNumber numberWithFloat:0.50],
                         [NSNumber numberWithFloat:0.75], nil];
    return animation;
}
```

One of the things we can do to eliminate some of the complexity and still get most of the control is to let the keyframe animation handle the timing. If we leave out setting the time values, the keyframe animation will just evenly distribute the values we provide over the time frame. If we provide three values, the first value is the starting value, the second value will be reached at 50% of the elapsed time, and the third value is at 100% of the time.

Keyframes and Paths

In addition to allowing you to set the *key values* along an animation keyframe, animations also allow you to use paths to animate dual-valued properties such as the position of a layer (position has both x and y values). For example, suppose we want to move a picture along a nonlinear path. All we have to do is create a path that plots the points precisely the way we want them and supply that to the animation. The path will then be used to determine values instead of an interpolation.

CLICK HERE to purchase this book now.

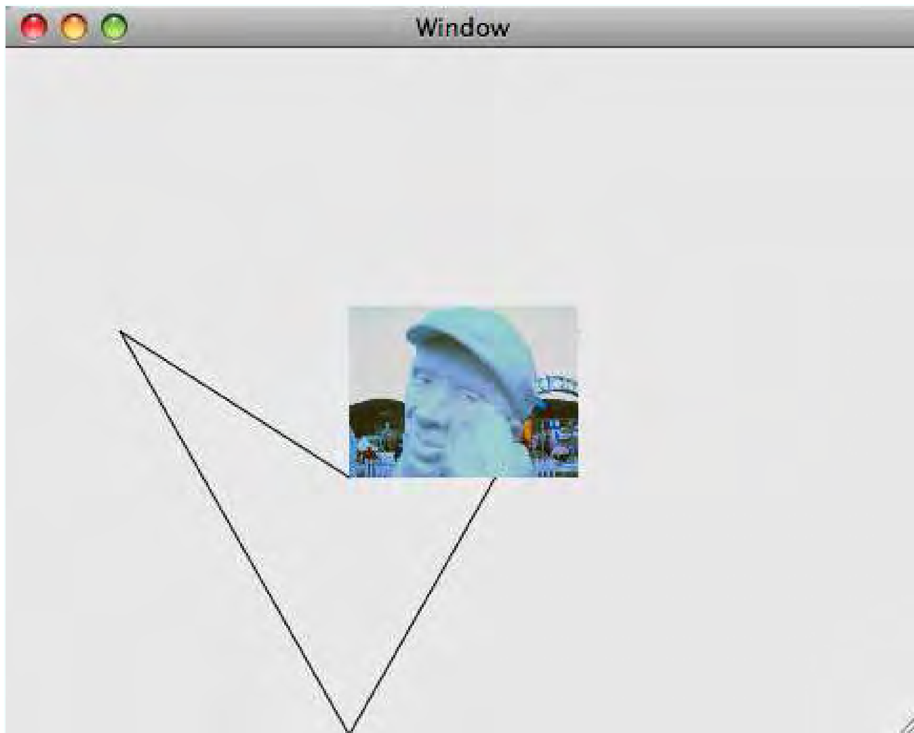

Figure 3.2: Keyframe movement

The path technique will work only with dual-valued properties, that is, any property with two values. Basically, this is any property that is typed as NSPoint or NSSize. The x values in the path are used by the animation to change either the x value of the point or the width value of the size, and the y values correspond to the y value of the point or the height value of the size. The next example will show this in action.

We will use this technique to move a view around the screen with a keyframe animation. We will use an instance of CGPath to animate the frameOrigin property of the picture in the pointed heart shape shown in Figure 3.2. The picture moves across the screen following the path. The heart-shaped path that is used for the keyframing is also drawn in the background so we can see where the picture is and where it is headed.

CLICK HERE to purchase this book now.

Let's look at the code that makes all this happen:

Download AnimationTypes/KeyFrameMoveAView/KeyFrameView.m

```
Line 1  - (void)addBounceAnimation {
     -    [mover setAnimations:[NSDictionary dictionaryWithObjectsAndKeys:
     -                          self.originAnimation, @"frameOrigin", nil]];
     -  }
     5
     -  - (id)initWithFrame:(NSRect)frame {
     -    self = [super initWithFrame:frame];
     -    if (self) {
     -      // inset by 3/8's
    10      CGFloat xInset = 3.0f * (NSWidth(frame) / 8.0f);
     -      CGFloat yInset = 3.0f * (NSHeight(frame) / 8.0f);
     -      NSRect moverFrame = NSInsetRect(frame, xInset, yInset);
     -      mover = [[NSImageView alloc] initWithFrame:moverFrame];
     -      [mover setImageScaling:NSScaleToFit];
    15      [mover setImage:[NSImage imageNamed:@"photo.jpg"]];
     -      [self addSubview:mover];
     -      [self addBounceAnimation];
     -    }
     -    return self;
    20  }
```

In this first bit of code, we are initializing the mover view (the NSImageView that holds our picture) to the center of the screen, and then on line 17 we add the animation to the view. Recall from Section 2.3, *Finding Animations*, on page 25 that adding an animation to the animations dictionary will place our custom animation in the search path. This of course will cause our animation to be used instead of the default. Next let's look at the animation creation code:

Download AnimationTypes/KeyFrameMoveAView/KeyFrameView.m

```
Line 1  - (CAKeyframeAnimation *)originAnimation {
     -    CAKeyframeAnimation *originAnimation = [CAKeyframeAnimation animation];
     -    originAnimation.path = self.heartPath;
     -    originAnimation.duration = 2.0f;
     5    originAnimation.calculationMode = kCAAnimationPaced;
     -    return originAnimation;
     -  }
```

We should pay attention to two parts of this code. First on line 3, we are setting the path to the animation. Recall that the animation will use the x and y values of this path as the x and y values of the frameOrigin for our moving view. The next thing to notice is on line 5, where we set the calculationMode property of the animation. Setting this value to kCAAnimationPaced causes the animation to equally distribute the time across the whole path. By default, a keyframe animation will distribute the

CLICK HERE to purchase this book now.

time equally across the path fragments so each path fragment would have the same amount of time to move the view along. This makes the long path fragments move the view quickly and the short segments move the view slowly. We've chosen instead to make our entire journey at a constant speed. Now let's look at the way to create the path:

Download AnimationTypes/KeyFrameMoveAView/KeyFrameView.m

```
- (CGPathRef)heartPath {
  NSRect frame = [mover frame];
  if(heartPath == NULL) {
    heartPath = CGPathCreateMutable();
    CGPathMoveToPoint(heartPath, NULL, NSMinX(frame), NSMinY(frame));
    CGPathAddLineToPoint(heartPath, NULL, NSMinX(frame) - NSWidth(frame),
                         NSMinY(frame) + NSHeight(frame) * 0.85);
    CGPathAddLineToPoint(heartPath, NULL, NSMinX(frame),
                         NSMinY(frame) - NSHeight(frame) * 1.5);
    CGPathAddLineToPoint(heartPath, NULL, NSMinX(frame) + NSWidth(frame),
                         NSMinY(frame) + NSHeight(frame) * 0.85);
    CGPathAddLineToPoint(heartPath, NULL, NSMinX(frame), NSMinY(frame));
    CGPathCloseSubpath(heartPath);
  }
  return heartPath;
}
```

This is very typical Quartz path creation code. If you want more information, look at [GL06]. Next up let's look at the animation code:

Download AnimationTypes/KeyFrameMoveAView/KeyFrameView.m

```
Line 1  - (void)bounce {
     -    NSRect rect = [mover frame];
     -    [[mover animator] setFrameOrigin:rect.origin];
     -  }
```

It's fairly simple code here too. All we do is call the setFrameOrigin: method on our moving view, and the animation takes care of the rest (bounce is called from keyDown: when any key is hit). Recall that since we have added an animation to the animations dictionary under the frameOrigin key, the animator will find it during its search and use ours instead of the default animation. Also notice that we are setting the frame origin with its current value. Since we want the animation to end up back where it started, this is expected. If we wanted to have the view animate along a path to another location, we'd set that new location here. But we would have to be careful to make sure the path's final point matches with the destination we set; otherwise, we'd get a choppy animation. You can find more detail on this later in Section 3.5, *Custom Animation and Interpolation*, on page 45.

CLICK HERE to purchase this book now.

As you can see, keyframe animations give us a very fine-grained level of control over the various properties we seek to animate. We can specify as many intermediate points as needed to achieve the effect we want. We also have complete control over the time intervals spent for each section of our path.

3.3 Grouping Animations

Animations can be grouped and then triggered with the change of a single attribute. For example, we could group an alpha fade, a frame movement, and a resize. Then add the group animation to the view so that it is triggered when the frame origin is set.

When working with groups, you need to set the keyPath for each of the animations that is part of the group. The group is added (not the individual animation objects) to the animations dictionary so it is discovered and run as discussed in Section 2.3, *Finding Animations*, on page 25. However, the constituent animations are not associated with any particular key (since they are in the group and not part of the animations dictionary) and thus will not cause any animation. So, we need to set their keyPaths. To do that, we can use animationWithKeyPath: at animation creation time, or we can set the keyPath property after creation. Don't worry if this is not crystal clear right now; you will see in the example how to set this up.

Another interesting point about the keys an animation is associated with is that they do not have to be associated with a key on the view they are animating. The animation is associated with a keyPath, so the animation can affect anything that is reachable from the view via a keyPath. For the typical use, we will be using a simple key (frameOrigin, frameSize, and so on), but when we start looking at filters later in Chapter 7, *Core Animation*, on page 83, we will use the keyPath to animate the properties of the filter.

Any of the animations we've seen thus far can be put into groups. The combinations of what is possible are almost endless. We can even put a group into another group and have lots and lots of animations happening at the same time. Of course, we have to temper our imaginations by making sure that the animation is useful and not just eye candy.

Here's an example of nesting animations into a group in which we rotate and enlarge a picture. It starts at the center of the screen, grows and rotates, and then returns to its original position.

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Core Animation for Mac OS X and the iPhone's Home Page
http://pragprog.com/titles/bdcora
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/bdcora.

Contact Us

Phone Orders: 1-800-699-PROG (+1 919 847 3884)
Online Orders: www.pragprog.com/catalog
Customer Service: orders@pragprog.com
Non-English Versions: translations@pragprog.com
Pragmatic Teaching: academic@pragprog.com
Author Proposals: proposals@pragprog.com