Advanced Animation Tricks

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

Transition Animations Support Changes to Layer Visibility

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Customizing the Timing of an Animation

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

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

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

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

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

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

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

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

Pausing and Resuming Animations

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

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

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


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

Explicit Transactions Let You Change Animation Parameters

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

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

Listing 5-5 Creating an explicit transaction

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

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

Listing 5-6 Changing the default duration of animations

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

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

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

Listing 5-7 Nesting explicit transactions

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

// Change the animation duration to two seconds
[CATransaction setValue:[NSNumber numberWithFloat:2.0f]
```

```
                forKey:kCATransactionAnimationDuration];
// Move the layer to a new position
theLayer.position = CGPointMake(0.0,0.0);

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

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

[CATransaction commit]; // Inner transaction

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

Adding Perspective to Your Animations

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

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

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

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

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

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

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