Animations

Animations provide fluid visual transitions between different states of your user interface. In iOS, animations are used extensively to reposition views, change their size, remove them from view hierarchies, and hide them. You might use animations to convey feedback to the user or to implement interesting visual effects.

In iOS, creating sophisticated animations does not require you to write any drawing code. All of the animation techniques described in this chapter use the built-in support provided by Core Animation. All you have to do is trigger the animation and let Core Animation handle the rendering of individual frames. This makes creating sophisticated animations very easy with only a few lines of code.

What Can Be Animated?

Both UIKit and Core Animation provide support for animations, but the level of support provided by each technology varies. In UIKit, animations are performed using `UIView` objects. Views support a basic set of animations that cover many common tasks. For example, you can animate changes to properties of views or use transition animations to replace one set of views with another.

Table 4-1 lists the *animatable* properties—the properties that have built-in animation support—of the `UIView` class. Being animatable does not mean animations happen automatically. Changing the value of these properties normally just updates the property (and the view) immediately without an animation. To animate such a change, you must change the property's value from inside an animation block, which is described in Animating Property Changes in a View.

Table 4-1 Animatable `UIView` properties

Property	Changes you can make
`frame`	Modify this property to change the view's size and position relative to its superview's coordinate system. (If the `transform` property does not contain the identity transform, modify the `bounds` or `center` properties instead.)
`bounds`	Modify this property to change the view's size.
`center`	Modify this property to change the view's position relative to its superview's coordinate system.
`transform`	Modify this property to scale, rotate, or translate the view relative to its center point. Transformations using this property are always performed in 2D space. (To perform 3D transformations, you must animate the view's layer object using Core Animation.)
`alpha`	Modify this property to gradually change the transparency of the view.
`backgroundColor`	Modify this property to change the view's background color.
`contentStretch`	Modify this property to change the way the view's contents are stretched to fill the available space.

Animated view transitions are a way for you to make changes to your view hierarchy beyond those offered by view controllers. Although you should use view controllers to manage succinct view hierarchies, there may be times when you want to replace all or part of a view hierarchy. In those situations, you can use view-based transitions to animate the addition and removal of your views.

In places where you want to perform more sophisticated animations, or animations not supported by the `UIView` class, you can use Core Animation and the view's underlying layer to create the animation. Because view and layer objects are intricately linked together, changes to a view's layer affect the view itself. Using Core Animation, you can animate the following types of changes for your view's layer:

- The size and position of the layer
- The center point used when performing transformations
- Transformations to the layer or its sublayers in 3D space
- The addition or removal of a layer from the layer hierarchy
- The layer's Z-order relative to other sibling layers
- The layer's shadow
- The layer's border (including whether the layer's corners are rounded)
- The portion of the layer that stretches during resizing operations
- The layer's opacity
- The clipping behavior for sublayers that lie outside the layer's bounds
- The current contents of the layer
- The rasterization behavior of the layer

Note: If your view hosts custom layer objects—that is, layer objects without an associated view—you must use Core Animation to animate any changes to them.

Although this chapter addresses a few Core Animation behaviors, it does so in relation to initiating them from your view code. For more complete information about how to use Core Animation to animate layers, see *Core Animation Programming Guide* and *Core Animation Cookbook*.

Animating Property Changes in a View

In order to animate changes to a property of the `UIView` class, you must wrap those changes inside an animation block. The term *animation block* is used in the generic sense to refer to any code that designates animatable changes. In iOS 4 and later, you create an animation block using block objects. In earlier versions of iOS, you mark the beginning and end of an animation block using special class methods of the `UIView` class. Both techniques support the same configuration options and offer the same amount of control over the animation execution. However, the block-based methods are preferred whenever possible.

The following sections focus on the code you need in order to animate changes to view properties. For information about how to create animated transitions between sets of views, see Creating Animated Transitions Between Views.

Starting Animations Using the Block-Based Methods

In iOS 4 and later, you use the block-based class methods to initiate animations. There are several block-based methods that offer different levels of configuration for the animation block. These methods are:

- `animateWithDuration:animations:`
- `animateWithDuration:animations:completion:`
- `animateWithDuration:delay:options:animations:completion:`

Because these are class methods, the animation blocks you create with them are not tied to a single view. Thus, you can use these methods to create a single animation that involves changes to multiple views. For example, Listing 4-1 shows the code needed to fade in one view while fading out another

over a one second time period. When this code executes, the specified animations are started immediately on another thread so as to avoid blocking the current thread or your application's main thread.

Listing 4-1 Performing a simple block-based animation

```
[UIView animateWithDuration:1.0 animations:^{
        firstView.alpha = 0.0;
        secondView.alpha = 1.0;
}];
```

The animations in the preceding example run only once using an ease-in, ease-out animation curve. If you want to change the default animation parameters, you must use the `animateWithDuration:delay:options:animations:completion:` method to perform your animations. This method lets you customize the following animation parameters:

- The delay to use before starting the animation
- The type of timing curve to use during the animation
- The number of times the animation should repeat
- Whether the animation should reverse itself automatically when it reaches the end
- Whether touch events are delivered to views while the animations are in progress
- Whether the animation should interrupt any in-progress animations or wait until those are complete before starting

Another thing that both the `animateWithDuration:animations:completion:` and `animateWithDuration:delay:options:animations:completion:` methods support is the ability to specify a completion handler block. You might use a completion handler to signal your application that a specific animation has finished. Completion handlers are also the way to link separate animations together.

Listing 4-2 shows an example of an animation block that uses a completion handler to initiate a new animation after the first one finishes. The first call to `animateWithDuration:delay:options:animations:completion:` sets up a fade-out animation and configures it with some custom options. When that animation is complete, its completion handler runs and sets up the second half of the animation, which fades the view back in after a delay.

Using a completion handler is the primary way that you link multiple animations.

Listing 4-2 Creating an animation block with custom options

```
- (IBAction)showHideView:(id)sender
{
    // Fade out the view right away
    [UIView animateWithDuration:1.0
        delay: 0.0
        options: UIViewAnimationOptionCurveEaseIn
        animations:^{
             thirdView.alpha = 0.0;
        }
        completion:^(BOOL finished){
            // Wait one second and then fade in the view
            [UIView animateWithDuration:1.0
                 delay: 1.0
                 options:UIViewAnimationOptionCurveEaseOut
                 animations:^{
```

```
                thirdView.alpha = 1.0;
            }
            completion:nil];
    }];
}
```

Important: Changing the value of a property while an animation involving that property is already in progress does not stop the current animation. Instead, the current animation continues and animates to the new value you just assigned to the property.

Starting Animations Using the Begin/Commit Methods

If your application runs in iOS 3.2 and earlier, you must use the `beginAnimations:context:` and `commitAnimations` class methods of `UIView` to define your animation blocks. These methods mark the beginning and end of your animation block. Any animatable properties you change between these methods are animated to their new values after you call the `commitAnimations` method. Execution of the animations occurs on a secondary thread so as to avoid blocking the current thread or your application's main thread.

Note: If you are writing an application for iOS 4 or later, you should use the block-based methods for animating your content instead. For information on how to use those methods, see Starting Animations Using the Block-Based Methods.

Listing 4-3 shows the code needed to implement the same behavior as Listing 4-1 but using the begin/commit methods. As in Listing 4-1, this code fades one view out while fading another in over one second of time. However, in this example, you must set the duration of the animation using a separate method call.

Listing 4-3 Performing a simple begin/commit animation

```
[UIView beginAnimations:@"ToggleViews" context:nil];
[UIView setAnimationDuration:1.0];

// Make the animatable changes.
firstView.alpha = 0.0;
secondView.alpha = 1.0;

// Commit the changes and perform the animation.
[UIView commitAnimations];
```

By default, all animatable property changes within an animation block are animated. If you want to animate some changes but not others, use the `setAnimationsEnabled:` method to disable animations temporarily, make any changes that you do not want animated, and then call `setAnimationsEnabled:` again to reenable animations. You can determine if animations are current enabled by calling the `areAnimationsEnabled` class method.

Note: Changing the value of a property while an animation involving that property is in progress does not stop the current animation. Instead, the animation continues and animates to the new value you just assigned to the property.

Configuring the Parameters for Begin/Commit Animations

To configure the animation parameters for a begin/commit animation block, you use any of several `UIView` class methods. Table 4-2 lists these methods and describes how you use them to configure your animations. Most of these methods should be called only from inside a begin/commit animation block but some may also be used with block-based animations. If you do not call one of these methods from your animation block, a default value for the corresponding attribute is used. For more information about the default value associated with each method, see the method description in *UIView Class Reference*.

Table 4-2 Methods for configuring animation blocks

Method	Usage
`setAnimationStartDate:` `setAnimationDelay:`	Use either of these methods to specify when the executions should begin executing. If the specified start date is in the past (or the delay is 0), the animations begin as soon as possible.
`setAnimationDuration:`	Use this method to set the period of time over which to execute the animations.
`setAnimationCurve:`	Use this method to set the timing curve of the animations. This controls whether animations execute linearly or change speed at certain times.
`setAnimationRepeatCount:` `setAnimationRepeatAutoreverses:`	Use these methods to set the number of times the animation repeats and whether the animation runs in reverse at the end of each complete cycle. For more information about using these methods, see Implementing Animations That Reverse Themselves.

`setAnimationDelegate:`
`setAnimationWillStartSelector:`
`setAnimationDidStopSelector:`
	Use these methods to execute code immediately before or after the animations. For more information about using a delegate, see Configuring an Animation Delegate.

`setAnimationBeginsFromCurrentState:`	Use this method to stop all previous animations immediately and start the new animations from the stopping point. If you pass `NO` to this method, instead of `YES`, the new animations do not begin executing until the previous animations stop.

Listing 4-4 shows the code needed to implement the same behavior as the code in Listing 4-2 but using the begin/commit methods. As before, this code fades out a view, waits one second, and then fades it back in. In order to implement the second part of the animation, the code sets up an animation delegate and implements a did-stop handler method. That handler method then sets up the second half of the animations and runs them.

Listing 4-4 Configuring animation parameters using the begin/commit methods

```
// This method begins the first animation.
- (IBAction)showHideView:(id)sender
{
    [UIView beginAnimations:@"ShowHideView" context:nil];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseIn];
    [UIView setAnimationDuration:1.0];
```

```
    [UIView setAnimationDelegate:self];
    [UIView
setAnimationDidStopSelector:@selector(showHideDidStop:finished:context:)];

    // Make the animatable changes.
    thirdView.alpha = 0.0;

    // Commit the changes and perform the animation.
    [UIView commitAnimations];
}

// Called at the end of the preceding animation.
- (void)showHideDidStop:(NSString *)animationID finished:(NSNumber *)finished
context:(void *)context
{
    [UIView beginAnimations:@"ShowHideView2" context:nil];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseOut];
    [UIView setAnimationDuration:1.0];
    [UIView setAnimationDelay:1.0];

    thirdView.alpha = 1.0;

    [UIView commitAnimations];
}
```

Configuring an Animation Delegate

If you want to execute code immediately before or after an animation, you must associate a delegate object and a start or stop selector with your begin/commit animation block. You set your delegate object using the `setAnimationDelegate:` class method of `UIView` and you set your start and stop selectors using the `setAnimationWillStartSelector:` and `setAnimationDidStopSelector:` class methods. During the animation, the animation system calls your delegate methods at the appropriate times to give you a chance to perform your code.

The signatures of your animation delegate methods need to be similar to the following:

```
- (void)animationWillStart:(NSString *)animationID context:(void *)context;
- (void)animationDidStop:(NSString *)animationID finished:(NSNumber *)finished
context:(void *)context;
```

The *animationID* and *context* parameters for both methods are the same parameters that you passed to the `beginAnimations:context:` method at the beginning of the animation block:

- *animationID*—An application-supplied string used to identify the animation.
- *context*—An application-supplied object that you can use to pass additional information to the delegate.

The `setAnimationDidStopSelector:` selector method has an additional parameter—a Boolean value that is `YES` if the animation ran to completion. If the value of this parameter is `NO`, the animation was either canceled or stopped prematurely by another animation.

Note: Although animation delegates can be used in the block-based methods, there is generally no need to use them there. Instead, place any code you want to run before the animations at the

beginning of your block and place any code you want to run after the animations finish in a completion handler.

Nesting Animation Blocks

You can assign different timing and configuration options to parts of an animation block by nesting additional animation blocks. As the name implies, a nested animation block is a new animation block created inside an existing animation block. Nested animations are started at the same time as any parent animations but run (for the most part) with their own configuration options. By default, nested animations do inherit the parent's duration and animation curve but even those options can be overridden as needed.

Listing 4-5 shows an example of how a nested animation is used to change the timing, duration, and behavior of some animations in the overall group. In this case, two views are being faded to total transparency, but the transparency of the `anotherView` object is changed back and forth several times before it is finally hidden. The `UIViewAnimationOptionOverrideInheritedCurve` and `UIViewAnimationOptionOverrideInheritedDuration` keys used in the nested animation block allow the curve and duration values from the first animation to be modified for the second animation. If these keys were not present, the duration and curve of the outer animation block would be used instead.

Listing 4-5 Nesting animations that have different configurations

```
[UIView animateWithDuration:1.0
        delay: 1.0
        options:UIViewAnimationOptionCurveEaseOut
        animations:^{
            aView.alpha = 0.0;

            // Create a nested animation that has a different
            // duration, timing curve, and configuration.
            [UIView animateWithDuration:0.2
                 delay:0.0
                 options: UIViewAnimationOptionOverrideInheritedCurve |
                          UIViewAnimationOptionCurveLinear |
                          UIViewAnimationOptionOverrideInheritedDuration |
                          UIViewAnimationOptionRepeat |
                          UIViewAnimationOptionAutoreverse
                 animations:^{
                      [UIView setAnimationRepeatCount:2.5];
                      anotherView.alpha = 0.0;
                 }
                 completion:nil];

        }
        completion:nil];
```

If you are using the begin/commit methods to create your animations, nesting works in much the same way as with the block-based methods. Each successive call to `beginAnimations:context:` within an already open animation block creates a new nested animation block that you can configure as needed. Any configuration changes you make apply to the most recently opened animation block. All animation blocks must be closed with a call to `commitAnimations` before the animations are submitted and executed.

Implementing Animations That Reverse Themselves

When creating reversible animations in conjunction with a repeat count, consider specifying a non integer value for the repeat count. For an autoreversing animation, each complete cycle of the animation involves animating from the original value to the new value and back again. If you want your animation to end on the new value, adding `0.5` to the repeat count causes the animation to complete the extra half cycle needed to end at the new value. If you do not include this half step, your animation will animate to the original value and then snap quickly to the new value, which may not be the visual effect you want.

Creating Animated Transitions Between Views

View transitions help you hide sudden changes associated with adding, removing, hiding, or showing views in your view hierarchy. You use view transitions to implement the following types of changes:

- **Change the visible subviews of an existing view.** You typically choose this option when you want to make relatively small changes to an existing view.
- **Replace one view in your view hierarchy with a different view.** You typically choose this option when you want to replace a view hierarchy that spans all or most of the screen.

> **Important:** View transitions should not be confused with transitions initiated by view controllers, such as the presentation of modal view controllers or the pushing of new view controllers onto a navigation stack. View transitions affect the view hierarchy only, whereas view-controller transitions change the active view controller as well. Thus, for view transitions, the view controller that was active when you initiated the transition remains active when the transition finishes.
>
> For more information about how you can use view controllers to present new content, see *View Controller Programming Guide for iOS*.

Changing the Subviews of a View

Changing the subviews of a view allows you to make moderate changes to the view. For example, you might add or remove subviews to toggle the superview between two different states. By the time the animations finish, the same view is displayed but its contents are now different.

In iOS 4 and later, you use the `transitionWithView:duration:options:animations:completion:` method to initiate a transition animation for a view. In the animations block passed to this method, the only changes that are normally animated are those associated with showing, hiding, adding, or removing subviews. Limiting animations to this set allows the view to create a snapshot image of the before and after versions of the view and animate between the two images, which is more efficient. However, if you need to animate other changes, you can include the `UIViewAnimationOptionAllowAnimatedContent` option when calling the method. Including that option prevents the view from creating snapshots and animates all changes directly.

Listing 4-6 is an example of how to use a transition animation to make it seem as if a new text entry page has been added. In this example, the main view contains two embedded text views. The text views are configured identically, but one is always visible while the other is always hidden. When the user taps the button to create a new page, this method toggles the visibility of the two views, resulting in a new empty page with an empty text view ready to accept text. After the transition is complete, the view saves the text from the old page using a private method and resets the now hidden text view so that it can be reused later. The view then arranges its pointers so that it can be ready to do the same thing if the user requests yet another new page.

Listing 4-6 Swapping an empty text view for an existing one

```
- (IBAction)displayNewPage:(id)sender
{
```

```
    [UIView transitionWithView:self.view
        duration:1.0
        options:UIViewAnimationOptionTransitionCurlUp
        animations:^{
            currentTextView.hidden = YES;
            swapTextView.hidden = NO;
        }
        completion:^(BOOL finished){
            // Save the old text and then swap the views.
            [self saveNotes:temp];

            UIView*    temp = currentTextView;
            currentTextView = swapTextView;
            swapTextView = temp;
        }];
}
```

If you need to perform view transitions in iOS 3.2 and earlier, you can use the `setAnimationTransition:forView:cache:` method to specify the parameters for the transition. The view you pass to that method is the same one you would pass in as the first parameter to the `transitionWithView:duration:options:animations:completion:` method. Listing 4-7 shows the basic structure of the animation block you need to create. Note that to implement the completion block shown in Listing 4-6, you would need to configure an animation delegate with a did-stop handler as described in Configuring an Animation Delegate.

Listing 4-7 Changing subviews using the begin/commit methods

```
    [UIView beginAnimations:@"ToggleSiblings" context:nil];
    [UIView setAnimationTransition:UIViewAnimationTransitionCurlUp forView:self.view
cache:YES];
    [UIView setAnimationDuration:1.0];

    // Make your changes

    [UIView commitAnimations];
```

Replacing a View with a Different View

Replacing views is something you do when you want your interface to be dramatically different. Because this technique swaps only views (and not view controllers), you are responsible for designing your application's controller objects appropriately. This technique is simply a way of presenting new views quickly using some standard transitions.

In iOS 4 and later, you use the `transitionFromView:toView:duration:options:completion:` method to transition between two views. This method actually removes the first view from your hierarchy and inserts the other, so you should make sure you have a reference to the first view if you want to keep it. If you want to hide views instead of remove them from your view hierarchy, pass the `UIViewAnimationOptionShowHideTransitionViews` key as one of the options.

Listing 4-8 shows the code needed to swap between two main views managed by a single view controller. In this example, the view controller's root view always displays one of two child views (`primaryView` or `secondaryView`). Each view presents the same content but does so in a different way. The view controller uses the `displayingPrimary` member variable (a Boolean value) to keep track of which view is displayed at any given time. The flip direction changes depending on which view is being displayed.

Listing 4-8 Toggling between two views in a view controller

```
- (IBAction)toggleMainViews:(id)sender {
    [UIView transitionFromView:(displayingPrimary ? primaryView : secondaryView)
        toView:(displayingPrimary ? secondaryView : primaryView)
        duration:1.0
        options:(displayingPrimary ? UIViewAnimationOptionTransitionFlipFromRight :
                    UIViewAnimationOptionTransitionFlipFromLeft)
        completion:^(BOOL finished) {
            if (finished) {
                displayingPrimary = !displayingPrimary;
            }
    }];
}
```

Note: In addition to swapping out views, your view controller code needs to manage the loading and unloading of both the primary and secondary views. For information on how views are loaded and unloaded by a view controller, see *View Controller Programming Guide for iOS*.

Linking Multiple Animations Together

The `UIView` animation interfaces provide support for linking separate animation blocks so that they perform sequentially instead of at the same time. The process for linking animation blocks depends on whether you are using the block-based animation methods or the begin/commit methods:

- For block-based animations, use the completion handler supported by the `animateWithDuration:animations:completion:` and `animateWithDuration:delay:options:animations:completion:` methods to execute any follow-on animations.
- For begin/commit animations, associate a delegate object and a did-stop selector with the animation. For information about how to associate a delegate with your animations, see Configuring an Animation Delegate.

An alternative to linking animations together is to use nested animations with different delay factors so as to start the animations at different times. For more information on how to nest animations, see Nesting Animation Blocks.

Animating View and Layer Changes Together

Applications can freely mix view-based and layer-based animation code as needed but the process for configuring your animation parameters depends on who owns the layer. Changing a view-owned layer is the same as changing the view itself, and any animations you apply to the layer's properties respect the animation parameters of the current view-based animation block. The same is not true for layers that you create yourself. Custom layer objects ignore view-based animation block parameters and use the default Core Animation parameters instead.

If you want to customize the animation parameters for layers you create, you must use Core Animation directly. Typically, animating layers using Core Animation involves creating a `CABasicAnimation` object or some other concrete subclass of `CAAnimation`. You then add that

animation to the corresponding layer. You can apply the animation from either inside or outside a view-based animation block.

Listing 4-9 shows an animation that modifies a view and a custom layer at the same time. The view in this example contains a custom `CALayer` object at the center of its bounds. The animation rotates the view counter clockwise while rotating the layer clockwise. Because the rotations are in opposite directions, the layer maintains its original orientation relative to the screen and does not appear to rotate significantly. However, the view beneath that layer spins 360 degrees and returns to its original orientation. This example is presented primarily to demonstrate how you can mix view and layer animations. This type of mixing should not be used in situations where precise timing is needed.

Listing 4-9 Mixing view and layer animations

```
[UIView animateWithDuration:1.0
    delay:0.0
    options: UIViewAnimationOptionCurveLinear
    animations:^{
        // Animate the first half of the view rotation.
        CGAffineTransform  xform =
CGAffineTransformMakeRotation(DEGREES_TO_RADIANS(-180));
        backingView.transform = xform;

        // Rotate the embedded CALayer in the opposite direction.
        CABasicAnimation*    layerAnimation = [CABasicAnimation
animationWithKeyPath:@"transform"];
        layerAnimation.duration = 2.0;
        layerAnimation.beginTime = 0; //CACurrentMediaTime() + 1;
        layerAnimation.valueFunction = [CAValueFunction
functionWithName:kCAValueFunctionRotateZ];
        layerAnimation.timingFunction = [CAMediaTimingFunction
                        functionWithName:kCAMediaTimingFunctionLinear];
        layerAnimation.fromValue = [NSNumber numberWithFloat:0.0];
        layerAnimation.toValue = [NSNumber
numberWithFloat:DEGREES_TO_RADIANS(360.0)];
        layerAnimation.byValue = [NSNumber
numberWithFloat:DEGREES_TO_RADIANS(180.0)];
        [manLayer addAnimation:layerAnimation forKey:@"layerAnimation"];
    }
    completion:^(BOOL finished){
        // Now do the second half of the view rotation.
        [UIView animateWithDuration:1.0
             delay: 0.0
             options: UIViewAnimationOptionCurveLinear
             animations:^{
                 CGAffineTransform  xform =
CGAffineTransformMakeRotation(DEGREES_TO_RADIANS(-359));
                 backingView.transform = xform;
             }
             completion:^(BOOL finished){
                 backingView.transform = CGAffineTransformIdentity;
         }];
}];
```

Note: In Listing 4-9, you could also create and apply the `CABasicAnimation` object outside of the view-based animation block to achieve the same results. All of the animations ultimately rely on Core Animation for their execution. Thus, if they are submitted at approximately the same time, they run together.

If precise timing between your view and layer based animations is required, it is recommended that you create all of the animations using Core Animation. You may find that some animations are easier to perform using Core Animation anyway. For example, the view-based rotation in Listing 4-9 requires a multistep sequence for rotations of more than 180 degrees, whereas the Core Animation portion uses a rotation value function that rotates from start to finish through a middle value.

For more information about how to create and configure animations using Core Animation, see *Core Animation Programming Guide* and *Core Animation Cookbook*.