View Programming Guide for iOS

Developer

Contents

About Windows and Views 7
At a Glance 7
Views Manage Your Application's Visual Content 7
Windows Coordinate the Display of Your Views 8
Animations Provide the User with Visible Feedback for Interface Changes 8
The Role of Interface Builder 8
See Also 9

View and Window Architecture 10
View Architecture Fundamentals 10
View Hierarchies and Subview Management 11
The View Drawing Cycle 12
Content Modes 13
Stretchable Views 15
Built-In Animation Support 16
View Geometry and Coordinate Systems 17
The Relationship of the Frame, Bounds, and Center Properties 18
Coordinate System Transformations 20
Points Versus Pixels 21
The Runtime Interaction Model for Views 23
Tips for Using Views Effectively 25
Views Do Not Always Have a Corresponding View Controller 25
Minimize Custom Drawing 26
Take Advantage of Content Modes 26
Declare Views as Opaque Whenever Possible 26
Adjust Your View's Drawing Behavior When Scrolling 26
Do Not Customize Controls by Embedding Subviews 27

Windows 28
Tasks That Involve Windows 28
Creating and Configuring a Window 29
Creating Windows in Interface Builder 29
Creating a Window Programmatically 30
Adding Content to Your Window 30

Changing the Window Level 31
Monitoring Window Changes 31
Displaying Content on an External Display 32
Handling Screen Connection and Disconnection Notifications 33
Configuring a Window for an External Display 35
Configuring the Screen Mode of an External Display 37

Views 38
Creating and Configuring View Objects 38
Creating View Objects Using Interface Builder 39
Creating View Objects Programmatically 39
Setting the Properties of a View 40
Tagging Views for Future Identification 42
Creating and Managing a View Hierarchy 42
Adding and Removing Subviews 43
Hiding Views 46
Locating Views in a View Hierarchy 47
Translating, Scaling, and Rotating Views 47
Converting Coordinates in the View Hierarchy 50
Adjusting the Size and Position of Views at Runtime 51
Being Prepared for Layout Changes 51
Handling Layout Changes Automatically Using Autoresizing Rules 52
Tweaking the Layout of Your Views Manually 54
Modifying Views at Runtime 54
Interacting with Core Animation Layers 56
Changing the Layer Class Associated with a View 56
Embedding Layer Objects in a View 57
Defining a Custom View 58
Checklist for Implementing a Custom View 58
Initializing Your Custom View 59
Implementing Your Drawing Code 60
Responding to Events 61
Cleaning Up After Your View 63

Animations 64
What Can Be Animated? 64
Animating Property Changes in a View 66
Starting Animations Using the Block-Based Methods 66
Starting Animations Using the Begin/Commit Methods 68
Nesting Animation Blocks 72

Implementing Animations That Reverse Themselves 73
Creating Animated Transitions Between Views 74
Changing the Subviews of a View 74
Replacing a View with a Different View 76
Linking Multiple Animations Together 77
Animating View and Layer Changes Together 77

Document Revision History 80

Figures, Tables, and Listings

View and Window Architecture 10

Figure 1-1 Architecture of the views in a sample application 11
Figure 1-2 Content mode comparisons 14
Figure 1-3 Stretching the background of a button 15
Figure 1-4 Coordinate system orientation in UIKit 17
Figure 1-5 Relationship between a view's frame and bounds 19
Figure 1-6 Rotating a view and its content 21
Figure 1-7 UIKit interactions with your view objects 23
Table 1-1 Screen dimensions for iOS-based devices 22

Windows 28

Listing 2-1 Registering for screen connect and disconnect notifications 33
Listing 2-2 Handling connect and disconnect notifications 34
Listing 2-3 Configuring a window for an external display 35

Views 38

Figure 3-1 Layered views in the Clock application 43
Figure 3-2 Rotating a view 45 degrees 49
Figure 3-3 Converting values in a rotated view 51
Figure 3-4 View autoresizing mask constants 53
Table 3-1 Usage of some key view properties 40
Table 3-2 Autoresizing mask constants 52
Listing 3-1 Adding a view to a window 44
Listing 3-2 Adding views to an existing view hierarchy 45
Listing 3-3 Adding a custom layer to a view 57
Listing 3-4 Initializing a view subclass 59
Listing 3-5 A drawing method 61
Listing 3-6 Implementing the `dealloc` method 63

Animations 64

Table 4-1 Animatable `UIView` properties 64
Table 4-2 Methods for configuring animation blocks 69
Listing 4-1 Performing a simple block-based animation 66
Listing 4-2 Creating an animation block with custom options 67

Listing 4-3 Performing a simple begin/commit animation 69
Listing 4-4 Configuring animation parameters using the begin/commit methods 70
Listing 4-5 Nesting animations that have different configurations 72
Listing 4-6 Swapping an empty text view for an existing one 75
Listing 4-7 Changing subviews using the begin/commit methods 75
Listing 4-8 Toggling between two views in a view controller 76
Listing 4-9 Mixing view and layer animations 78

About Windows and Views

In iOS, you use windows and views to present your application's content on the screen. Windows do not have any visible content themselves but provide a basic container for your application's views. Views define a portion of a window that you want to fill with some content. For example, you might have views that display images, text, shapes, or some combination thereof. You can also use views to organize and manage other views.

At a Glance

Every application has at least one window and one view for presenting its content. UIKit and other system frameworks provide predefined views that you can use to present your content. These views range from simple buttons and text labels to more complex views such as table views, picker views, and scroll views. In places where the predefined views do not provide what you need, you can also define custom views and manage the drawing and event handling yourself.

Views Manage Your Application's Visual Content

A view is an instance of the `UIView` class (or one of its subclasses) and manages a rectangular area in your application window. Views are responsible for drawing content, handling multitouch events, and managing the layout of any subviews. Drawing involves using graphics technologies such as Core Graphics, OpenGL ES, or UIKit to draw shapes, images, and text inside a view's rectangular area. A view responds to touch events in its rectangular area either by using gesture recognizers or by handling touch events directly. In the view hierarchy, parent views are responsible for positioning and sizing their child views and can do so dynamically. This ability to modify child views dynamically lets your views adjust to changing conditions, such as interface rotations and animations.

You can think of views as building blocks that you use to construct your user interface. Rather than use one view to present all of your content, you often use several views to build a view hierarchy. Each view in the hierarchy presents a particular portion of your user interface and is generally optimized for a specific type of content. For example, UIKit has views specifically for presenting images, text and other types of content.

Relevant Chapters: "View and Window Architecture" (page 10)

"Views" (page 38)

Windows Coordinate the Display of Your Views

A window is an instance of the `UIWindow` class and handles the overall presentation of your application's user interface. Windows work with views (and their owning view controllers) to manage interactions with, and changes to, the visible view hierarchy. For the most part, your application's window never changes. After the window is created, it stays the same and only the views displayed by it change. Every application has at least one window that displays the application's user interface on a device's main screen. If an external display is connected to the device, applications can create a second window to present content on that screen as well.

Relevant Chapters: "Windows" (page 28)

Animations Provide the User with Visible Feedback for Interface Changes

Animations provide users with visible feedback about changes to your view hierarchy. The system defines standard animations for presenting modal views and transitioning between different groups of views. However, many attributes of a view can also be animated directly. For example, through animation you can change the transparency of a view, its position on the screen, its size, its background color, or other attributes. And if you work directly with the view's underlying Core Animation layer object, you can perform many other animations as well.

Relevant Chapters: "Animations" (page 64)

The Role of Interface Builder

Interface Builder is an application that you use to graphically construct and configure your application's windows and views. Using Interface Builder, you assemble your views and place them in a nib file, which is a resource file that stores a freeze-dried version of your views and other objects. When you load a nib file at runtime, the objects inside it are reconstituted into actual objects that your code can then manipulate programmatically.

Interface Builder greatly simplifies the work you have to do in creating your application's user interface. Because support for Interface Builder and nib files is incorporated throughout iOS, little effort is required to incorporate nib files into your application's design.

For more information about how to use Interface Builder, see *Interface Builder User Guide*. For information about how view controllers manage the nib files containing their views, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

See Also

Because views are very sophisticated and flexible objects, it would be impossible to cover all of their behaviors in one document. However, other documents are available to help you learn about other aspects of managing views and your user interface as a whole.

- View controllers are an important part of managing your application's views. A view controller presides over all of the views in a single view hierarchy and facilitates the presentation of those views on the screen. For more information about view controllers and the role they play, see *View Controller Programming Guide for iOS*.
- Views are the key recipients of gesture and touch events in your application. For more information about using gesture recognizers and handling touch events directly, see *Event Handling Guide for iOS*.
- Custom views must use the available drawing technologies to render their content. For information about using these technologies to draw within your views, see *Drawing and Printing Guide for iOS*.
- In places where the standard view animations are not sufficient, you can use Core Animation. For information about implementing animations using Core Animation, see *Core Animation Programming Guide*.

View and Window Architecture

Views and windows present your application's user interface and handle the interactions with that interface. UIKit and other system frameworks provide a number of views that you can use as-is with little or no modification. You can also define custom views for places where you need to present content differently than the standard views allow.

Whether you use the system views or create your own custom views, you need to understand the infrastructure provided by the `UIView` and `UIWindow` classes. These classes provide sophisticated facilities for managing the layout and presentation of views. Understanding how those facilities work is important for making sure your views behave appropriately when changes occur in your application.

View Architecture Fundamentals

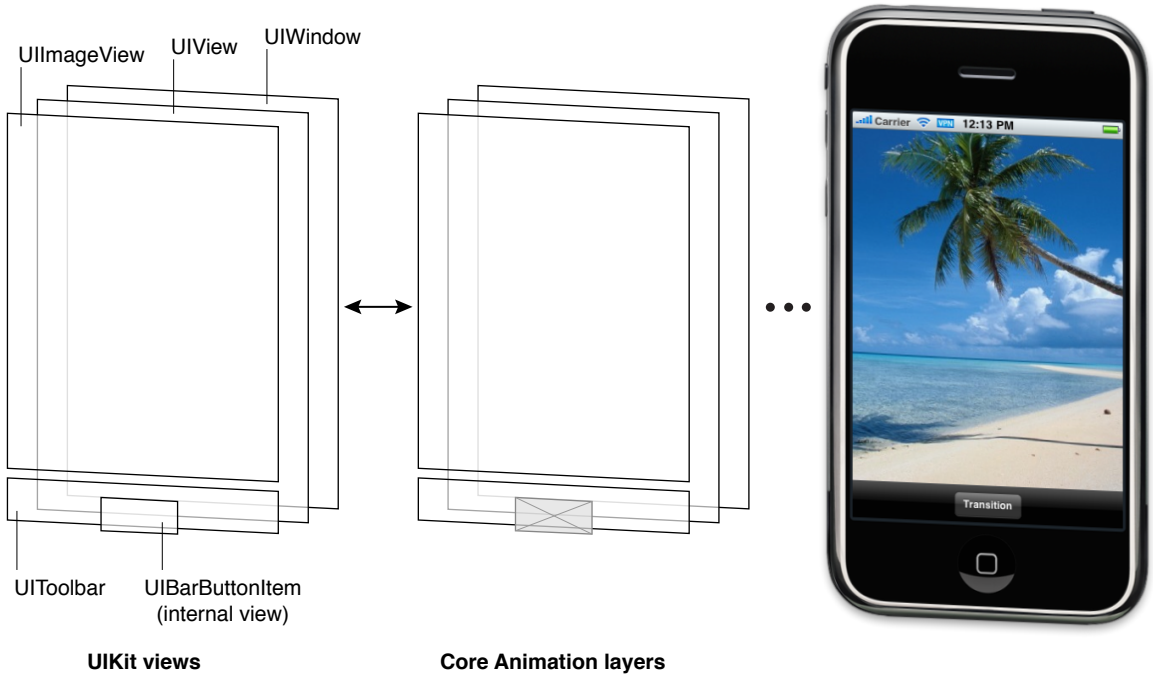
Most of the things you might want to do visually are done with view objects—instances of the `UIView` class. A view object defines a rectangular region on the screen and handles the drawing and touch events in that region. A view can also act as a parent for other views and coordinate the placement and sizing of those views. The `UIView` class does most of the work in managing these relationships between views, but you can also customize the default behavior as needed.

Views work in conjunction with Core Animation layers to handle the rendering and animating of a view's content. Every view in UIKit is backed by a layer object (usually an instance of the `CALayer` class), which manages the backing store for the view and handles view-related animations. Most operations you perform should be through the `UIView` interface. However, in situations where you need more control over the rendering or animation behavior of your view, you can perform operations through its layer instead.

To understand the relationship between views and layers, it helps to look at an example. Figure 1-1 shows the view architecture from the *ViewTransitions* sample application along with the relationship to the underlying Core Animation layers. The views in the application include a window (which is also a view), a generic `UIView` object that acts as a container view, an image view, a toolbar for displaying controls, and a bar button item (which is not a view itself but which manages a view internally). (The actual *ViewTransitions* sample application includes an additional image view that is used to implement transitions. For simplicity, and because that view is usually hidden, it is not included in Figure 1-1.) Every view has a corresponding layer object that can be

accessed from that view's `layer` property. (Because a bar button item is not a view, you cannot access its layer directly.) Behind those layer objects are Core Animation rendering objects and ultimately the hardware buffers used to manage the actual bits on the screen.

Figure 1-1 Architecture of the views in a sample application



The use of Core Animation layer objects has important implications for performance. The actual drawing code of a view object is called as little as possible, and when the code is called, the results are cached by Core Animation and reused as much as possible later. Reusing already-rendered content eliminates the expensive drawing cycle usually needed to update views. Reuse of this content is especially important during animations, where the existing content can be manipulated. Such reuse is much less expensive than creating new content.

View Hierarchies and Subview Management

In addition to providing its own content, a view can act as a container for other views. When one view contains another, a parent-child relationship is created between the two views. The child view in the relationship is known as the **subview** and the parent view is known as the **superview**. The creation of this type of relationship has implications for both the visual appearance of your application and the application's behavior.

Visually, the content of a subview obscures all or part of the content of its parent view. If the subview is totally opaque, then the area occupied by the subview completely obscures the corresponding area of the parent. If the subview is partially transparent, the content from the two views is blended together prior to being displayed

on the screen. Each superview stores its subviews in an ordered array and the order in that array also affects the visibility of each subview. If two sibling subviews overlap each other, the one that was added last (or was moved to the end of the subview array) appears on top of the other.

The superview-subview relationship also impacts several view behaviors. Changing the size of a parent view has a ripple effect that can cause the size and position of any subviews to change too. When you change the size of a parent view, you can control the resizing behavior of each subview by configuring the view appropriately. Other changes that affect subviews include hiding a superview, changing a superview's alpha (transparency), or applying a mathematical transform to a superview's coordinate system.

The arrangement of views in a view hierarchy also determines how your application responds to events. When a touch occurs inside a specific view, the system sends an event object with the touch information directly to that view for handling. However, if the view does not handle a particular touch event, it can pass the event object along to its superview. If the superview does not handle the event, it passes the event object to its superview, and so on up the responder chain. Specific views can also pass the event object to an intervening responder object, such as a view controller. If no object handles the event, it eventually reaches the application object, which generally discards it.

For more information about how to create view hierarchies, see "Creating and Managing a View Hierarchy" (page 42).

The View Drawing Cycle

The `UIView` class uses an on-demand drawing model for presenting content. When a view first appears on the screen, the system asks it to draw its content. The system captures a snapshot of this content and uses that snapshot as the view's visual representation. If you never change the view's content, the view's drawing code may never be called again. The snapshot image is reused for most operations involving the view. If you do change the content, you notify the system that the view has changed. The view then repeats the process of drawing the view and capturing a snapshot of the new results.

When the contents of your view change, you do not redraw those changes directly. Instead, you invalidate the view using either the `setNeedsDisplay` or `setNeedsDisplayInRect:` method. These methods tell the system that the contents of the view changed and need to be redrawn at the next opportunity. The system waits until the end of the current run loop before initiating any drawing operations. This delay gives you a chance to invalidate multiple views, add or remove views from your hierarchy, hide views, resize views, and reposition views all at once. All of the changes you make are then reflected at the same time.

> **Note:** Changing a view's geometry does not automatically cause the system to redraw the view's content. The view's `contentMode` property determines how changes to the view's geometry are interpreted. Most content modes stretch or reposition the existing snapshot within the view's boundaries and do not create a new one. For more information about how content modes affect the drawing cycle of your view, see "Content Modes" (page 13).

When the time comes to render your view's content, the actual drawing process varies depending on the view and its configuration. System views typically implement private drawing methods to render their content. Those same system views often expose interfaces that you can use to configure the view's actual appearance. For custom `UIView` subclasses, you typically override the `drawRect:` method of your view and use that method to draw your view's content. There are also other ways to provide a view's content, such as setting the contents of the underlying layer directly, but overriding the `drawRect:` method is the most common technique.

For more information about how to draw content for custom views, see "Implementing Your Drawing Code" (page 60).

Content Modes

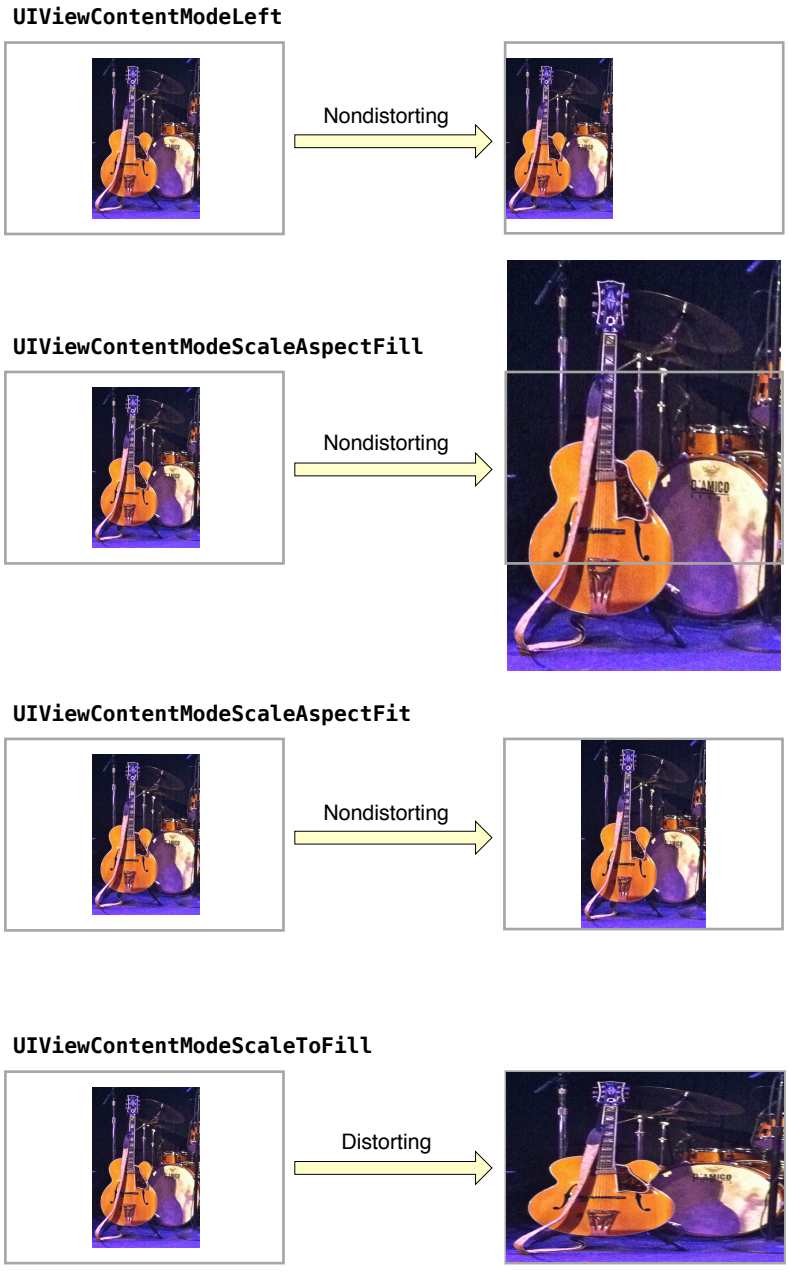
Each view has a content mode that controls how the view recycles its content in response to changes in the view's geometry and whether it recycles its content at all. When a view is first displayed, it renders its content as usual and the results are captured in an underlying bitmap. After that, changes to the view's geometry do not always cause the bitmap to be recreated. Instead, the value in the `contentMode` property determines whether the bitmap should be scaled to fit the new bounds or simply pinned to one corner or edge of the view.

The content mode of a view is applied whenever you do the following:

- Change the width or height of the view's `frame` or `bounds` rectangles.
- Assign a transform that includes a scaling factor to the view's `transform` property.

By default, the `contentMode` property for most views is set to `UIViewContentModeScaleToFill`, which causes the view's contents to be scaled to fit the new frame size. Figure 1-2 shows the results that occur for some content modes that are available. As you can see from the figure, not all content modes result in the view's bounds being filled entirely, and those that do might distort the view's content.

Figure 1-2 Content mode comparisons



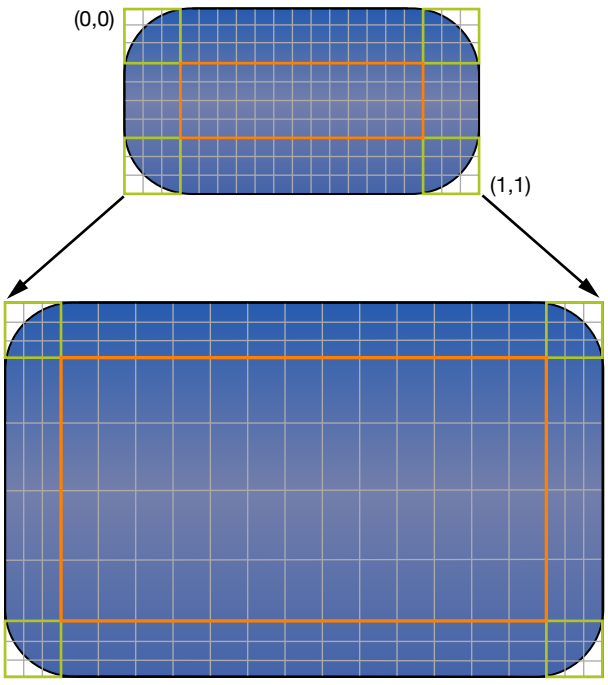
Content modes are good for recycling the contents of your view, but you can also set the content mode to the `UIViewContentModeRedraw` value when you specifically want your custom views to redraw themselves during scaling and resizing operations. Setting your view's content mode to this value forces the system to call your view's `drawRect:` method in response to geometry changes. In general, you should avoid using this value whenever possible, and you should certainly not use it with the standard system views.

For more information about the available content modes, see *UIView Class Reference*.

Stretchable Views

You can designate a portion of a view as stretchable so that when the size of the view changes only the content in the stretchable portion is affected. You typically use stretchable areas for buttons or other views where part of the view defines a repeatable pattern. The stretchable area you specify can allow for stretching along one or both axes of the view. Of course, when stretching a view along two axes, the edges of the view must also define a repeatable pattern to avoid any distortion. Figure 1-3 shows how this distortion manifests itself in a view. The color from each of the view's original pixels is replicated to fill the corresponding area in the larger view.

Figure 1-3 Stretching the background of a button



You specify the stretchable area of a view using the `contentStretch` property. This property accepts a rectangle whose values are normalized to the range `0.0` to `1.0`. When stretching the view, the system multiplies these normalized values by the view's current bounds and scale factor to determine which pixel or pixels need to be stretched. The use of normalized values alleviates the need for you to update the `contentStretch` property every time the bounds of your view change.

The view's content mode also plays a role in determining how the view's stretchable area is used. Stretchable areas are only used when the content mode would cause the view's content to be scaled. This means that stretchable views are supported only with the `UIViewContentModeScaleToFill`, `UIViewContentModeScaleAspectFit`, and `UIViewContentModeScaleAspectFill` content modes. If you specify a content mode that pins the content to an edge or corner (and thus does not actually scale the content), the view ignores the stretchable area.

> **Note:** The use of the `contentStretch` property is recommended over the creation of a stretchable `UIImage` object when specifying the background for a view. Stretchable views are handled entirely in the Core Animation layer, which typically offers better performance.

Built-In Animation Support

One of the benefits of having a layer object behind every view is that you can animate many view-related changes easily. Animations are a useful way to communicate information to the user and should always be considered during the design of your application. Many properties of the `UIView` class are **animatable**—that is, semiautomatic support exists for animating from one value to another. To perform an animation for one of these animatable properties, all you have to do is:

1. Tell UIKit that you want to perform an animation.
2. Change the value of the property.

Among the properties you can animate on a `UIView` object are the following:

`frame`—Use this to animate position and size changes for the view.

`bounds`—Use this to animate changes to the size of the view.

`center`—Use this to animate the position of the view.

`transform`—Use this to rotate or scale the view.

`alpha`—Use this to change the transparency of the view.

`backgroundColor`—Use this to change the background color of the view.

`contentStretch`—Use this to change how the view's contents stretch.

One place where animations are very important is when transitioning from one set of views to another. Typically, you use a view controller to manage the animations associated with major changes between parts of your user interface. For example, for interfaces that involve navigating from higher-level to lower-level information, you typically use a navigation controller to manage the transitions between the views displaying each successive level of data. However, you can also create transitions between two sets of views using animations instead of a view controller. You might do so in places where the standard view-controller animations do not yield the results you want.

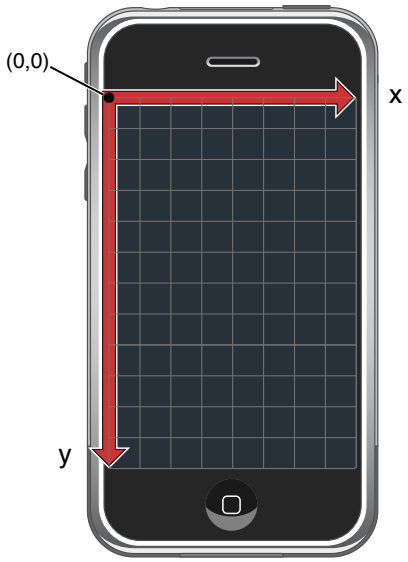
In addition to the animations you create using UIKit classes, you can also create animations using Core Animation layers. Dropping down to the layer level gives you much more control over the timing and properties of your animations.

For details about how to perform view-based animations, see "Animations" (page 64). For more information about creating animations using Core Animation, see *Core Animation Programming Guide* and *Core Animation Cookbook*.

View Geometry and Coordinate Systems

The default coordinate system in UIKit has its origin in the top-left corner and has axes that extend down and to the right from the origin point. Coordinate values are represented using floating-point numbers, which allow for precise layout and positioning of content regardless of the underlying screen resolution. Figure 1-4 shows this coordinate system relative to the screen. In addition to the screen coordinate system, windows and views define their own local coordinate systems that allow you to specify coordinates relative to the view or window origin instead of relative to the screen.

Figure 1-4 Coordinate system orientation in UIKit



Because every view and window defines its own local coordinate system, you need to be aware of which coordinate system is in effect at any given time. Every time you draw into a view or change its geometry, you do so relative to some coordinate system. In the case of drawing, you specify coordinates relative to the view's own coordinate system. In the case of geometry changes, you specify coordinates relative to the superview's coordinate system. The `UIWindow` and `UIView` classes both include methods to help you convert from one coordinate system to another.

Important: Some iOS technologies define default coordinate systems whose origin point and orientation differ from those used by UIKit. For example, Core Graphics and OpenGL ES use a coordinate system whose origin lies in the lower-left corner of the view or window and whose y-axis points upward relative to the screen. Your code must take such differences into account when drawing or creating content and adjust coordinate values (or the default orientation of the coordinate system) as needed.

The Relationship of the Frame, Bounds, and Center Properties

A view object tracks its size and location using its `frame`, `bounds`, and `center` properties:

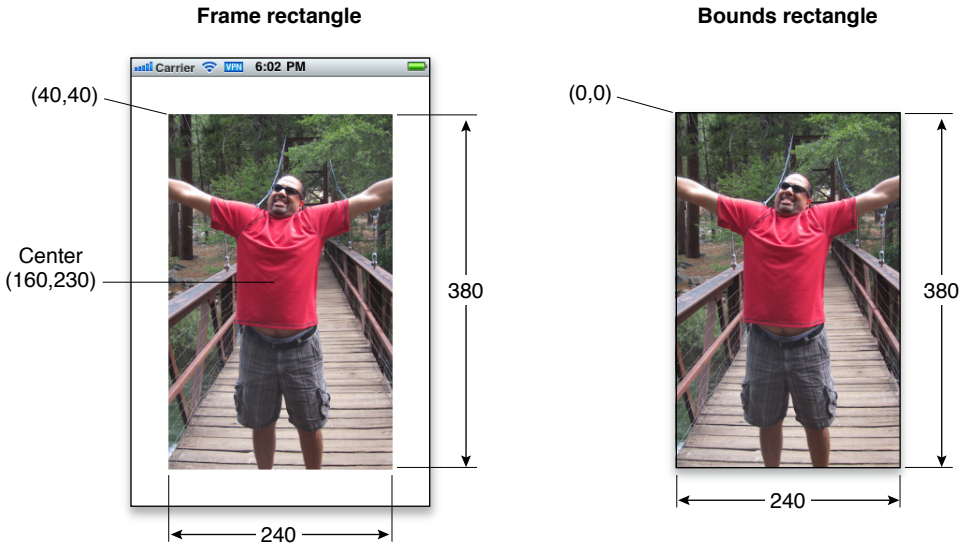
- The `frame` property contains the **frame rectangle**, which specifies the size and location of the view in its superview's coordinate system.
- The `bounds` property contains the **bounds rectangle**, which specifies the size of the view (and its content origin) in the view's own local coordinate system.
- The `center` property contains the known center point of the view in the superview's coordinate system.

You use the `center` and `frame` properties primarily for manipulating the geometry of the current view. For example, you use these properties when building your view hierarchy or changing the position or size of a view at runtime. If you are changing only the position of the view (and not its size), the `center` property is the preferred way to do so. The value in the `center` property is always valid, even if scaling or rotation factors have been added to the view's transform. The same is not true for the value in the `frame` property, which is considered invalid if the view's transform is not equal to the identity transform.

You use the `bounds` property primarily during drawing. The bounds rectangle is expressed in the view's own local coordinate system. The default origin of this rectangle is (0, 0) and its size matches the size of the frame rectangle. Anything you draw inside this rectangle is part of the view's visible content. If you change the origin of the bounds rectangle, anything you draw inside the new rectangle becomes part of the view's visible content.

Figure 1-5 shows the relationship between the frame and bounds rectangles for an image view. In the figure, the upper-left corner of the image view is located at the point (40, 40) in its superview's coordinate system and the size of the rectangle is 240 by 380 points. For the bounds rectangle, the origin point is (0, 0) and the size of the rectangle is similarly 240 by 380 points.

Figure 1-5 Relationship between a view's frame and bounds



Although you can change the `frame`, `bounds`, and `center` properties independent of the others, changes to one property affect the others in the following ways:

- When you set the `frame` property, the size value in the `bounds` property changes to match the new size of the frame rectangle. The value in the `center` property similarly changes to match the new center point of the frame rectangle.
- When you set the `center` property, the origin value in the `frame` changes accordingly.
- When you set the size of the `bounds` property, the size value in the `frame` property changes to match the new size of the bounds rectangle.

By default, a view's frame is not clipped to its superview's frame. Thus, any subviews that lie outside of their superview's frame are rendered in their entirety. You can change this behavior, though, by setting the superview's `clipsToBounds` property to `YES`. Regardless of whether or not subviews are clipped visually, touch events always respect the bounds rectangle of the target view's superview. In other words, touch events occurring in a part of a view that lies outside of its superview's bounds rectangle are not delivered to that view.

Coordinate System Transformations

Coordinate system transformations offer a way to alter your view (or its contents) quickly and easily. An **affine transform** is a mathematical matrix that specifies how points in one coordinate system map to points in a different coordinate system. You can apply affine transforms to your entire view to change the size, location, or orientation of the view relative to its superview. You can also use affine transforms in your drawing code to perform the same types of manipulations to individual pieces of rendered content. How you apply the affine transform therefore depends on context:

- To modify your entire view, modify the affine transform in the `transform` property of your view.
- To modify specific pieces of content in your view's `drawRect:` method, modify the affine transform associated with the active graphics context.

You typically modify the `transform` property of a view when you want to implement animations. For example, you could use this property to create an animation of your view rotating around its center point. You would not use this property to make permanent changes to your view, such as modifying its position or size a view within its superview's coordinate space. For that type of change, you should modify the frame rectangle of your view instead.

> **Note:** When modifying the `transform` property of your view, all transformations are performed relative to the center point of the view.

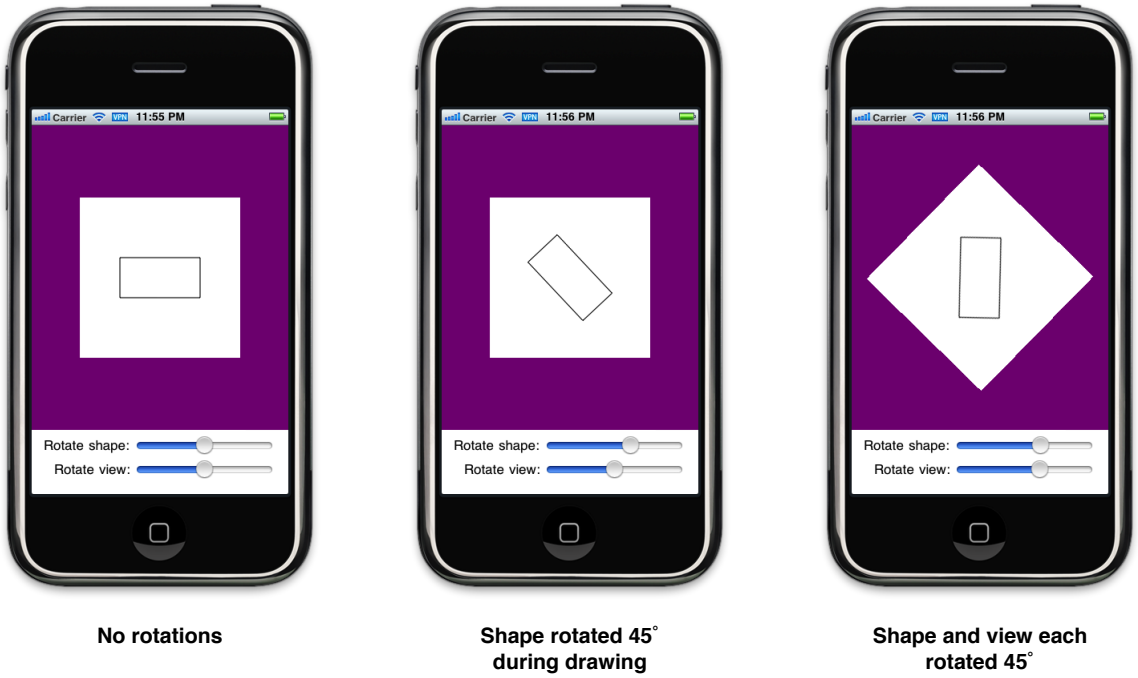
In your view's `drawRect:` method, you use affine transforms to position and orient the items you plan to draw. Rather than fix the position of an object at some location in your view, it is simpler to create each object relative to a fixed point, typically (0, 0), and use a transform to position the object immediately prior to drawing. That way, if the position of the object changes in your view, all you have to do is modify the transform, which is much faster and less expensive than recreating the object at its new location. You can retrieve the affine transform associated with a graphics context using the `CGContextGetCTM` function and you can use the related Core Graphics functions to set or modify this transform during drawing.

The **current transformation matrix (CTM)** is the affine transform in use at any given time. When manipulating the geometry of your entire view, the CTM is the affine transform stored in your view's `transform` property. Inside your `drawRect:` method, the CTM is the affine transform associated with the active graphics context.

The coordinate system of each subview builds upon the coordinate systems of its ancestors. So when you modify a view's `transform` property, that change affects the view and all of its subviews. However, these changes affect only the final rendering of the views on the screen. Because each view draws its content and lays out its subviews relative to its own bounds, it can ignore its superview's transform during drawing and layout.

Figure 1-6 demonstrates how two different rotation factors combine visually when rendered. Inside the view's `drawRect:` method, applying a 45 degree rotation factor to a shape causes that shape to appear rotated by 45 degrees. Applying a separate 45 degree rotation factor to the view then causes the shape to appear to be rotated by 90 degrees. The shape is still rotated by only 45 degrees relative to the view that drew it, but the view rotation makes it appear to be rotated by more.

Figure 1-6 Rotating a view and its content



No rotations | **Shape rotated 45° during drawing** | **Shape and view each rotated 45°**

> **Important:** If a view's `transform` property is not the identity transform, the value of that view's `frame` property is undefined and must be ignored. When applying transforms to a view, you must use the view's `bounds` and `center` properties to get the size and position of the view. The frame rectangles of any subviews are still valid because they are relative to the view's bounds.

For information about modifying your view's transform property at runtime, see "Translating, Scaling, and Rotating Views" (page 47). For information about how to use transforms to position content during drawing, see *Drawing and Printing Guide for iOS*.

Points Versus Pixels

In iOS, all coordinate values and distances are specified using floating-point values in units referred to as **points**. The measurable size of a point varies from device to device and is largely irrelevant. The main thing to understand about points is that they provide a fixed frame of reference for drawing.

Table 1-1 lists the screen dimensions (measured in points) for different types of iOS-based devices in a portrait orientation. The width dimension is listed first, followed by the height dimension of the screen. As long as you design your interface to these screen sizes, your views will display correctly on the corresponding type of device.

Table 1-1 Screen dimensions for iOS-based devices

Device	Screen dimensions (in points)
iPhone and iPod touch devices with 4-inch Retina display	320 x 568
Other iPhone and iPod touch devices	320 x 480
iPad	768 x 1024

The point-based measuring system used for each type of device defines what is known as the **user coordinate space**. This is the standard coordinate space you use for nearly all of your code. For example, you use points and the user coordinate space when manipulating the geometry of a view or calling Core Graphics functions to draw the contents of your view. Although coordinates in the user coordinate space sometimes map directly to the pixels on the device's screen, you should never assume that this is the case. Instead, you should always remember the following:

One point does not necessarily correspond to one pixel on the screen.

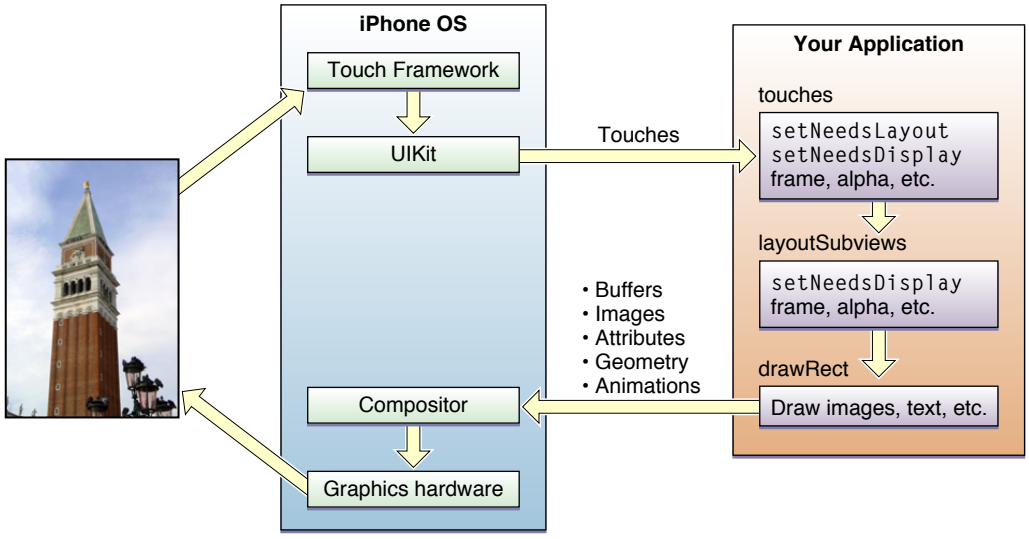
At the device level, all coordinates you specify in your view must be converted to pixels at some point. However, the mapping of points in the user coordinate space to pixels in the **device coordinate space** is normally handled by the system. Both UIKit and Core Graphics use a primarily vector-based drawing model where all coordinate values are specified using points. Thus, if you draw a curve using Core Graphics, you specify the curve using the same values, regardless of the resolution of the underlying screen.

When you need to work with images or other pixel-based technologies such as OpenGL ES, iOS provides help in managing those pixels. For static image files stored as resources in your application bundle, iOS defines conventions for specifying your images at different pixel densities and for loading the image that best matches the current screen resolution. Views also provide information about the current scale factor so that you can adjust any pixel-based drawing code manually to accommodate higher-resolution screens. The techniques for dealing with pixel-based content at different screen resolutions is described in "Supporting High-Resolution Screens In Views" in *Drawing and Printing Guide for iOS*.

The Runtime Interaction Model for Views

Any time a user interacts with your user interface, or any time your own code programmatically changes something, a complex sequence of events takes place inside of UIKit to handle that interaction. At specific points during that sequence, UIKit calls out to your view classes and gives them a chance to respond on behalf of your application. Understanding these callout points is important to understanding where your views fit into the system. Figure 1-7 shows the basic sequence of events that starts with the user touching the screen and ends with the graphics system updating the screen content in response. The same sequence of events would also occur for any programmatically initiated actions.

Figure 1-7 UIKit interactions with your view objects



The following steps break the event sequence in Figure 1-7 (page 23) down even further and explain what happens at each stage and how you might want your application to react in response.

1. The user touches the screen.
2. The hardware reports the touch event to the UIKit framework.
3. The UIKit framework packages the touch into a `UIEvent` object and dispatches it to the appropriate view. (For a detailed explanation of how UIKit delivers events to your views, see *Event Handling Guide for iOS* .)
4. The event-handling code of your view responds to the event. For example, your code might:
    - Change the properties (frame, bounds, alpha, and so on) of the view or its subviews.
    - Call the `setNeedsLayout` method to mark the view (or its subviews) as needing a layout update.
    - Call the `setNeedsDisplay` or `setNeedsDisplayInRect:` method to mark the view (or its subviews) as needing to be redrawn.
    - Notify a controller about changes to some piece of data.

Of course, it is up to you to decide which of these things the view should do and which methods it should call.

5. If the geometry of a view changed for any reason, UIKit updates its subviews according to the following rules:
    a. If you have configured autoresizing rules for your views, UIKit adjusts each view according to those rules. For more information about how autoresizing rules work, see "Handling Layout Changes Automatically Using Autoresizing Rules" (page 52).
    b. If the view implements the `layoutSubviews` method, UIKit calls it.

       You can override this method in your custom views and use it to adjust the position and size of any subviews. For example, a view that provides a large scrollable area would need to use several subviews as "tiles" rather than create one large view, which is not likely to fit in memory anyway. In its implementation of this method, the view would hide any subviews that are now offscreen or reposition them and use them to draw newly exposed content. As part of this process, the view's layout code can also invalidate any views that need to be redrawn.
6. If any part of any view was marked as needing to be redrawn, UIKit asks the view to redraw itself.

   For custom views that explicitly define a `drawRect:` method, UIKit calls that method. Your implementation of this method should redraw the specified area of the view as quickly as possible and nothing else. Do not make additional layout changes at this point and do not make other changes to your application's data model. The purpose of this method is to update the visual content of your view.

   Standard system views typically do not implement a `drawRect:` method but instead manage their drawing at this time.
7. Any updated views are composited with the rest of the application's visible content and sent to the graphics hardware for display.
8. The graphics hardware transfers the rendered content to the screen.

Note: The preceding update model applies primarily to applications that use standard system views and drawing techniques. Applications that use OpenGL ES for drawing typically configure a single full-screen view and draw directly to the associated OpenGL ES graphics context. In such a case, the view may still handle touch events but, because it is full-screen, it would not need to lay out subviews. For more information about using OpenGL ES, see *OpenGL ES Programming Guide for iOS*.

In the preceding set of steps, the primary integration points for your own custom views are:

- The event-handling methods:
    - `touchesBegan:withEvent:`
    - `touchesMoved:withEvent:`

  - `touchesEnded:withEvent:`
  - `touchesCancelled:withEvent:`
- The `layoutSubviews` method
- The `drawRect:` method

These are the most commonly overridden methods for views but you may not need to override all of them. If you use gesture recognizers to handle events, you do not need to override any of the event-handling methods. Similarly, if your view does not contain subviews or its size does not change, there is no reason to override the `layoutSubviews` method. Finally, the `drawRect:` method is needed only when the contents of your view can change at runtime and you are using native technologies such as UIKit or Core Graphics to do your drawing.

It is also important to remember that these are the primary integration points but not the only ones. Several methods of the `UIView` class are designed to be override points for subclasses. You should look at the method descriptions in *UIView Class Reference* to see which methods might be appropriate for you to override in your custom implementations.

Tips for Using Views Effectively

Custom views are useful for situations where you need to draw something the standard system views do not provide, but it is your responsibility to ensure that the performance of your views is good enough. UIKit does everything it can to optimize view-related behaviors and help you achieve good performance in your custom views. However, you can help UIKit in this aspect by considering the following tips.

> **Important:** Before optimizing your drawing code, you should always gather data about your view's current performance. Measuring the current performance lets you confirm whether there actually is a problem and, if there is, gives you a baseline measurement against which you can compare future optimizations.

Views Do Not Always Have a Corresponding View Controller

There is rarely a one-to-one relationship between individual views and view controllers in your application. The job of a view controller is to manage a view hierarchy, which often consists of more than one view used to implement some self-contained feature. For iPhone applications, each view hierarchy typically fills the entire screen, although for iPad applications a view hierarchy may fill only part of the screen.

As you design your application's user interface, it is important to consider the role that view controllers will play. View controllers provide a lot of important behaviors, such as coordinating the presentation of views on the screen, coordinating the removal of those views from the screen, releasing memory in response to low-memory warnings, and rotating views in response to interface orientation changes. Circumventing these behaviors could cause your application to behave incorrectly or in unexpected ways.

For more information view controllers and their role in applications, see *View Controller Programming Guide for iOS*.

Minimize Custom Drawing

Although custom drawing is necessary at times, it is also something you should avoid whenever possible. The only time you should truly do any custom drawing is when the existing system view classes do not provide the appearance or capabilities that you need. Any time your content can be assembled with a combination of existing views, your best bet is to combine those view objects into a custom view hierarchy.

Take Advantage of Content Modes

Content modes minimize the amount of time spent redrawing your views. By default, views use the `UIViewContentModeScaleToFill` content mode, which scales the view's existing contents to fit the view's frame rectangle. You can change this mode as needed to adjust your content differently, but you should avoid using the `UIViewContentModeRedraw` content mode if you can. Regardless of which content mode is in effect, you can always force your view to redraw its contents by calling `setNeedsDisplay` or `setNeedsDisplayInRect:`.

Declare Views as Opaque Whenever Possible

UIKit uses the `opaque` property of each view to determine whether the view can optimize compositing operations. Setting the value of this property to `YES` for a custom view tells UIKit that it does not need to render any content behind your view. Less rendering can lead to increased performance for your drawing code and is generally encouraged. Of course, if you set the `opaque` property to `YES`, your view *must* fill its bounds rectangle completely with fully opaque content.

Adjust Your View's Drawing Behavior When Scrolling

Scrolling can incur numerous view updates in a short amount of time. If your view's drawing code is not tuned appropriately, scrolling performance for your view could be sluggish. Rather than trying to ensure that your view's content is pristine at all times, consider changing your view's behavior when a scrolling operation begins. For example, you can reduce the quality of your rendered content temporarily or change the content mode while a scroll is in progress. When scrolling stops, you can then return your view to its previous state and update the contents as needed.

Do Not Customize Controls by Embedding Subviews

Although it is technically possible to add subviews to the standard system controls—objects that inherit from `UIControl`—you should never customize them in this way. Controls that support customizations do so through explicit and well-documented interfaces in the control class itself. For example, the `UIButton` class contains methods for setting the title and background images for the button. Using the defined customization points means that your code will always work correctly. Circumventing these methods, by embedding a custom image view or label inside the button, might cause your application to behave incorrectly now or at some point in the future if the button's implementation changes.

Windows

Every iOS application needs at least one window—an instance of the `UIWindow` class—and some may include more than one window. A window object has several responsibilities:

- It contains your application's visible content.
- It plays a key role in the delivery of touch events to your views and other application objects.
- It works with your application's view controllers to facilitate orientation changes.

In iOS, windows do not have title bars, close boxes, or any other visual adornments. A window is always just a blank container for one or more views. Also, applications do not change their content by showing new windows. When you want to change the displayed content, you change the frontmost views of your window instead.

Most iOS applications create and use only one window during their lifetime. This window spans the entire main screen of the device and is loaded from the application's main nib file (or created programmatically) early in the life of the application. However, if an application supports the use of an external display for video out, it can create an additional window to display content on that external display. All other windows are typically created by the system, and are usually created in response to specific events, such as an incoming phone call.

Tasks That Involve Windows

For many applications, the only time the application interacts with its window is when it creates the window at startup. However, you can use your application's window object to perform a few application-related tasks:

- **Use the window object to convert points and rectangles to or from the window's local coordinate system.** For example, if you are provided with a value in window coordinates, you might want to convert it to the coordinate system of a specific view before trying to use it. For information on how to convert coordinates, see "Converting Coordinates in the View Hierarchy" (page 50).
- **Use window notifications to track window-related changes.** Windows generate notifications when they are shown or hidden or when they accept or resign the key status. You can use these notifications to perform actions in other parts of your application. For more information, see "Monitoring Window Changes" (page 31).

Creating and Configuring a Window

You can create and configure your application's main window programmatically or using Interface Builder. In either case, you create the window at launch time and should retain it and store a reference to it in your application delegate object. If your application creates additional windows, have the application create them lazily when they are needed. For example, if your application supports displaying content on an external display, it should wait until a display is connected before creating the corresponding window.

You should always create your application's main window at launch time regardless of whether your application is being launched into the foreground or background. Creating and configuring a window is not an expensive operation by itself. However, if your application is launched straight into the background, you should avoid making the window visible until your application enters the foreground.

Creating Windows in Interface Builder

Creating your application's main window using Interface Builder is simple because the Xcode project templates do it for you. Every new Xcode application project includes a main nib file (usually with the name `MainWindow.xib` or some variant thereof) that includes the application's main window. In addition, these templates also define an outlet for that window in the application delegate object. You use this outlet to access the window object in your code.

> **Important:** When creating your window in Interface Builder, it is recommended that you enable the Full Screen at Launch option in the attributes inspector. If this option is not enabled and your window is smaller than the screen of the target device, touch events will not be received by some of your views. This is because windows (like all views) do not receive touch events outside of their bounds rectangle. Because views are not clipped to the window's bounds by default, the views still appear visible but events do not reach them. Enabling the Full Screen at Launch option ensures that the window is sized appropriately for the current screen.

If you are retrofitting a project to use Interface Builder, creating a window using Interface Builder is a simple matter of dragging a window object to your nib file. Of course, you should also do the following:

- To access the window at runtime, you should connect the window to an outlet, typically one defined in your application delegate or the File's Owner of the nib file.
- If your retrofit plans include making your new nib file the main nib file of your application, you must also set the `NSMainNibFile` key in your application's `Info.plist` file to the name of your nib file. Changing the value of this key ensures that the nib file is loaded and available for use by the time the `application:didFinishLaunchingWithOptions:` method of your application delegate is called.

For more information about creating and configuring nib files, see *Interface Builder User Guide*. For information about how to load nib files into your application at runtime, see "Nib Files" in *Resource Programming Guide*.

Creating a Window Programmatically

If you prefer to create your application's main window programmatically, you should include code similar to the following in the `application:didFinishLaunchingWithOptions:` method of your application delegate:

```
self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]]
autorelease];
```

In the preceding example, `self.window` is assumed to be a declared property of your application delegate that is configured to retain the window object. If you were creating a window for an external display instead, you would assign it to a different variable and you would need to specify the bounds of the non main `UIScreen` object representing that display.

When creating windows, you should always set the size of the window to the full bounds of the screen. You should not reduce the size of the window to accommodate the status bar or any other items. The status bar always floats on top of the window anyway, so the only thing you should shrink to accommodate the status bar is the view you put into your window. And if you are using view controllers, the view controller should handle the sizing of your views automatically.

Adding Content to Your Window

Each window typically has a single root view object (managed by a corresponding view controller) that contains all of the other views representing your content. Using a single root view simplifies the process of changing your interface; to display new content, all you have to do is replace the root view. To install a view in your window, use the `addSubview:` method. For example, to install a view that is managed by a view controller, you would use code similar to the following:

```
[window addSubview:viewController.view];
```

In place of the preceding code, you can alternatively configure the `rootViewController` property of the window in your nib file. This property offers a convenient way to configure the root view of the window using a nib file instead of programmatically. If this property is set when the window is loaded from its nib file, UIKit automatically installs the view from the associated view controller as the root view of the window. This property is used only to install the root view and is not used by the window to communicate with the view controller.

You can use any view you want for a window's root view. Depending on your interface design, the root view can be a generic `UIView` object that acts as a container for one or more subviews, the root view can be a standard system view, or the root view can be a custom view that you define. Some standard system views that are commonly used as root views include scroll views, table views, and image views.

When configuring the root view of the window, you are responsible for setting its initial size and position within the window. For applications that do not include a status bar, or that display a translucent status bar, set the view size to match the size of the window. For applications that show an opaque status bar, position your view below the status bar and reduce its size accordingly. Subtracting the status bar height from the height of your view prevents the top portion of your view from being obscured.

> **Note:** If the root view of your window is provided by a container view controller (such as a tab bar controller, navigation controller, or split-view controller), you do not need to set the initial size of the view yourself. The container view controller automatically sizes its view appropriately based on whether the status bar is visible.

Changing the Window Level

Each `UIWindow` object has a configurable `windowLevel` property that determines how that window is positioned relative to other windows. For the most part, you should not need to change the level of your application's windows. New windows are automatically assigned to the normal window level at creation time. The normal window level indicates that the window presents application-related content. Higher window levels are reserved for information that needs to float above the application content, such as the system status bar or alert messages. And although you can assign windows to these levels yourself, the system usually does this for you when you use specific interfaces. For example, when you show or hide the status bar or display an alert view, the system automatically creates the needed windows to display those items.

Monitoring Window Changes

If you want to track the appearance or disappearance of windows inside your application, you can do so using these window-related notifications:

- `UIWindowDidBecomeVisibleNotification`
- `UIWindowDidBecomeHiddenNotification`
- `UIWindowDidBecomeKeyNotification`
- `UIWindowDidResignKeyNotification`

These notifications are delivered in response to programmatic changes in your application's windows. Thus, when your application shows or hides a window, the `UIWindowDidBecomeVisibleNotification` and `UIWindowDidBecomeHiddenNotification` notifications are delivered accordingly. These notifications are not delivered when your application moves into the background execution state. Even though your window is not displayed on the screen while your application is in the background, it is still considered visible within the context of your application.

The `UIWindowDidBecomeKeyNotification` and `UIWindowDidResignKeyNotification` notifications help your application keep track of which window is the **key window**—that is, which window is currently receiving keyboard events and other non touch-related events. Whereas touch events are delivered to the window in which the touch occurred, events that do not have an associated coordinate value are delivered to the key window of your application. Only one window at a time may be key.

Displaying Content on an External Display

To display content on an external display, you must create an additional window for your application and associate it with the screen object representing the external display. New windows are normally associated with the main screen by default. Changing the window's associated screen object causes the contents of that window to be rerouted to the corresponding display. Once the window is associated with the correct screen, you can add views to it and show it just like you do for your application's main screen.

The `UIScreen` class maintains a list of screen objects representing the available hardware displays. Normally, there is only one screen object representing the main display for any iOS-based device, but devices that support connecting to an external display can have an additional screen object available. Devices that support an external display include iPhone and iPod touch devices that have Retina displays and the iPad. Older devices, such as iPhone 3GS, do not support external displays.

> **Note:** Because external displays are essentially a video-out connection, you should not expect touch events for views and controls in a window that is associated with an external display. In addition, it is your application's responsibility to update the contents of the window as needed. Thus, to mirror the contents of your main window, your application would need to create a duplicate set of views for the external display's window and update them in tandem with the views in your main window.

The process for displaying content on an external display is described in the following sections. However, the following steps summarize the basic process:

1. At application startup, register for the screen connection and disconnection notifications.
2. When it is time to display content on the external display, create and configure a window.
    - Use the `screens` property of `UIScreen` to obtain the screen object for the external display.
    - Create a `UIWindow` object and size it appropriately for the screen (or for your content).
    - Assign the `UIScreen` object for the external display to the `screen` property of the window.
    - Adjust the resolution of the screen object as needed to support your content.
    - Add any appropriate views to the window.

3. Show the window and update it normally.

Handling Screen Connection and Disconnection Notifications

Screen connection and disconnection notifications are crucial for handling changes to external displays gracefully. When the user connects or disconnects a display, the system sends appropriate notifications to your application. You should use these notifications to update your application state and create or release the window associated with the external display.

The important thing to remember about the connection and disconnection notifications is that they can come at any time, even when your application is suspended in the background. Therefore, it is best to observe the notifications from an object that is going to exist for the duration of your application's runtime, such as your application delegate. If your application is suspended, the notifications are queued until your application exits the suspended state and starts running in either the foreground or background.

Listing 2-1 shows the code used to register for connection and disconnection notifications. This method is called by the application delegate at initialization time but you could register for these notifications from other places in your application, too. The implementation of the handler methods is shown in Listing 2-2 (page 34).

Listing 2-1 Registering for screen connect and disconnect notifications

```
- (void)setupScreenConnectionNotificationHandlers
{
    NSNotificationCenter* center = [NSNotificationCenter defaultCenter];

    [center addObserver:self selector:@selector(handleScreenConnectNotification:)
            name:UIScreenDidConnectNotification object:nil];
   [center addObserver:self selector:@selector(handleScreenDisconnectNotification:)
            name:UIScreenDidDisconnectNotification object:nil];
}
```

If your application is active when an external display is attached to the device, it should create a second window for that display and fill it with some content. The content does not need to be the final content you want to present. For example, if your application is not ready to use the extra screen, it can use the second window to display some placeholder content. If you do not create a window for the screen, or if you create a window but do not show it, a black field is displayed on the external display.

Listing 2-2 shows how to create a secondary window and fill it with some content. In this example, the application creates the window in the handler methods it uses to receive screen connection notifications. (For information about registering for connection and disconnection notifications, see Listing 2-1 (page 33).) The handler method for the connection notification creates a secondary window, associates it with the newly connected screen and calls a method of the application's main view controller to add some content to the window and show it. The handler method for the disconnection notification releases the window and notifies the main view controller so that it can adjust its presentation accordingly.

Listing 2-2 Handling connect and disconnect notifications

```
- (void)handleScreenConnectNotification:(NSNotification*)aNotification
{
    UIScreen*    newScreen = [aNotification object];
    CGRect        screenBounds = newScreen.bounds;

    if (!_secondWindow)
    {
        _secondWindow = [[UIWindow alloc] initWithFrame:screenBounds];
        _secondWindow.screen = newScreen;

        // Set the initial UI for the window.
        [viewController displaySelectionInSecondaryWindow:_secondWindow];
    }
}

- (void)handleScreenDisconnectNotification:(NSNotification*)aNotification
{
    if (_secondWindow)
    {
        // Hide and then delete the window.
        _secondWindow.hidden = YES;
        [_secondWindow release];
        _secondWindow = nil;

        // Update the main screen based on what is showing here.
        [viewController displaySelectionOnMainScreen];
```

```
    }

}
```

Configuring a Window for an External Display

To display a window on an external screen, you must associate it with the correct screen object. This process involves locating the proper `UIScreen` object and assigning it to the window's `screen` property. You can get the list of screen objects from the `screens` class method of `UIScreen`. The array returned by this method always contains at least one object representing the main screen. If a second object is present, that object represents a connected external display.

Listing 2-3 shows a method that is called at application startup to see if an external display is already attached. If it is, the method creates a window, associates it with the external display, and adds some placeholder content before showing the window. In this case, the placeholder content is a white background and a label indicating that there is no content to display. To show the window, this method changes the value of its `hidden` property rather than calling `makeKeyAndVisible`. It does this because the window contains only static content and is not used to handle events.

Listing 2-3 Configuring a window for an external display

```
- (void)checkForExistingScreenAndInitializeIfPresent
{
    if ([[UIScreen screens] count] > 1)
    {
        // Associate the window with the second screen.
        // The main screen is always at index 0.
        UIScreen*    secondScreen = [[UIScreen screens] objectAtIndex:1];
        CGRect        screenBounds = secondScreen.bounds;

        _secondWindow = [[UIWindow alloc] initWithFrame:screenBounds];
        _secondWindow.screen = secondScreen;

        // Add a white background to the window
       UIView*           whiteField = [[UIView alloc] initWithFrame:screenBounds];
        whiteField.backgroundColor = [UIColor whiteColor];

        [_secondWindow addSubview:whiteField];
```

```
        [whiteField release];


        // Center a label in the view.
       NSString*    noContentString = [NSString stringWithFormat:@"<no content>"];
        CGSize        stringSize = [noContentString sizeWithFont:[UIFont
systemFontOfSize:18]];


        CGRect        labelSize = CGRectMake((screenBounds.size.width -
stringSize.width) / 2.0,
                                     (screenBounds.size.height - stringSize.height)
 / 2.0,
                                      stringSize.width, stringSize.height);


        UILabel*    noContentLabel = [[UILabel alloc] initWithFrame:labelSize];
        noContentLabel.text = noContentString;
        noContentLabel.font = [UIFont systemFontOfSize:18];
        [whiteField addSubview:noContentLabel];


        // Go ahead and show the window.
        _secondWindow.hidden = NO;
    }
}
```

Important: You should always associate a screen with a window before showing the window. While it is possible to change screens for a window that is currently visible, doing so is an expensive operation and should be avoided.

As soon as the window for an external screen is displayed, your application can begin updating it like any other window. You can add and remove subviews as needed, change the contents of subviews, animate changes to the views, and invalidate their contents as needed.

Configuring the Screen Mode of an External Display

Depending on your content, you might want to change the screen mode before associating your window with it. Many screens support multiple resolutions, some of which use different pixel aspect ratios. Screen objects use the most common screen mode by default, but you can change that mode to one that is more suitable for your content. For example, if you are implementing a game using OpenGL ES and your textures are designed for a 640 x 480 pixel screen, you might change the screen mode for screens with higher default resolutions.

If you plan to use a screen mode other than the default one, you should apply that mode to the `UIScreen` object before associating the screen with a window. The `UIScreenMode` class defines the attributes of a single screen mode. You can get a list of the modes supported by a screen from its `availableModes` property and iterate through the list for one that matches your needs.

For more information about screen modes, see *UIScreenMode Class Reference*.

Views

Because view objects are the main way your application interacts with the user, they have many responsibilities. Here are just a few:

- Layout and subview management
  - A view defines its own default resizing behaviors in relation to its parent view.
  - A view can manage a list of subviews.
  - A view can override the size and position of its subviews as needed.
  - A view can convert points in its coordinate system to the coordinate systems of other views or the window.
- Drawing and animation
  - A view draws content in its rectangular area.
  - Some view properties can be animated to new values.
- Event handling
  - A view can receive touch events.
  - A view participates in the responder chain.

This chapter focuses on the steps for creating, managing, and drawing views and for handling the layout and management of view hierarchies. For information about how to handle touch events (and other events) in your views, see *Event Handling Guide for iOS* .

Creating and Configuring View Objects

You create views as self-contained objects either programmatically or using Interface Builder, and then you assemble them into view hierarchies for use.

Creating View Objects Using Interface Builder

The simplest way to create views is to assemble them graphically using Interface Builder. From Interface Builder, you can add views to your interface, arrange those views into hierarchies, configure each view's settings, and connect view-related behaviors to your code. Because Interface Builder uses live view objects—that is, actual instances of the view classes—what you see at design time is what you get at runtime. You then save those live objects in a nib file, which is a resource file that preserves the state and configuration of your objects.

You usually create nib files in order to store an entire view hierarchy for one of your application's view controllers. The top level of the nib file usually contains a single view object that represents your view controller's view. (The view controller itself is typically represented by the File's Owner object.) The top-level view should be sized appropriately for the target device and contain all of the other views that are to be presented. It is rare to use a nib file to store only a portion of your view controller's view hierarchy.

When using nib files with a view controller, all you have to do is initialize the view controller with the nib file information. The view controller handles the loading and unloading of your views at the appropriate times. However, if your nib file is not associated with a view controller, you can load the nib file contents manually using an `NSBundle` or `UINib` object, which use the data in the nib file to reconstitute your view objects.

For more information about how to use Interface Builder to create and configure your views, see *Interface Builder User Guide*. For information about how view controllers load and manage their associated nib files, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*. For more information about how to load views programmatically from a nib file, see "Nib Files" in *Resource Programming Guide*.

Creating View Objects Programmatically

If you prefer to create views programmatically, you can do so using the standard allocation/initialization pattern. The default initialization method for views is the `initWithFrame:` method, which sets the initial size and position of the view relative to its (soon-to-be-established) parent view. For example, to create a new generic `UIView` object, you could use code similar to the following:

```
CGRect  viewRect = CGRectMake(0, 0, 100, 100);
UIView* myView = [[UIView alloc] initWithFrame:viewRect];
```

> **Note:** Although all views support the `initWithFrame:` method, some may have a preferred initialization method that you should use instead. For information about any custom initialization methods, see the reference documentation for the class.

After you create a view, you must add it to a window (or to another view in a window) before it can become visible. For information on how to add views to your view hierarchy, see "Adding and Removing Subviews" (page 43).

Setting the Properties of a View

The `UIView` class has several declared properties for controlling the appearance and behavior of the view. These properties are for manipulating the size and position of the view, the view's transparency, its background color, and its rendering behavior. All of these properties have appropriate default values that you can change later as needed. You can also configure many of these properties from Interface Builder using the Inspector window.

Table 3-1 lists some of the more commonly used properties (and some methods) and describes their usage. Related properties are listed together so that you can see the options you have for affecting certain aspects of the view.

Table 3-1 Usage of some key view properties

Properties	Usage
`alpha`, `hidden`, `opaque`	These properties affect the opacity of the view. The `alpha` and `hidden` properties change the view's opacity directly. The `opaque` property tells the system how it should composite your view. Set this property to `YES` if your view's content is fully opaque and therefore does not reveal any of the underlying view's content. Setting this property to `YES` improves performance by eliminating unnecessary compositing operations.

Properties	Usage
`bounds`, `frame`, `center`, `transform`	These properties affect the size and position of the view. The `center` and `frame` properties represent the position of the view relative to its parent view. The `frame` also includes the size of the view. The `bounds` property defines the view's visible content area in its own coordinate system. The `transform` property is used to animate or move the entire view in complex ways. For example, you would use a transform to rotate or scale the view. If the current transform is not the identity transform, the `frame` property is undefined and should be ignored. For information about the relationship between the `bounds`, `frame`, and `center` properties, see "The Relationship of the Frame, Bounds, and Center Properties" (page 18). For information about how transforms affect a view, see "Coordinate System Transformations" (page 20).
`autoresizingMask`, `autoresizesSubviews`	These properties affect the automatic resizing behavior of the view and its subviews. The `autoresizingMask` property controls how a view responds to changes in its parent view's bounds. The `autoresizesSubviews` property controls whether the current view's subviews are resized at all.
`contentMode`, `contentStretch`, `contentScaleFactor`	These properties affect the rendering behavior of content inside the view. The `contentMode` and `contentStretch` properties determine how the content is treated when the view's width or height changes. The `contentScaleFactor` property is used only when you need to customize the drawing behavior of your view for high-resolution screens. For more information on how the content mode affects your view, see "Content Modes" (page 13). For information about how the content stretch rectangle affects your view, see "Stretchable Views" (page 15). For information about how to handle scale factors, see "Supporting High-Resolution Screens In Views" in *Drawing and Printing Guide for iOS*.
`gestureRecognizers`, `userInteraction-Enabled`, `multipleTouchEnabled`, `exclusiveTouch`	These properties affect how your view processes touch events. The `gestureRecognizers` property contains gesture recognizers attached to the view. The other properties control what touch events the view supports. For information about how to respond to events in your views, see *Event Handling Guide for iOS*.

Properties	Usage
`backgroundColor`, `subviews`, `drawRect:` method, `layer`, (`layerClass` method)	These properties and methods help you manage the actual content of your view. For simple views, you can set a background color and add one or more subviews. The `subviews` property itself contains a read-only list of subviews, but there are several methods for adding and rearranging subviews. For views with custom drawing behavior, you must override the `drawRect:` method. For more advanced content, you can work directly with the view's Core Animation `layer`. To specify an entirely different type of layer for the view, you must override the `layerClass` method.

For information about the basic properties common to all views, see *UIView Class Reference* . For more information about specific properties of a view, see the reference documentation for that view.

Tagging Views for Future Identification

The `UIView` class contains a `tag` property that you can use to tag individual view objects with an integer value. You can use tags to uniquely identify views inside your view hierarchy and to perform searches for those views at runtime. (Tag-based searches are faster than iterating the view hierarchy yourself.) The default value for the `tag` property is `0`.

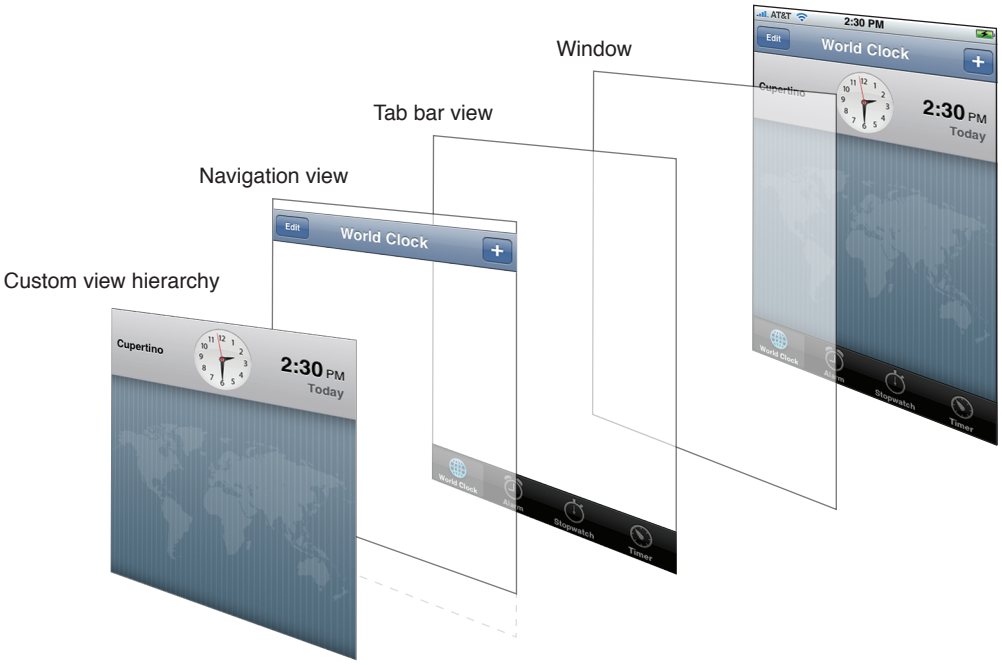
To search for a tagged view, use the `viewWithTag:` method of `UIView`. This method performs a depth-first search of the receiver and its subviews. It does not search superviews or other parts of the view hierarchy. Thus, calling this method from the root view of a hierarchy searches all views in the hierarchy but calling it from a specific subview searches only a subset of views.

Creating and Managing a View Hierarchy

Managing view hierarchies is a crucial part of developing your application's user interface. The organization of your views influences both the visual appearance of your application and how your application responds to changes and events. For example, the parent-child relationships in the view hierarchy determine which objects might handle a specific touch event. Similarly, parent-child relationships define how each view responds to interface orientation changes.

Figure 3-1 shows an example of how the layering of views creates the desired visual effect for an application. In the case of the Clock application, the view hierarchy is composed of a mixture of views derived from different sources. The tab bar and navigation views are special view hierarchies provided by the tab bar and navigation controller objects to manage portions of the overall user interface. Everything between those bars belongs to the custom view hierarchy that the Clock application provides.

Figure 3-1 Layered views in the Clock application



There are several ways to build view hierarchies in iOS applications, including graphically in Interface Builder and programmatically in your code. The following sections show you how to assemble your view hierarchies and, having done that, how to find views in the hierarchy and convert between different view coordinate systems.

Adding and Removing Subviews

Interface Builder is the most convenient way to build view hierarchies because you assemble your views graphically, see the relationships between the views, and see exactly how those views will appear at runtime. When using Interface Builder, you save your resulting view hierarchy in a nib file, which you load at runtime as the corresponding views are needed.

If you prefer to create your views programmatically instead, you create and initialize them and then use the following methods to arrange them into hierarchies:

- To add a subview to a parent, call the `addSubview:` method of the parent view. This method adds the subview to the end of the parent's list of subviews.
- To insert a subview in the middle of the parent's list of subviews, call any of the `insertSubview:...` methods of the parent view. Inserting a subview in the middle of the list visually places that view behind any views that come later in the list.
- To reorder existing subviews inside their parent, call the `bringSubviewToFront:`, `sendSubviewToBack:`, or `exchangeSubviewAtIndex:withSubviewAtIndex:` methods of the parent view. Using these methods is faster than removing the subviews and reinserting them.
- To remove a subview from its parent, call the `removeFromSuperview` method of the subview (not the parent view).

When adding a subview to its parent, the subview's current frame rectangle denotes its initial position inside the parent view. A subview whose frame lies outside of its superview's visible bounds is not clipped by default. If you want your subview to be clipped to the superview's bounds, you must explicitly set the `clipsToBounds` property of the superview to `YES`.

The most common example of adding a subview to another view occurs in the `application:didFinishLaunchingWithOptions:` method of almost every application. Listing 3-1 shows a version of this method that installs the view from the application's main view controller into the application window. Both the window and the view controller are stored in the application's main nib file, which is loaded before the method is called. However, the view hierarchy managed by the view controller is not actually loaded until the `view` property is accessed.

Listing 3-1 Adding a view to a window

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

    return YES;
}
```

Another common place where you might add subviews to a view hierarchy is in the `loadView` or `viewDidLoad` methods of a view controller. If you are building your views programmatically, you put your view creation code in the `loadView` method of your view controller. Whether you create your views programmatically or load them from a nib file, you could include additional view configuration code in the `viewDidLoad` method.

Listing 3-2 shows the `viewDidLoad` method of the `TransitionsViewController` class from the *UICatalog* sample application. The `TransitionsViewController` class manages the animations associated with transitioning between two views. The application's initial view hierarchy (consisting of a root view and toolbar) is loaded from a nib file. The code in the `viewDidLoad` method subsequently creates the container view and image views used to manage the transitions. The purpose of the container view is to simplify the code needed to implement the transition animations between the two image views. The container view has no real content of its own.

Listing 3-2 Adding views to an existing view hierarchy

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = NSLocalizedString(@"TransitionsTitle", @"");

   // create the container view which we will use for transition animation (centered
 horizontally)
    CGRect frame = CGRectMake(round((self.view.bounds.size.width - kImageWidth) /
 2.0),
                                                        kTopPlacement, kImageWidth,
 kImageHeight);
    self.containerView = [[[UIView alloc] initWithFrame:frame] autorelease];
    [self.view addSubview:self.containerView];

    // The container view can represent the images for accessibility.
    [self.containerView setIsAccessibilityElement:YES];
    [self.containerView setAccessibilityLabel:NSLocalizedString(@"ImagesTitle",
@"")];

    // create the initial image view
    frame = CGRectMake(0.0, 0.0, kImageWidth, kImageHeight);
    self.mainView = [[[UIImageView alloc] initWithFrame:frame] autorelease];
    self.mainView.image = [UIImage imageNamed:@"scene1.jpg"];
```

```
    [self.containerView addSubview:self.mainView];

    // create the alternate image view (to transition between)
    CGRect imageFrame = CGRectMake(0.0, 0.0, kImageWidth, kImageHeight);
    self.flipToView = [[[UIImageView alloc] initWithFrame:imageFrame] autorelease];
    self.flipToView.image = [UIImage imageNamed:@"scene2.jpg"];
}
```

Important: Superviews automatically retain their subviews, so after embedding a subview it is safe to release that subview. In fact, doing so is recommended because it prevents your application from retaining the view one time too many and causing a memory leak later. Just remember that if you remove a subview from its superview and intend to reuse it, you must retain the subview again. The `removeFromSuperview` method autoreleases a subview before removing it from its superview. If you do not retain the view before the next event loop cycle, the view will be released.

For more information about Cocoa memory management conventions, see *Advanced Memory Management Programming Guide*.

When you add a subview to another view, UIKit notifies both the parent and child views of the change. If you implement custom views, you can intercept these notifications by overriding one or more of the `willMoveToSuperview:`, `willMoveToWindow:`, `willRemoveSubview:`, `didAddSubview:`, `didMoveToSuperview`, or `didMoveToWindow` methods. You can use these notifications to update any state information related to your view hierarchy or to perform additional tasks.

After creating a view hierarchy, you can navigate it programmatically using the `superview` and `subviews` properties of your views. The `window` property of each view contains the window in which that view is currently displayed (if any). Because the root view in a view hierarchy has no parent, its `superview` property is set to `nil`. For views that are currently onscreen, the window object is the root view of the view hierarchy.

Hiding Views

To hide a view visually, you can either set its `hidden` property to `YES` or change its `alpha` property to `0.0`. A hidden view does not receive touch events from the system. However, hidden views do participate in autoresizing and other layout operations associated with the view hierarchy. Thus, hiding a view is often a convenient alternative to removing views from your view hierarchy, especially if you plan to show the views again at some point soon.

Important: If you hide a view that is currently the first responder, the view does not automatically resign its first responder status. Events targeted at the first responder are still delivered to the hidden view. To prevent this from happening, you should force your view to resign the first responder status when you hide it. For more information about the responder chain, see *Event Handling Guide for iOS*.

If you want to animate a view's transition from visible to hidden (or the reverse), you must do so using the view's `alpha` property. The `hidden` property is not an animatable property, so any changes you make to it take effect immediately.

Locating Views in a View Hierarchy

There are two ways to locate views in a view hierarchy:

- Store pointers to any relevant views in an appropriate location, such as in the view controller that owns the views.
- Assign a unique integer to each view's `tag` property and use the `viewWithTag:` method to locate it.

Storing references to relevant views is the most common approach to locating views and makes accessing those views very convenient. If you used Interface Builder to create your views, you can connect objects in your nib file (including the File's Owner object that represents the managing controller object) to one another using outlets. For views you create programmatically, you can store references to those views in private member variables. Whether you use outlets or private member variables, you are responsible for retaining the views as needed and then releasing them as well. The best way to ensure objects are retained and released properly is to use declared properties.

Tags are a useful way to reduce hard-coded dependencies and support more dynamic and flexible solutions. Rather than storing a pointer to a view, you could locate it using its tag. Tags are also a more persistent way of referring to views. For example, if you wanted to save the list of views that are currently visible in your application, you would write out the tags of each visible view to a file. This is simpler than archiving the actual view objects, especially in situations where you are tracking only which views are currently visible. When your application is subsequently loaded, you would then re-create your views and use the saved list of tags to set the visibility of each view, and thereby return your view hierarchy to its previous state.

Translating, Scaling, and Rotating Views

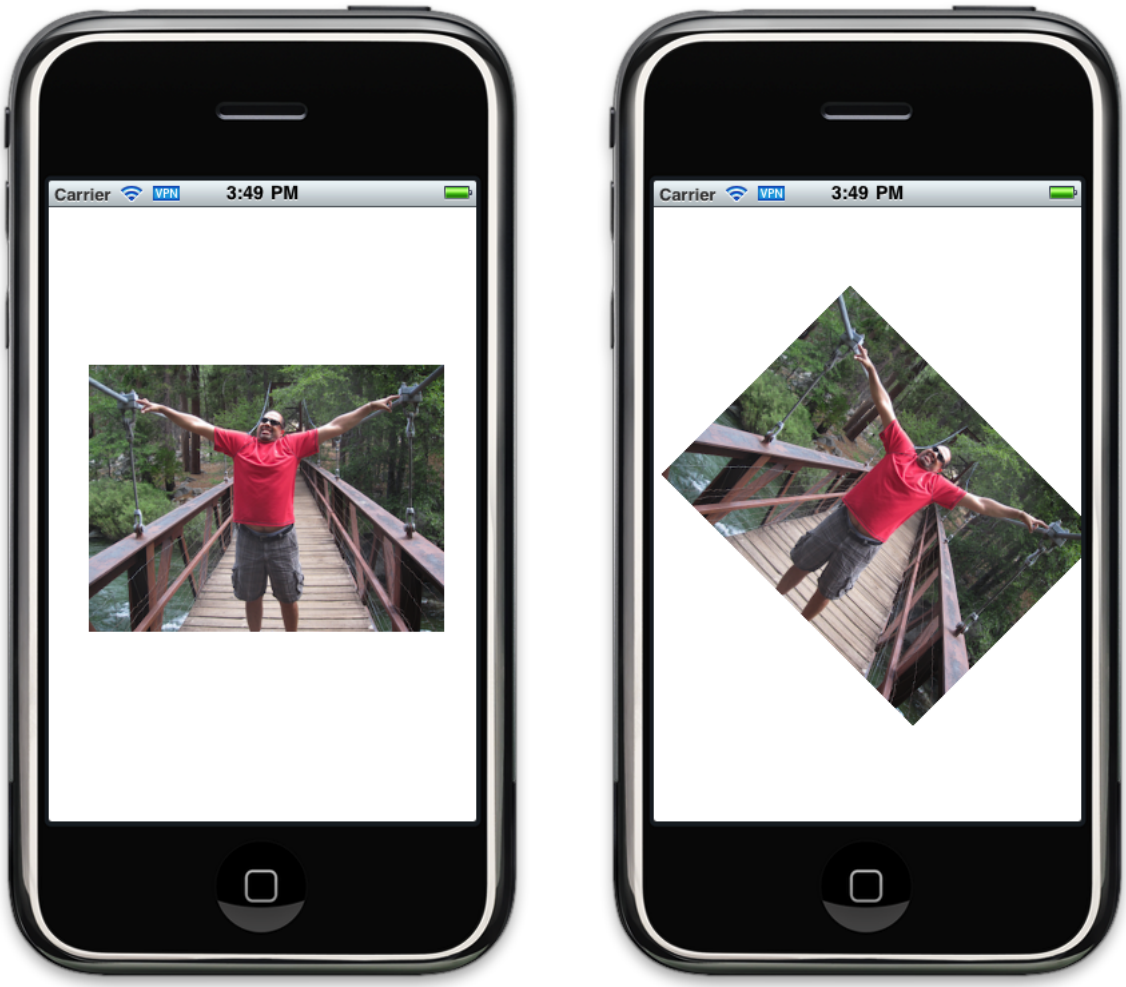
Every view has an associated affine transform that you can use to translate, scale, or rotate the view's content. View transforms alter the final rendered appearance of the view and are often used to implement scrolling, animations, or other visual effects.

The `transform` property of `UIView` contains a `CGAffineTransform` structure with the transformations to apply. By default, this property is set to the identity transform, which does not modify the appearance of the view. You can assign a new transform to this property at any time. For example, to rotate a view by 45 degrees, you could use the following code:

```
// M_PI/4.0 is one quarter of a half circle, or 45 degrees.
CGAffineTransform xform = CGAffineTransformMakeRotation(M_PI/4.0);
self.view.transform = xform;
```

Applying the transform in the preceding code to a view would rotate that view clockwise about its center point. Figure 3-2 shows how this transformation would look if it were applied to an image view embedded in an application.

Figure 3-2 Rotating a view 45 degrees



Unrotated **Rotated 45˚**

When applying multiple transformations to a view, the order in which you add those transformations to the `CGAffineTransform` structure is significant. Rotating the view and then translating it is not the same as translating the view and then rotating it. Even if the amounts of rotation and translation are the same in each case, the sequence of the transformations affects the final results. In addition, any transformations you add are applied to the view relative to its center point. Thus, applying a rotation factor rotates the view around its center point. Scaling a view changes the width and height of the view but does not change its center point.

For more information about creating and using affine transforms, see "Transforms" in *Quartz 2D Programming Guide*.

Converting Coordinates in the View Hierarchy

At various times, particularly when handling events, an application may need to convert coordinate values from one frame of reference to another. For example, touch events report the location of each touch in the window's coordinate system but view objects often need that information in the view's local coordinate system. The `UIView` class defines the following methods for converting coordinates to and from the view's local coordinate system:

```
convertPoint:fromView:
convertRect:fromView:
convertPoint:toView:
convertRect:toView:
```

The `convert...:fromView:` methods convert coordinates from some other view's coordinate system to the local coordinate system (bounds rectangle) of the current view. Conversely, the `convert...:toView:` methods convert coordinates from the current view's local coordinate system (bounds rectangle) to the coordinate system of the specified view. If you specify `nil` as the reference view for any of the methods, the conversions are made to and from the coordinate system of the window that contains the view.

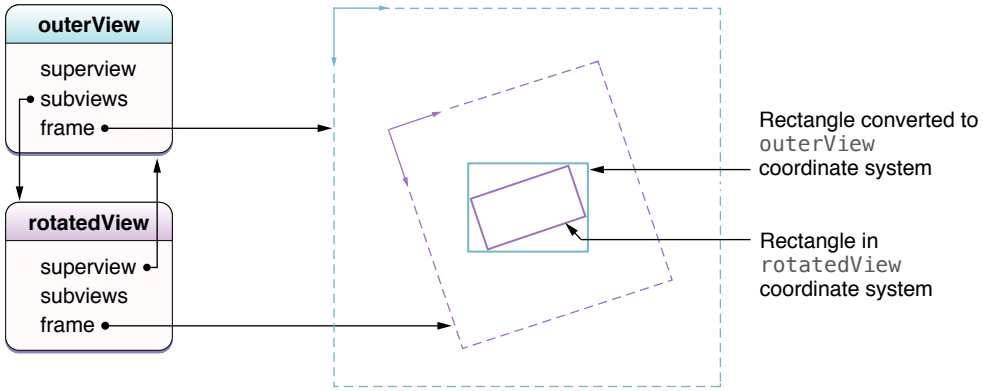
In addition to the `UIView` conversion methods, the `UIWindow` class also defines several conversion methods. These methods are similar to the `UIView` versions except that instead of converting to and from a view's local coordinate system, these methods convert to and from the window's coordinate system.

```
convertPoint:fromWindow:
convertRect:fromWindow:
convertPoint:toWindow:
convertRect:toWindow:
```

When converting coordinates in rotated views, UIKit converts rectangles under the assumption that you want the returned rectangle to reflect the screen area covered by the source rectangle. Figure 3-3 shows an example of how rotations can cause the size of the rectangle to change during a conversion. In the figure, an outer

parent view contains a rotated subview. Converting a rectangle in the subview's coordinate system to the parent's coordinate system yields a rectangle that is physically larger. This larger rectangle is actually the smallest rectangle in the bounds of `outerView` that completely encloses the rotated rectangle.

Figure 3-3 Converting values in a rotated view



Adjusting the Size and Position of Views at Runtime

Whenever the size of a view changes, the size and position of its subviews must change accordingly. The `UIView` class supports both the automatic and manual layout of views in a view hierarchy. With automatic layout, you set the rules that each view should follow when its parent view resizes, and then forget about resizing operations altogether. With manual layout, you manually adjust the size and position of views as needed.

Being Prepared for Layout Changes

Layout changes can occur whenever any of the following events happens in a view:

- The size of a view's bounds rectangle changes.
- An interface orientation change occurs, which usually triggers a change in the root view's bounds rectangle.
- The set of Core Animation sublayers associated with the view's layer changes and requires layout.
- Your application forces layout to occur by calling the `setNeedsLayout` or `layoutIfNeeded` method of a view.
- Your application forces layout by calling the `setNeedsLayout` method of the view's underlying layer object.

Handling Layout Changes Automatically Using Autoresizing Rules

When you change the size of a view, the position and size of any embedded subviews usually needs to change to account for the new size of their parent. The `autoresizesSubviews` property of the superview determines whether the subviews resize at all. If this property is set to `YES`, the view uses the `autoresizingMask` property of each subview to determine how to size and position that subview. Size changes to any subviews trigger similar layout adjustments for their embedded subviews.

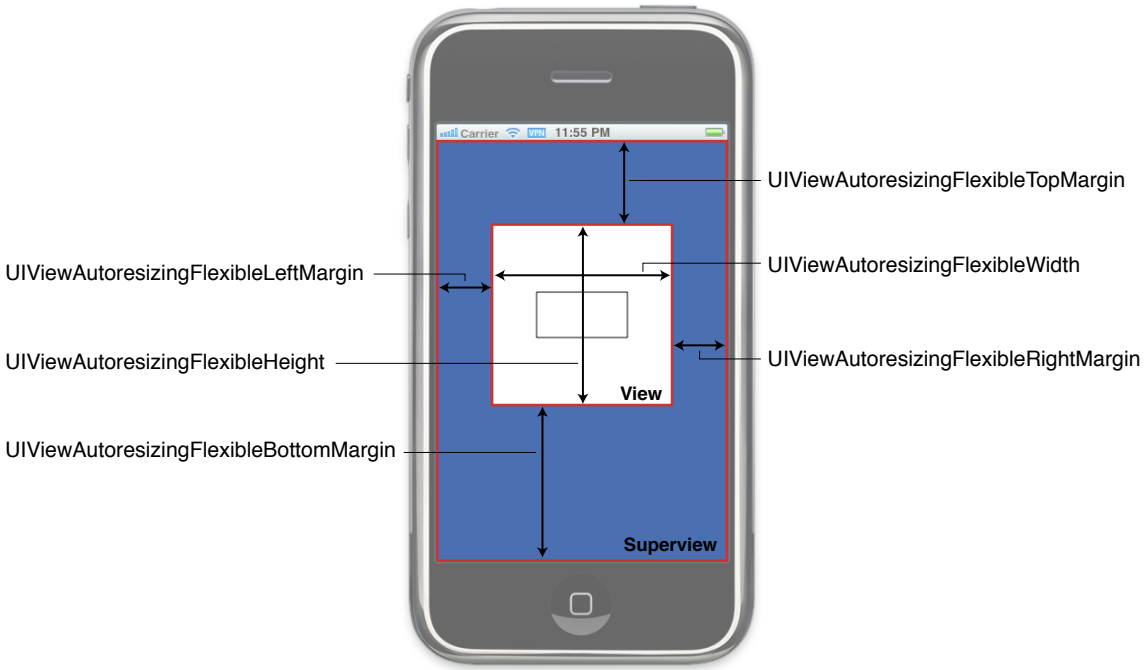
For each view in your view hierarchy, setting that view's `autoresizingMask` property to an appropriate value is an important part of handling automatic layout changes. Table 3-2 lists the autoresizing options you can apply to a given view and describes their effects during layout operations. You can combine constants using an OR operator or just add them together before assigning them to the `autoresizingMask` property. If you are using Interface Builder to assemble your views, you use the Autosizing inspector to set these properties.

Table 3-2 Autoresizing mask constants

Autoresizing mask	Description
`UIViewAutoresizingNone`	The view does not autoresize. (This is the default value.)
`UIViewAutoresizing-FlexibleHeight`	The view's height changes when the superview's height changes. If this constant is not included, the view's height does not change.
`UIViewAutoresizing-FlexibleWidth`	The view's width changes when the superview's width changes. If this constant is not included, the view's width does not change.
`UIViewAutoresizing-FlexibleLeftMargin`	The distance between the view's left edge and the superview's left edge grows or shrinks as needed. If this constant is not included, the view's left edge remains a fixed distance from the left edge of the superview.
`UIViewAutoresizing-FlexibleRightMargin`	The distance between the view's right edge and the superview's right edge grows or shrinks as needed. If this constant is not included, the view's right edge remains a fixed distance from the right edge of the superview.
`UIViewAutoresizing-FlexibleBottomMargin`	The distance between the view's bottom edge and the superview's bottom edge grows or shrinks as needed. If this constant is not included, the view's bottom edge remains a fixed distance from the bottom edge of the superview.
`UIViewAutoresizing-FlexibleTopMargin`	The distance between the view's top edge and the superview's top edge grows or shrinks as needed. If this constant is not included, the view's top edge remains a fixed distance from the top edge of the superview.

Figure 3-4 shows a graphical representation of how the options in the autoresizing mask apply to a view. The presence of a given constant indicates that the specified aspect of the view is flexible and may change when the superview's bounds change. The absence of a constant indicates that the view's layout is fixed in that aspect. When you configure a view that has more than one flexible attribute along a single axis, UIKit distributes any size changes evenly among the corresponding spaces.

Figure 3-4 View autoresizing mask constants



The easiest way to configure autoresizing rules is using the Autosizing controls in the Size inspector of Interface Builder. The flexible width and height constants from the preceding figure have the same behavior as the width and size indicators in the Autosizing controls diagram. However, the behavior and use of margin indicators is effectively reversed. In Interface Builder, the presence of a margin indicator means that the margin has a fixed size and the absence of the indicator means the margin has a flexible size. Fortunately, Interface Builder provides an animation to show you how changes to the autoresizing behaviors affect your view.

> **Important:** If a view's `transform` property does not contain the identity transform, the frame of that view is undefined and so are the results of its autoresizing behaviors.

After the automatic autoresizing rules for all affected views have been applied, UIKit goes back and gives each view a chance to make any necessary manual adjustments to its superview. For more information about how to manage the layout of views manually, see "Tweaking the Layout of Your Views Manually" (page 54).

Tweaking the Layout of Your Views Manually

Whenever the size of a view changes, UIKit applies the autoresizing behaviors of that view's subviews and then calls the `layoutSubviews` method of the view to let it make manual changes. You can implement the `layoutSubviews` method in custom views when the autoresizing behaviors by themselves do not yield the results you want. Your implementation of this method can do any of the following:

- Adjust the size and position of any immediate subviews.
- Add or remove subviews or Core Animation layers.
- Force a subview to be redrawn by calling its `setNeedsDisplay` or `setNeedsDisplayInRect:` method.

One place where applications often lay out subviews manually is when implementing a large scrollable area. Because it is impractical to have a single large view for its scrollable content, applications often implement a root view that contains a number of smaller tile views. Each tile represents a portion of the scrollable content. When a scroll event happens, the root view calls its `setNeedsLayout` method to initiate a layout change. Its `layoutSubviews` method then repositions the tile views based on the amount of scrolling that occurred. As tiles scroll out of the view's visible area, the `layoutSubviews` method moves the tiles to the incoming edge, replacing their contents in the process.

When writing your layout code, be sure to test your code in the following ways:

- Change the orientation of your views to make sure the layout looks correct in all supported interface orientations.
- Make sure your code responds appropriately to changes in the height of the status bar. When a phone call is active, the status bar height increases in size, and when the user ends the call, the status bar decreases in size.

For information about how autoresizing behaviors affect the size and position of your views, see "Handling Layout Changes Automatically Using Autoresizing Rules" (page 52). For an example of how to implement tiling, see the *ScrollViewSuite* sample.

Modifying Views at Runtime

As applications receive input from the user, they adjust their user interface in response to that input. An application might modify its views by rearranging them, changing their size or position, hiding or showing them, or loading an entirely new set of views. In iOS applications, there are several places and ways in which you perform these kinds of actions:

- In a view controller:

    - A view controller has to create its views before showing them. It can load the views from a nib file or create them programmatically. When those views are no longer needed, it disposes of them.
    - When a device changes orientations, a view controller might adjust the size and position of views to match. As part of its adjustment to the new orientation, it might hide some views and show others.
    - When a view controller manages editable content, it might adjust its view hierarchy when moving to and from edit mode. For example, it might add extra buttons and other controls to facilitate editing various aspects of its content. This might also require the resizing of any existing views to accommodate the extra controls.
- In animation blocks:
    - When you want to transition between different sets of views in your user interface, you hide some views and show others from inside an animation block.
    - When implementing special effects, you might use an animation block to modify various properties of the view. For example, to animate changes to the size of a view, you would change the size of its frame rectangle.
- Other ways:
    - When touch events or gestures occur, your interface might respond by loading a new set of views or changing the current set of views. For information about handling events, see *Event Handling Guide for iOS*.
    - When the user interacts with a scroll view, a large scrollable area might hide and show tile subviews. For more information about supporting scrollable content, see *Scroll View Programming Guide for iOS*.
    - When the keyboard appears, you might reposition or resize views so that they do not lie underneath the keyboard. For information about how to interact with the keyboard, see *Text Programming Guide for iOS*.

View controllers are a common place to initiate changes to your views. Because a view controller manages the view hierarchy associated with the content being displayed, it is ultimately responsible for everything that happens to those views. When loading its views or handling orientation changes, the view controller can add new views, hide or replace existing ones, and make any number of changes to make the views ready for the display. And if you implement support for editing your view's content, the `setEditing:animated:` method in `UIViewController` gives you a place to transition your views to and from their editable versions.

Animation blocks are another common place to initiate view-related changes. The animation support built into the `UIView` class makes it easy to animate changes to view properties. You can also use the `transitionWithView:duration:options:animations:completion:` or `transitionFromView:toView:duration:options:completion:` methods to swap out entire sets of views for new ones.

For more information about animating views and initiating view transitions, see "Animations" (page 64). For more information on how you use view controllers to manage view-related behaviors, see *View Controller Programming Guide for iOS*.

Interacting with Core Animation Layers

Each view object has a dedicated Core Animation layer that manages the presentation and animation of the view's content on the screen. Although you can do a lot with your view objects, you can also work directly with the corresponding layer objects as needed. The layer object for the view is stored in the view's `layer` property.

Changing the Layer Class Associated with a View

The type of layer associated with a view cannot be changed after the view is created. Therefore, each view uses the `layerClass` class method to specify the class of its layer object. The default implementation of this method returns the `CALayer` class and the only way to change this value is to subclass, override the method, and return a different value. You can change this value to use a different kind of layer. For example, if your view uses tiling to display a large scrollable area, you might want to use the `CATiledLayer` class to back your view.

Implementation of the `layerClass` method should simply create the desired `Class` object and return it. For example, a view that uses tiling would have the following implementation for this method:

```
+ (Class)layerClass
{
    return [CATiledLayer class];
}
```

Each view calls its `layerClass` method early in its initialization process and uses the returned class to create its layer object. In addition, the view always assigns itself as the delegate of its layer object. At this point, the view owns its layer and the relationship between the view and layer must not change. You must also not assign the same view as the delegate of any other layer object. Changing the ownership or delegate relationships of the view will cause drawing problems and potential crashes in your application.

For more information about the different types of layer objects provided by Core Animation, see *Core Animation Reference Collection*.

Embedding Layer Objects in a View

If you prefer to work primarily with layer objects instead of views, you can incorporate custom layer objects into your view hierarchy as needed. A custom layer object is any instance of `CALayer` that is not owned by a view. You typically create custom layers programmatically and incorporate them using Core Animation routines. Custom layers do not receive events or participate in the responder chain but do draw themselves and respond to size changes in their parent view or layer according to the Core Animation rules.

Listing 3-3 shows an example of the `viewDidLoad` method from a view controller that creates a custom layer object and adds it to its root view. The layer is used to display a static image that is animated. Instead of adding the layer to the view itself, you add it to the view's underlying layer.

Listing 3-3 Adding a custom layer to a view

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Create the layer.
    CALayer* myLayer = [[CALayer alloc] init];

    // Set the contents of the layer to a fixed image. And set
    // the size of the layer to match the image size.
    UIImage layerContents = [[UIImage imageNamed:@"myImage"] retain];
    CGSize imageSize = layerContents.size;

    myLayer.bounds = CGRectMake(0, 0, imageSize.width, imageSize.height);
    myLayer = layerContents.CGImage;

    // Add the layer to the view.
    CALayer*    viewLayer = self.view.layer;
    [viewLayer addSublayer:myLayer];

    // Center the layer in the view.
    CGRect        viewBounds = backingView.bounds;
    myLayer.position = CGPointMake(CGRectGetMidX(viewBounds),
CGRectGetMidY(viewBounds));

    // Release the layer, since it is retained by the view's layer
```

```
    [myLayer release];
}
```

You can add any number of sublayers and arrange them into sublayer hierarchies, if you want. However, at some point, those layers must be attached to the layer object of a view.

For information on how to work with layers directly, see *Core Animation Programming Guide*.

Defining a Custom View

If the standard system views do not do exactly what you need, you can define a custom view. Custom views give you total control over the appearance of your application's content and how interactions with that content are handled.

> **Note:** If you are using OpenGL ES to do your drawing, you should use the `GLKView` class instead of subclassing `UIView`. For more information about how to draw using OpenGL ES, see *OpenGL ES Programming Guide for iOS*.

Checklist for Implementing a Custom View

The job of a custom view is to present content and manage interactions with that content. The successful implementation of a custom view involves more than just drawing and handling events, though. The following checklist includes the more important methods you can override (and behaviors you can provide) when implementing a custom view:

- Define the appropriate initialization methods for your view:
  - For views you plan to create programmatically, override the `initWithFrame:` method or define a custom initialization method.
  - For views you plan to load from nib files, override the `initWithCoder:` method. Use this method to initialize your view and put it into a known state.
- Implement a `dealloc` method to handle the cleanup of any custom data.
- To handle any custom drawing, override the `drawRect:` method and do your drawing there.
- Set the `autoresizingMask` property of the view to define its autoresizing behavior.
- If your view class manages one or more integral subviews, do the following:
  - Create those subviews during your view's initialization sequence.
  - Set the `autoresizingMask` property of each subview at creation time.

    - If your subviews require custom layout, override the `layoutSubviews` method and implement your layout code there.
- To handle touch-based events, do the following:
    - Attach any suitable gesture recognizers to the view by using the `addGestureRecognizer:` method.
    - For situations where you want to process the touches yourself, override the `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:` methods. (Remember that you should always override the `touchesCancelled:withEvent:` method, regardless of which other touch-related methods you override.)
- If you want the printed version of your view to look different from the onscreen version, implement the `drawRect:forViewPrintFormatter:` method. For detailed information about how to support printing in your views, see *Drawing and Printing Guide for iOS*.

In addition to overriding methods, remember that there is a lot you can do with the view's existing properties and methods. For example, the `contentMode` and `contentStretch` properties let you change the final rendered appearance of your view and might be preferable to redrawing the content yourself. In addition to the `UIView` class itself, there are many aspects of a view's underlying `CALayer` object that you can configure directly or indirectly. You can even change the class of the layer object itself.

For more information about the methods and properties of the view class, see *UIView Class Reference*.

Initializing Your Custom View

Every new view object you define should include a custom `initWithFrame:` initializer method. This method is responsible for initializing the class at creation time and putting your view object into a known state. You use this method when creating instances of your view programmatically in your code.

Listing 3-4 shows a skeletal implementation of a standard `initWithFrame:` method. This method calls the inherited implementation of the method first and then initializes the instance variables and state information of the class before returning the initialized object. Calling the inherited implementation is traditionally performed first so that if there is a problem, you can abort your own initialization code and return `nil`.

Listing 3-4 Initializing a view subclass

```
- (id)initWithFrame:(CGRect)aRect {
    self = [super initWithFrame:aRect];
    if (self) {
          // setup the initial properties of the view
          ...
```

```
        }
    return self;
}
```

If you plan to load instances of your custom view class from a nib file, you should be aware that in iOS, the nib-loading code does not use the `initWithFrame:` method to instantiate new view objects. Instead, it uses the `initWithCoder:` method that is part of the `NSCoding` protocol.

Even if your view adopts the `NSCoding` protocol, Interface Builder does not know about your view's custom properties and therefore does not encode those properties into the nib file. As a result, your own `initWithCoder:` method should perform whatever initialization code it can to put the view into a known state. You can also implement the `awakeFromNib` method in your view class and use that method to perform additional initialization.

Implementing Your Drawing Code

For views that need to do custom drawing, you need to override the `drawRect:` method and do your drawing there. Custom drawing is recommended only as a last resort. In general, if you can use other views to present your content, that is preferred.

The implementation of your `drawRect:` method should do exactly one thing: draw your content. This method is not the place to be updating your application's data structures or performing any tasks not related to drawing. It should configure the drawing environment, draw your content, and exit as quickly as possible. And if your `drawRect:` method might be called frequently, you should do everything you can to optimize your drawing code and draw as little as possible each time the method is called.

Before calling your view's `drawRect:` method, UIKit configures the basic drawing environment for your view. Specifically, it creates a graphics context and adjusts the coordinate system and clipping region to match the coordinate system and visible bounds of your view. Thus, by the time your `drawRect:` method is called, you can begin drawing your content using native drawing technologies such as UIKit and Core Graphics. You can get a pointer to the current graphics context using the `UIGraphicsGetCurrentContext` function.

Important: The current graphics context is valid only for the duration of one call to your view's `drawRect:` method. UIKit might create a different graphics context for each subsequent call to this method, so you should not try to cache the object and use it later.

Listing 3-5 shows a simple implementation of a `drawRect:` method that draws a 10-pixel-wide red border around the view. Because UIKit drawing operations use Core Graphics for their underlying implementations, you can mix drawing calls, as shown here, to get the results you expect.

Listing 3-5 A drawing method

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGRect    myFrame = self.bounds;

    // Set the line width to 10 and inset the rectangle by
    // 5 pixels on all sides to compensate for the wider line.
    CGContextSetLineWidth(context, 10);
    CGRectInset(myFrame, 5, 5);

    [[UIColor redColor] set];
    UIRectFrame(myFrame);
}
```

If you know that your view's drawing code always covers the entire surface of the view with opaque content, you can improve system performance by setting the `opaque` property of your view to `YES`. When you mark a view as opaque, UIKit avoids drawing content that is located immediately behind your view. This not only reduces the amount of time spent drawing but also minimizes the work that must be done to composite your view with other content. However, you should set this property to `YES` only if you know your view's content is completely opaque. If your view cannot guarantee that its contents are always opaque, you should set the property to `NO`.

Another way to improve drawing performance, especially during scrolling, is to set the `clearsContextBeforeDrawing` property of your view to `NO`. When this property is set to `YES`, UIKIt automatically fills the area to be updated by your `drawRect:` method with transparent black before calling your method. Setting this property to `NO` eliminates the overhead for that fill operation but puts the burden on your application to fill the update rectangle passed to your `drawRect:` method with content.

Responding to Events

View objects are responder objects—instances of the `UIResponder` class—and are therefore capable of receiving touch events. When a touch event occurs, the window dispatches the corresponding event object to the view in which the touch occurred. If your view is not interested in an event, it can ignore it or pass it up the responder chain to be handled by a different object.

In addition to handling touch events directly, views can also use gesture recognizers to detect taps, swipes, pinches, and other types of common touch-related gestures. Gesture recognizers do the hard work of tracking touch events and making sure that they follow the right criteria to qualify them as the target gesture. Instead

of your application having to track touch events, you can create the gesture recognizer, assign an appropriate target object and action method to it, and install it on your view using the `addGestureRecognizer:` method. The gesture recognizer then calls your action method when the corresponding gesture occurs.

If you prefer to handle touch events directly, you can implement the following methods for your view, which are described in more detail in *Event Handling Guide for iOS*:

```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
touchesCancelled:withEvent:
```

The default behavior for views is to respond to only one touch at a time. If the user puts a second finger down, the system ignores the touch event and does not report it to your view. If you plan to track multi-finger gestures from your view's event-handler methods, you need to enable multitouch events by setting the `multipleTouchEnabled` property of your view to `YES`.

Some views, such as labels and images, disable event handling altogether initially. You can control whether a view is able to receive touch events by changing the value of the view's `userInteractionEnabled` property. You might temporarily set this property to `NO` to prevent the user from manipulating the contents of your view while a long operation is pending. To prevent events from reaching any of your views, you can also use the `beginIgnoringInteractionEvents` and `endIgnoringInteractionEvents` methods of the `UIApplication` object. These methods affect the delivery of events for the entire application, not just for a single view.

> **Note:** The animation methods of `UIView` typically disable touch events while animations are in progress. You can override this behavior by configuring the animation appropriately. For more information about performing animations, see "Animations" (page 64).

As it handles touch events, UIKit uses the `hitTest:withEvent:` and `pointInside:withEvent:` methods of `UIView` to determine whether a touch event occurred inside a given view's bounds. Although you rarely need to override these methods, you could do so to implement custom touch behaviors for your view. For example, you could override these methods to prevent subviews from handling touch events.

Cleaning Up After Your View

If your view class allocates any memory, stores references to any custom objects, or holds resources that must be released when the view is released, you must implement a `dealloc` method. The system calls the `dealloc` method when your view's retain count reaches zero and it is time to deallocate the view. Your implementation of this method should release any objects or resources held by the view and then call the inherited implementation, as shown in Listing 3-6. You should not use this method to perform any other types of tasks.

Listing 3-6 Implementing the `dealloc` method

```
- (void)dealloc {
    // Release a retained UIColor object
    [color release];

    // Call the inherited implementation
    [super dealloc];
}
```

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" (page 66).

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.

Property	Changes you can make
`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" (page 74).

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" (page 66).

Listing 4-3 shows the code needed to implement the same behavior as Listing 4-1 (page 66) 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.

Method	Usage
`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:` `setAnimationRepeat-Autoreverses:`	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" (page 73).

`setAnimationDelegate:`
`setAnimationWill-StartSelector:`
`setAnimationDid-StopSelector:`
	Use these methods to execute code immediately before or after the animations. For more information about using a delegate, see "Configuring an Animation Delegate" (page 71).

`setAnimationBegins-FromCurrentState:`	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 (page 67) 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 (page 75), you would need to configure an animation delegate with a did-stop handler as described in "Configuring an Animation Delegate" (page 71).

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" (page 71).

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" (page 72).

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 (page 78), 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 (page 78) 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*.

Document Revision History

This table describes the changes to *View Programming Guide for iOS*.

Date	Notes
2013-10-22	Updated advice about views that draw using OpenGL ES.
2011-03-08	Reorganized and expanded the content of the document. Added information on how to create view-based animations. Incorporated information on how to display content on an external display. Added information about how to work with high-resolution screens.
2010-05-17	New document describing the creation and management of views, windows, and other visual interface elements.

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