Scroll View Programming Guide for iOS

Developer

Contents

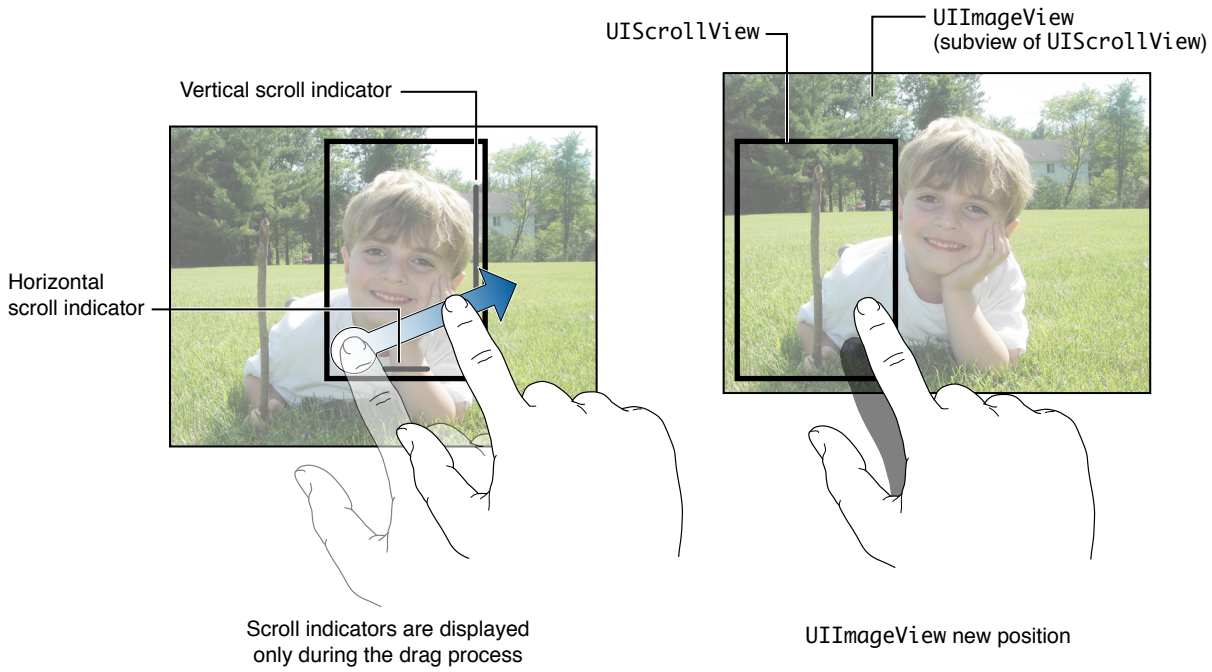
Figures and Listings

About Scroll View Programming

Scroll views are found in iOS applications when content that needs to be displayed and manipulated won't fit entirely on the screen. Scroll views have two main purposes:

- To let users drag the area of the content they want to display
- To let users zoom into or out of the displayed content using the pinch gestures

The following figure shows a typical use of a `UIScrollView` class. The subview is a `UIImageView` containing the image of a boy. When the user drags his or her finger on the screen, the viewport onto the image moves and, as you can see in the diagram, the scroll indicators are shown. When the user lifts the finger, the indicators disappear.



At a Glance

The `UIScrollView` class provides the following functionality:

- Scrolling content that will not fit entirely on the screen

- Zooming content, allowing your application to support the standard pinch gestures to zoom in and out
- Restricting scrolling to a single screen of content at a time (paging mode)

The `UIScrollView` class contains no specially defined view for the content that it displays; instead it simply scrolls its subviews. This simple model is possible because scroll views on iOS have no additional controls for initiating scrolling.

Basic View Scrolling Is the Easiest to Implement

Scrolling via drag or flick gestures requires no subclassing or delegation. With the exception of setting the content size of the `UIScrollView` instance programmatically, the entire interface can be created and designed in Interface Builder.

Related Chapters:: "Creating and Configuring Scroll Views" (page 8)

To Support Pinch Zoom Gestures, You Use Delegation

Adding basic pinch-in and pinch-out zoom support requires that the scroll view use delegation. The delegate class must conform to the `UIScrollViewDelegate` protocol and implement a delegate method that specifies which of the scroll view's subviews should be zoomed. You must also specify one, or both, of the minimum and maximum magnification factors.

If your application needs to support double-tap to zoom, two-finger touch to zoom out, and simple single touch scrolling and panning (in addition to the standard pinch gestures) you'll need to implement code in your content view to handle this functionality.

Related Chapters:: "Basic Zooming Using the Pinch Gestures" (page 22).

To Support Pinch to Zoom and Tap to Zoom, You Implement Code in the Content View

If your application needs to support double-tap to zoom, two-finger touch to zoom out, and simple single-touch scrolling and panning (in addition to the standard pinch gestures), you implement code in your content view.

Related Chapters:: "Zooming by Tapping" (page 29).

To Support Paging Mode, You Need Only Three Subviews

To support paging mode, no subclassing or delegation is required. You simply specify the content size and enable paging mode. You can implement most paging applications using only three subviews, thereby saving memory space and increasing performance.

Related Notes:: "Scrolling Using Paging Mode" (page 35)

Prerequisites

Before reading this guide, read *iOS App Programming Guide* to understand the basic process for developing iOS applications. Also consider reading *View Controller Programming Guide for iOS* for general information about view controllers, which are frequently used in conjunction with scroll views.

How to Use This Document

The remaining chapters in this guide take you through increasingly complex tasks such as handling tap-to-zoom techniques, understanding the role of the delegate and its messaging sequence, and nesting scroll views in your application.

See Also

You will find the following sample-code projects to be instructive models for your own table view implementations:

- *Scrolling* demonstrates basic scrolling.
- *PageControl* demonstrates using scroll views in paging mode.
- *ScrollViewSuite* of sample projects. These are advanced examples that demonstrate the tap-to-scroll techniques as well as other significantly advanced projects, including tiling to allow large, detailed images, to be displayed in a memory efficient manner.

Creating and Configuring Scroll Views

Scroll views are created as any other view is, either programmatically or in Interface Builder. Only a small amount of additional configuration is required to achieve basic scrolling capabilities.

Creating Scroll Views

A scroll view is created and inserted into a controller or view hierarchy like any other view. There are only two additional steps required to complete the scroll view configuration:

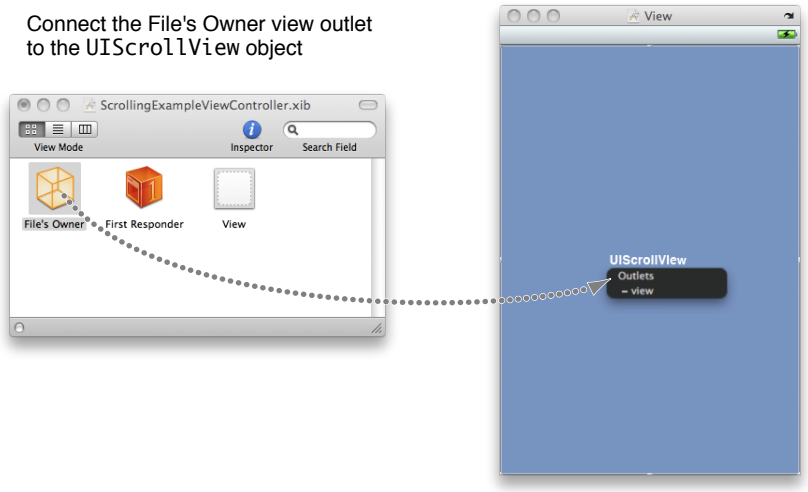
1. You must set the `contentSize` property to the size of the scrollable content. This specifies the size of the scrollable area.
2. You must also add a view or views that are displayed and scrolled by the scroll view. These view(s) provide the displayed content.

You can optionally configure any visual cues your application requires—vertical and horizontal scroll indicators, drag bouncing, zoom bouncing, and directional constraint of scrolling

Creating Scroll Views in Interface Builder

To create a scroll view in Interface Builder, you drag the `UIScrollView` icon located in the **Library->Cocoa Touch->Data Views** section of the Library palette into the view 'window'. You then connect the `UIViewController` subclass's view outlet to the scroll view. Figure 1-1 shows the connection, assuming that the File's Owner is the `UIViewController` subclass (a common design pattern).

Figure 1-1 How a `UIViewController` subclass connects to a scroll view



Even though the `UIScrollView` inspector in Interface Builder allows you to set many of the properties of the scroll view instance, you are still responsible for setting the `contentSize` property, which defines the size of the scrollable area, in your application code. If you've connected the scroll view to the `view` property of a controller instance (typically the File's Owner), initializing the `contentSize` property occurs in the controller's `viewDidLoad` method shown in Listing 1-1.

Listing 1-1 Setting a scroll view's size

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIScrollView *tempScrollView=(UIScrollView *)self.view;
    tempScrollView.contentSize=CGSizeMake(1280,960);
}
```

After the scroll view's size has been configured, your application can then add the required subview(s) that provide the view content, either programmatically or by inserting them into the scroll view in Interface Builder.

Creating Scroll Views Programmatically

It's also possible to create a scroll view entirely in code. This is typically done in your controller class, specifically, in the implementation of the `loadView` method. A sample implementation is shown in Listing 1-2.

Listing 1-2 Creating s scroll view programmatically

```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    scrollView.contentSize=CGSizeMake(320,758);

    // do any further configuration to the scroll view
    // add a view, or views, as a subview of the scroll view.

    // release scrollView as self.view retains it
    self.view=scrollView;
    [scrollView release];
}
```

This code creates a scroll view that is the size of the full screen (less the status bar), sets the `scrollView` object as the controller's view, and sets the `contentSize` property to 320 x 758 pixels. This code creates a scroll view that will scroll vertically.

There would be more code in this method implementation, for example, code that would insert the subview or views and configure those as required. Also, this code assumes that the controller doesn't have a `view` set already. If it did, you would be responsible for releasing the existing view before setting the scroll view as the controller's view.

Adding Subviews

After you have created and configured the scroll view, you must add a subview or subviews to display the content. Whether you should use a single subview or multiple subviews directly in your scroll view is a design decision that is usually based on one requirement: Does your scroll view need to support zooming?

If you intend to support zoom in your scroll view, the most common technique is to use a single subview that encompasses the entire `contentSize` of the scroll view and then add additional subviews to that view. This allows you to specify the single 'collection' content view as the view to zoom, and all its subviews will zoom according to its state.

If zooming is not a requirement, then whether your scroll view uses a single subview (with or without its own subviews) or multiple subviews is an application dependent decision.

Note: While returning a single subview is the most common case, your application might require the ability to allow multiple views within the same scroll view to support zooming. In that case, you would return the appropriate subview using the delegate method `viewForZoomingInScrollView:`, discussed further in "Basic Zooming Using the Pinch Gestures" (page 22).

Configuring The Scroll View Content Size, Content Inset, And Scroll Indicators

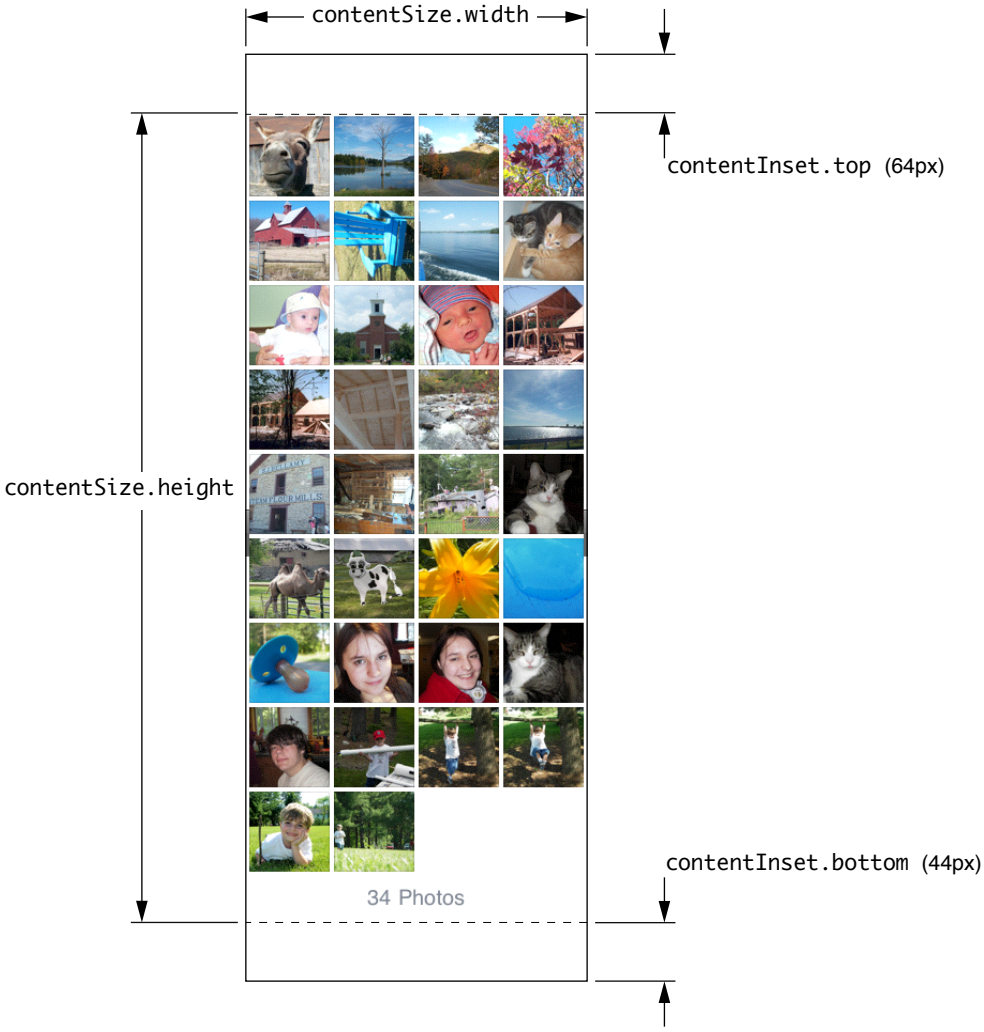
The `contentSize` property is the size of the content that you need to display in the scroll view. In "Creating Scroll Views in Interface Builder," it is set to 320 wide by 758 pixels high. The image in Figure 1-2 shows the content of the scroll view with the `contentSize` width and height indicated.

Figure 1-2 Content with `contentSize` dimensions labeled

You may want to add padding around the edges of the scroll view content, typically at the top and bottom so controllers and toolbars don't interfere with the content. To add padding, use the `contentInset` property to specify a buffer area around the content of the scroll view. One way of thinking of it is that it makes the scroll view content area larger without changing the size of the subview or the size of the view's content.

The `contentInset` property is a `UIEdgeInsets` struct with the fields `top`, `bottom`, `left`, `right`. Figure 1-3 shows the content with the `contentInset` and `contentSize` indicated.

Figure 1-3 Content with `contentSize` and `contentInset` indicated



As shown in Figure 1-3 (page 13) specifying `(64,44,0,0)` for the `contentInset` property results in an additional buffer area that is 64 pixels at the top of the content (20 pixels for the status bar and 44 pixels for the navigation controller) and 44 pixels at the bottom (the height of the toolbar). Setting `contentInset` to these values allows displaying the navigation control and toolbar on screen, yet still allows scrolling to display the entire content of the scroll view.

Listing 1-3 Setting the `contentInset` property

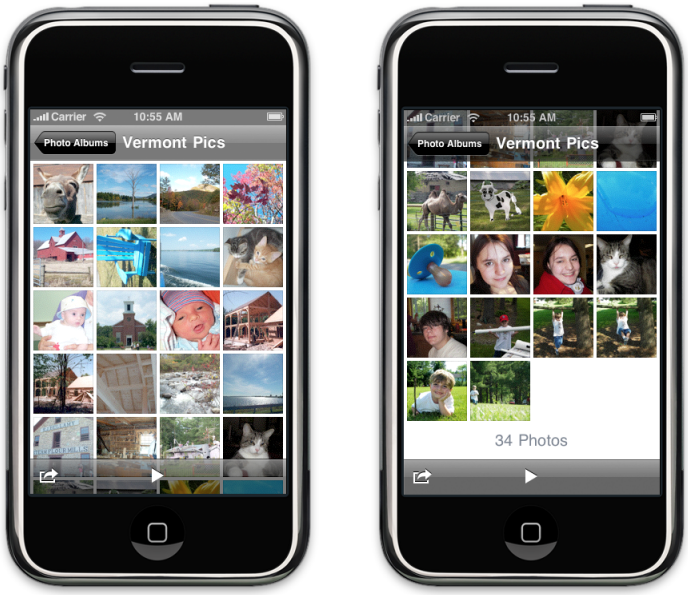
```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    self.view=scrollView;
    scrollView.contentSize=CGSizeMake(320,758);
    scrollView.contentInset=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);

    // do any further configuration to the scroll view
    // add a view, or views, as a subview of the scroll view.

    // release scrollView as self.view retains it
    self.view=scrollView;
    [scrollView release];
}
```

Figure 1-4 shows the results of setting the `contentInset` top and bottom parameters to those values. When scrolled to the top (as shown on the left), the screen leaves space for the navigation bar and the status bar. The image on the right shows the content scrolled to the bottom with space for the toolbar. In both cases you can see the content through the transparent navigation bar and toolbar when scrolled, yet when the content is scrolled fully to the top or bottom, all content is visible.

Figure 1-4 The results of setting values for the `contentInset` top and bottom



Scrolled to top **Scrolled to bottom**

However, changing the `contentInset` value has an unexpected side effect when your scroll view displays scroll indicators. As the user drags the content to the top or bottom of the screen, the scroll indicator scrolls over any content displayed in the areas that are within the area defined by `contentInset` for example, in the navigation control and toolbar.

To correct this, you must set the `scrollIndicatorInsets` property. As with the `contentInset` property, the `scrollIndicatorInsets` property is defined as a `UIEdgeInsets` struct. Setting the vertical inset values restricts the vertical scroll indicators from being displayed beyond that inset and this also results in the horizontal scroll indicators being displayed outside the `contentInset` rect area.

Altering the `contentInset` without also setting the `scrollIndicatorInsets` property allows the scroll indicators to be drawn over the navigation controller and the toolbar, an unwanted result. However, setting the `scrollIndicatorInsets` values to match the `contentInset` value remedies this situation.

The corrected `loadView` implementation in Listing 1-4 shows the additional code required to configure the scroll view by adding the `scrollIndicatorInsets` initialization.

Listing 1-4 Setting the scroll view `contentInset` and `scrollIndicatorInsets` properties

```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    scrollView.contentSize=CGSizeMake(320,758);
    scrollView.contentInset=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);
    scrollView.scrollIndicatorInsets=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);

    // do any further configuration to the scroll view
    // add a view, or views, as a subview of the scroll view.

    // release scrollView as self.view retains it
    self.view=scrollView;
    [scrollView release];
}
```

Scrolling the Scroll View Content

The most common method of initiating the scrolling of a scroll view is a direct manipulation by the user touching the screen and dragging with his or her finger. The scroll content then scrolls in response to the action. This gesture is referred to as a *drag gesture*.

A variation of the drag gesture is the *flick gesture*. A flick gesture is a quick movement of a user's finger that makes initial contact with the screen, drags in the direction of the desired scroll, and then lifts from the screen. This gesture not only causes scrolling, it imparts a momentum, based on the speed of the user's dragging action, that causes scrolling to continue even after the gesture is completed. The scrolling then decelerates over a specified period of time. The flick gesture allows users to move large distances with a single action. At any time during the deceleration, the user can touch the screen to stop the scrolling in place. All this behavior is built into `UIScrollView` and requires no implementation on the part of the developer.

But sometimes it's necessary for the application to scroll content programmatically, for example, to display a specific portion of a document. In those cases, `UIScrollView` provides the required methods.

The `UIScrollView` delegate protocol `UIScrollViewDelegate` provides methods that allow your application to track the scrolling progress and respond as appropriate to the specific needs of your application.

Scrolling Programmatically

Scrolling the content of a scroll view is not always in response to the user's finger dragging or flicking the screen. There are times when your application will need to scroll to a certain content offset, so that a specific rectangular area is exposed, or to the top of the scroll view. `UIScrollView` provides methods to perform all these actions.

Scrolling to a Specific Offset

Scrolling to a specific top-left location (the `contentOffset` property) can be accomplished in two ways. The `setContentOffset:animated:` method scrolls the content to the specified content offset. If the animated parameter is `YES`, the scrolling will animate from the current position to the specified position at a constant rate. If the animated parameter is `NO`, the scrolling is immediate and no animation takes place. In both cases, the delegate is sent a `scrollViewDidScroll:` message. If animation is disabled, or if you set the content offset by setting the `contentOffset` property directly, the delegate receives a single `scrollViewDidScroll:`

message. If animation is enabled, then the delegate receives a series of `scrollViewDidScroll:` messages as the animation is in progress. When the animation is complete, the delegate receives a `scrollViewDidEndScrollingAnimation:` message.

Making a rectangle visible

It is also possible to scroll a rectangular area so that it is visible. This is especially useful when an application needs to display a control that is currently outside the visible area into the visible view. The `scrollRectToVisible:animated:` method scrolls the specified rectangle so that it is just visible inside the scroll view. If the animated parameter is `YES`, the rectangle is scrolled into view at a constant pace. As with `setContentOffset:animated:`, if animation is disabled, the delegate is sent a single `scrollViewDidScroll:` message. If animation is enabled, the delegate is sent a series of `scrollViewDidScroll:` messages as animation progresses. In the case of `scrollRectToVisible:animated:` the scroll view's tracking and dragging properties are also `NO`.

If animation is enabled for `scrollRectToVisible:animated:`, the delegate receives a `scrollViewDidEndScrollingAnimation:` message, providing notification that the scroll view has arrived at the specified location and animation is complete.

Scroll To Top

If the status bar is visible a scroll view can also scroll to the top of the content in response to a tap on the status bar. This practice is common in applications that provide a vertical representation of data. For example, the Photos application supports scroll to top, both in the album selection table view and when viewing thumbnails of the photos in an album and most `UITableView` implementations (a subclass of `UIScrollView`) also support scroll to top..

Your application enables this behavior by implementing the delegate method the scroll view property `scrollViewShouldScrollToTop:` and return `YES`. This delegate method allows fine-grained control over which scroll view will scroll to the top if there are multiple scroll views on the screen at one time by returning the scroll view to scroll.

When scrolling is complete, the delegate is sent a `scrollViewDidScrollToTop:` message, specifying the scroll view.

Delegate Messages Sent During Scrolling

As scrolling occurs, the scroll view tracks state using the `tracking`, `dragging`, `decelerating`, and `zooming` properties. In addition, the `contentOffset` property defines the point in the content that is visible at the top left of the scroll view's bounds.. The following table describes each of the state properties:

State property	Description
`tracking`	`YES` if the user's finger is in contact with the device screen.
`dragging`	`YES` if the user's finger is in contact with the device screen and has moved.
`decelerating`	`YES` if the scroll view is decelerating as a result of a flick gesture, or a bounce from dragging beyond the scroll view frame.
`zooming`	`YES` if the scroll view is tracking a pinch gesture to change its `zoomScale` property..
`contentOffset`	A `CGPoint` value that defines the top-left corner of the scroll view bounds.

It isn't necessary to poll these properties to determine the action in progress because the scroll view sends a detailed sequence of messages to the delegate, indicating the progress of the scrolling action. These methods allow your application to respond as necessary. The delegate methods can query those state properties to determine why the message was received or where the scroll view currently is.

The Simple Approach: Tracking The Start and Completion Of A Scroll Action

If your application is interested only in the beginning and ending of the scrolling process, you can implement only a small subset of the possible delegate methods.

Implement the `scrollViewWillBeginDragging:` method to receive notification that dragging will begin.

To determine when scrolling is complete you must implement two delegate methods: `scrollViewDidEndDragging:willDecelerate:` and `scrollViewDidEndDecelerating:`. Scrolling is completed either when the delegate receives the `scrollViewDidEndDragging:willDecelerate:` message with `NO` as the decelerate parameter, or when your delegate receives the `scrollViewDidEndDecelerating:` method. In either case, scrolling is complete.

The Complete Delegate Message Sequence

When the user touches the screen the tracking sequence begins. The `tracking` property is set to `YES` immediately, and remains `YES` as long as the user's finger is in contact with the screen, regardless of whether they are moving their finger.

If the user's finger remains stationary and the content view responds to touch events, it should handle the touch, and the sequence is complete.

However, if the user moves the finger, the sequence continues.

When the user begins to move his or her finger to initiate scrolling the scroll view first attempts (assuming the default values of the scroll view) to cancel any in progress touch handling, if it is attempting to do so.

> **Note:** Throughout the message sequence, it is possible that the `tracking` and `dragging` properties will always remain `NO` and the `zooming` property will be `YES`. This happens when scrolling occurs as a result of a zoom action, whether scrolling was initiated by a gesture or programatically. Your application may choose to take a different action if the delegate methods are sent as a result of zooming or scrolling.

The scroll view's `dragging` property is set to `YES`, and its delegate is sent the `scrollViewWillBeginDragging:` message.

As the user drags his or her finger, the `scrollViewDidScroll:` message is sent to the delegate. This message is sent continuously as scrolling continues. Your implementation of this method can query the scroll view's `contentOffset` property to determine the location of the top-left corner of the scroll view bounds. The `contentOffset` property is always the current location of the top-left corner of the scroll bounds, whether scrolling is in progress or not.

If the user performs a flick gesture, the `tracking` property is set to `NO`, because in order to perform a flick gesture, the user's finger breaks contact with the screen after the initial gesture begins the scroll. At this time the delegate receives a `scrollViewDidEndDragging:willDecelerate:` message. The deceleration parameter will be `YES` as the scrolling decelerates. The deceleration speed is controlled by the the `decelerationRate` property. By default, this property is set to `UIScrollViewDecelerationRateNormal`, which allows scrolling to continue for a considerable time. You can set the rate to `UIScrollViewDecelerationFast` to cause the deceleration to take significantly less time, the scrolled distance after the flick gesture to be much shorter. As a view decelerates, the `decelerating` property of the scroll view is `YES`.

If the user drags, stops dragging and then lifts their finger from the screen the delegate receives the `scrollViewDidEndDragging:willDecelerate:` message, however the deceleration parameter is `NO`. This is because no momentum has been imparted on the scroll view. Because the user's finger is no longer in contact with the screen the `tracking` property value is `NO`.

If the decelerate parameter of the `scrollViewDidEndDragging:willDecelerate:` message is `NO`, then the scroll view delegate will receive no more delegate messages for this drag action. The scroll view decelerating property also now returns a value of `NO`.

There is one other condition that causes the `scrollViewDidEndDragging:willDecelerate:` message to be sent to the delegate, even if the user has lifted his or her finger when is stationary. If the scroll view is configured to provide the visual cue of bouncing when the user drags the content past the edge of the scrolling area, the `scrollViewDidEndDragging:willDecelerate:` message is sent to the delegate with `YES` as the decelerate parameter. Bouncing is enabled when the `bounces` property is `YES` (the default state). The `alwaysBounceVertical` and `alwaysBounceHorizontal` properties do not affect the behavior of the scroll view when `bounces` is `NO`. If `bounces` is `YES`, they allow bouncing when the `contentSize` property value is smaller than the scroll view bounds.

Regardless of the condition that caused the scroll view to receive the `scrollViewDidEndDragging:willDecelerate:` message, if the decelerate parameter is `YES`, the scroll view is sent the `scrollViewWillBeginDecelerating:` message. During deceleration, the delegate continues to receive the `scrollViewDidScroll:` message, although the `tracking` and `dragging` values are now both `NO`. The `decelerating` property continues to be `YES`.

Finally, when the scroll view deceleration completes, the delegate is sent a `scrollViewDidEndDecelerating:` message, the `decelerating` property has a value of `NO`, and the scrolling sequence is complete.

Basic Zooming Using the Pinch Gestures

`UIScrollView` makes supporting the pinch gestures for zooming easy. Your application specifies the zoom factors (that is how much bigger or smaller you can make the content) and you implement a single delegate method. Taking those few steps allows your scroll view to support the pinch gestures for zooming.

Supporting the Pinch Zoom Gestures

The pinch-in and pinch-out zoom gestures are standard gestures that iOS application users expect to use when zooming in and out. Figure 3-1 shows examples of the pinch gestures.

Figure 3-1 The standard pinch-in and pinch-out gestures

To support zooming, you must set a delegate for your scroll view. The delegate object must conform to the `UIScrollViewDelegate` protocol. In many cases, the delegate will be the scroll view's controller class. That delegate class must implement the `viewForZoomingInScrollView:` method and return the view to zoom. The implementation of the delegate method shown below returns the value of the controller's `imageView` property, which is an instance of `UIImageView`. This specifies that the `imageView` property will be zoomed in response to zoom gestures, as well as any programmatic zooming.

```
- (UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView
{
    return self.imageView;
}
```

To specify the amount the user can zoom, you set the values of the `minimumZoomScale` and `maximumZoomScale` properties, both of which are initially set to `1.0`. The value of these properties can be set in Interface Builder's `UIScrollView` inspector panel, or programmatically. Listing 3-1 shows the code required in a subclass of UIViewController to support zooming. It assumes that the instance of the controller subclass is the delegate and implements the `viewForZoomingInScrollView:` delegate method shown above.

Listing 3-1 The `UIViewController` subclass implementation of the minimum required zoom methods

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.scrollView.minimumZoomScale=0.5;
    self.scrollView.maximumZoomScale=6.0;
    self.scrollView.contentSize=CGSizeMake(1280, 960);
    self.scrollView.delegate=self;
}
```

Specifying the zoom factor and the delegate object that implements the `viewForZoomingInScrollView:` method are the minimum requirements to support zooming using the pinch gestures.

Zooming Programmatically

A scroll view may need to zoom in response to touch events, such as double taps or other tap gestures, or in response to another user action other than a pinch gesture. To allow this, `UIScrollView` provides implementations of two methods: `setZoomScale:animated:` and `zoomToRect:animated:`.

The `setZoomScale:animated:` sets the current zoom scale to the specified value. The value must be within the range specified by the `minimumZoomScale` and `maximumZoomScale` range. If the animated parameter is `YES`, the zoom performs a constant animation until it reaches its completion; otherwise the change in scale is immediate. It is also possible to set the `zoomScale` property directly. This is the equivalent of calling `setZoomScale:animated:` passing `NO` as the animated parameter. When zooming using this method, or by changing the property directly, the view will be zoomed such that the center of the view remains stationary.

The `zoomToRect:animated:` method zooms the content in order to fill the specified rectangle. As with `setZoomScale:animated:` this method it has an animated parameter that determines whether the change in location and zoom results in animation taking place.

Your application will often want to set the zoom scale and location in response to a tap at a specific location. Because `setZoomScale:animated:` zooms around the center of the visible content, you will need a function that will take a specific location and zoom factor and convert that to a rectangle that is appropriate to `zoomToRect:animated:`. A utility method that takes a scroll view, a zoom scale, and a point to center the zoom rect on is shown in Listing 3-2

Listing 3-2 A utility method that converts a specified scale and center point to a rectangle for zooming

```
- (CGRect)zoomRectForScrollView:(UIScrollView *)scrollView withScale:(float)scale
 withCenter:(CGPoint)center {


    CGRect zoomRect;


    // The zoom rect is in the content view's coordinates.
    // At a zoom scale of 1.0, it would be the size of the
    // imageScrollView's bounds.
    // As the zoom scale decreases, so more content is visible,
    // the size of the rect grows.
    zoomRect.size.height = scrollView.frame.size.height / scale;
    zoomRect.size.width  = scrollView.frame.size.width  / scale;


    // choose an origin so as to get the right center.
    zoomRect.origin.x = center.x - (zoomRect.size.width  / 2.0);
    zoomRect.origin.y = center.y - (zoomRect.size.height / 2.0);


    return zoomRect;
}
}
```

This utility method is useful when responding to a double tap in a custom subclass that supports that gesture. To use this method simply pass the relevant UIScrollView instance, the new scale (often derived from the existing `zoomScale` by adding or multiplying a zoom amount), and the point around which to center the zoom. When responding to a double tap gesture the center point is typically the location of the tap. The rectangle that is returned by the method is suitable for passing to the `zoomToRect:animated:` method.

Informing the Delegate that the Zoom is Finished

When the user has completed the zoom pinch gestures or the programmatic zooming of the scroll view is completed, the `UIScrollView` delegate is informed by receiving a `scrollViewDidEndZooming:withView:atScale:` message.

This method is passed the scroll view instance, the scroll view subview that has been scrolled, and the scale factor at which the zoom completed as parameters. Upon receiving this delegate message your application can then take the appropriate action.

Ensuring that Zoomed Content is Sharp when Zoomed

When the content of a scroll view is zoomed, the content of the zoom view is simply scaled in response to the change in the scroll factor. This creates in larger or smaller, content, but doesn't cause the content to redraw. As a result the displayed content is not displayed sharply. When the zoomed content is an image, and your application doesn't display new, more detailed content, such as the Maps application, this may not be an issue.

If your application does need to display more detailed bitmap images in response to zooming you may want to examine the Tiling example in the *ScrollViewSuite* sample code. It uses a technique of pre-rendering the zoomed content in small chunks, and then displays those in separate views in the scroll view.

However, if your zoomed content is drawn in real time and needs to be displayed sharply when zoomed, your application class that draws the zoomed view needs to use Core Animation. The class needs to change the Core Animation class used as the `UIView` class's layer to `CATiledLayer` and draw using the Core Animation `drawLayer:inContext:` method.

Listing 3-3 (page 25) shows a complete implementation of a subclass that will draw a cross and allow you to zoom. The image remains sharp throughout the zoom action. The `Zoomable` view is a `UIView` subclass that has been added as a subview of a scroll view with a content size of (460,320) and is returned by the scroll view delegate method `viewForZoomingInScrollView:`. No action on the developer's part is required to cause the `Zoomable` view to be redrawn when zooming occurs.

Listing 3-3 Implementation of a UIView Subclass That Draw's Its Content Sharply During Zoom

```
#import "ZoomableView.h"
#import <QuartzCore/QuartzCore.h>


@implementation ZoomableView
```

```
// Set the UIView layer to CATiledLayer
+(Class)layerClass
{
    return [CATiledLayer class];
}


// Initialize the layer by setting
// the levelsOfDetailBias of bias and levelsOfDetail
// of the tiled layer
-(id)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];
    if(self) {
        CATiledLayer *tempTiledLayer = (CATiledLayer*)self.layer;
        tempTiledLayer.levelsOfDetail = 5;
        tempTiledLayer.levelsOfDetailBias = 2;
        self.opaque=YES;
    }
    return self;
}

// Implement -drawRect: so that the UIView class works correctly
// Real drawing work is done in -drawLayer:inContext
-(void)drawRect:(CGRect)r
{
}

-(void)drawLayer:(CALayer*)layer inContext:(CGContextRef)context
{
    // The context is appropriately scaled and translated such that you can draw
to this context
    // as if you were drawing to the entire layer and the correct content will be
 rendered.
    // We assume the current CTM will be a non-rotated uniformly scaled
```

```
// affine transform, which implies that
 // a == d and b == c == 0
 // CGFloat scale = CGContextGetCTM(context).a;
 // While not used here, it may be useful in other situations.

 // The clip bounding box indicates the area of the context that
 // is being requested for rendering. While not used here
 // your app may require it to do scaling in other
 // situations.
 // CGRect rect = CGContextGetClipBoundingBox(context);

 // Set and draw the background color of the entire layer
 // The other option is to set the layer as opaque=NO;
 // eliminate the following two lines of code
 // and set the scroll view background color
 CGContextSetRGBFillColor(context, 1.0,1.0,1.0,1.0);
 CGContextFillRect(context,self.bounds);

 // draw a simple plus sign
 CGContextSetRGBStrokeColor(context, 0.0, 0.0, 1.0, 1.0);
 CGContextBeginPath(context);
 CGContextMoveToPoint(context,35,255);
 CGContextAddLineToPoint(context,35,205);
 CGContextAddLineToPoint(context,135,205);
 CGContextAddLineToPoint(context,135,105);
 CGContextAddLineToPoint(context,185,105);
 CGContextAddLineToPoint(context,185,205);
 CGContextAddLineToPoint(context,285,205);
 CGContextAddLineToPoint(context,285,255);
 CGContextAddLineToPoint(context,185,255);
 CGContextAddLineToPoint(context,185,355);
 CGContextAddLineToPoint(context,135,355);
 CGContextAddLineToPoint(context,135,255);
 CGContextAddLineToPoint(context,35,255);
```

```
    CGContextClosePath(context);

    // Stroke the simple shape
    CGContextStrokePath(context);


}
```

Warning: Warning this method of drawing has a significant limitation; the UIKit drawing methods are not thread-safe, and the `drawLayer:inRect:` callbacks come in on background threads, therefore you must use the CoreGraphics drawing functions instead of the UIKit drawing methods.

Zooming by Tapping

While the basic `UIScrollView` class supports the pinch-in and pinch-out gestures with a very small amount of code, supporting a more rich zooming experience using tap detection requires more work on the part of your application.

The *iOS Human Interface Guidelines* defines a double-tap to zoom in and zoom out. That, however, assumes some specific constraints: that the view has a single level of zoom, such as in the Photos application, or that successive double-taps will zoom to the maximum amount and, once reached the next double-tap zooms back to the full-screen view. But some applications require a more flexible behavior when dealing with tap-to-zoom functionality, an example of this is the Maps application. Maps supports double-tap to zoom in, with additional double-taps zooming in further. To zoom out in successive amounts, Maps uses a two-finger touch, with the fingers close together, to zoom out in stages. While this gesture is not defined in the *iOS Human Interface Guidelines*, applications may choose to adopt it to mimic the Maps application, when the functionality is required.

In order for your application to support tap to zoom functionality, you do not need to subclass the `UIScrollView` class. Instead you implement the required touch handling in the class for which the `UIScrollView` delegate method `viewForZoomingInScrollView:` returns. That class will be responsible for tracking the number of fingers on the screen and the tap count. When it detects a single tap, a double tap, or a two-finger touch, it will respond accordingly. In the case of the double tap and two-finger touch, it should programmatically zoom the scroll view by the appropriate factor.

Implementing the Touch-Handling Code

Supporting the tap, double tap, and two-finger tap in the touch code of a subclass of `UIView` (or a descendent) requires implementing three methods: `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, and `touchesCanceled:withEvent:`. In addition, initialization of interaction, multiple touches, and tracking variables may be required. The following code fragments are from the *TapToZoom* example in the *ScrollViewSuite* sample code project, in the `TapDetectingImageView` class, which is a subclass of `UIImageView`.

Initialization

The gestures that are desired for the tap-to-zoom implementation require that user interaction and multiple touches are enabled for the view, and the methods to enable that functionality are called from the `initWithImage:` method. This method also initializes two instance variables that are used to track state in the touch methods. The `twoFingerTapIsPossible` property is a Boolean that is `YES` unless more than two fingers are in contact with the device screen. The `multipleTouches` property has a value of `NO` unless their are more than one touch events detected. A third property, `tapLocation`, is a CGPoint that is used to track the location of a double tap or the midpoint between the two fingers when a double touch is detected. This point is then used as the center point for zooming in or out using the programmatic zooming methods described in "Zooming Programmatically" (page 23).

```
- (id)initWithImage:(UIImage *)image {
    self = [super initWithImage:image];
    if (self) {
        [self setUserInteractionEnabled:YES];
        [self setMultipleTouchEnabled:YES];
        twoFingerTapIsPossible = YES;
        multipleTouches = NO;
    }
    return self;
}
```

Once the initialization has occurred the class is ready when it receives touch events.

The touchesBegan:withEvent: Implementation

The `touchesBegan:withEvent:` method first cancels any outstanding attempts to initiate handling a single finger tap, the `handleSingleTap` message. The message is canceled because, if it has been sent, it is invalid as this is an additional touch event, ruling out a single tap. If the message has not been sent because this is the first touch, canceling the perform is is ignored.

The method then updates the state of the tracking variables. If more than a single touch event has been received, then `multipleTouches` property is set to `YES`, because this may be a two finger touch. If more than two touch events have occurred, then the `twoFingerTapIsPossible` property is set to `NO`, touches by more than two fingers at a time are a gesture that is ignored.

The code for this method is as follows:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Cancel any pending handleSingleTap messages.
    [NSObject cancelPreviousPerformRequestsWithTarget:self
                                             selector:@selector(handleSingleTap)
                                               object:nil];

    // Update the touch state.
    if ([[event touchesForView:self] count] > 1)
        multipleTouches = YES;
    if ([[event touchesForView:self] count] > 2)
        twoFingerTapIsPossible = NO;

}
```

The touchesEnded:withEvent: Implementation

This method is the workhorse of the tap handling and is somewhat complex. However, the code is well documented and is simply shown below. The `midPointBetweenPoints` function is used to determine the point which a double touch will be centered upon when the `handleTwoFingerTap` method is called, which results in the view zooming out a level.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
   BOOL allTouchesEnded = ([touches count] == [[event touchesForView:self] count]);

   // first check for plain single/double tap, which is only possible if we haven't
 seen multiple touches
    if (!multipleTouches) {
        UITouch *touch = [touches anyObject];
        tapLocation = [touch locationInView:self];

        if ([touch tapCount] == 1) {
            [self performSelector:@selector(handleSingleTap)
                       withObject:nil
                       afterDelay:DOUBLE_TAP_DELAY];
```

```
    } else if([touch tapCount] == 2) {
        [self handleDoubleTap];
    }
}

// Check for a 2-finger tap if there have been multiple touches
// and haven't that situation has not been ruled out
else if (multipleTouches && twoFingerTapIsPossible) {

    // case 1: this is the end of both touches at once
    if ([touches count] == 2 && allTouchesEnded) {
        int i = 0;
        int tapCounts[2];
        CGPoint tapLocations[2];
        for (UITouch *touch in touches) {
            tapCounts[i] = [touch tapCount];
            tapLocations[i] = [touch locationInView:self];
            i++;
        }
        if (tapCounts[0] == 1 && tapCounts[1] == 1) {
            // it's a two-finger tap if they're both single taps
            tapLocation = midpointBetweenPoints(tapLocations[0],
                                                tapLocations[1]);
            [self handleTwoFingerTap];
        }
    }

    // Case 2: this is the end of one touch, and the other hasn't ended yet
    else if ([touches count] == 1 && !allTouchesEnded) {
        UITouch *touch = [touches anyObject];
        if ([touch tapCount] == 1) {
            // If touch is a single tap, store its location
            // so it can be averaged with the second touch location
            tapLocation = [touch locationInView:self];
```

```
            } else {
                twoFingerTapIsPossible = NO;
            }
        }

        // Case 3: this is the end of the second of the two touches
        else if ([touches count] == 1 && allTouchesEnded) {
            UITouch *touch = [touches anyObject];
            if ([touch tapCount] == 1) {
                // if the last touch up is a single tap, this was a 2-finger tap
                tapLocation = midpointBetweenPoints(tapLocation,
                                                    [touch locationInView:self]);
                [self handleTwoFingerTap];
            }
        }
    }

    // if all touches are up, reset touch monitoring state
    if (allTouchesEnded) {
        twoFingerTapIsPossible = YES;
        multipleTouches = NO;
    }
}
```

The touchesCancelled:withEvent: Implementation

The view receives a `touchesCancelled:withEvent:` message if the scroll view detects that the tap handling is no longer relevant because the finger has moved, which causes a scroll to being. This method simply resets the state variables.

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    twoFingerTapIsPossible = YES;
    multipleTouches = NO;
}
```

The ScrollView Suite Example

The *ScrollViewSuite* sample code project has excellent examples of implementing zooming using tap gestures. The `TapToZoom` example in the suite implements a subclass of `UIImageView` that supports zooming using the tapping behavior as displayed in the Maps application. The implementation is generic enough, through it's use of a delegate (typically the controller that manages the scroll view) to implement the actual handling of a tap, double-tap, or double touch, that you should be able to easily adapt the code, and design, to your own views.

The `TapDetectingImageView` class is the subclass of `UIImageView` that implements the touch handling, using the `RootViewController` class as the delegate that handles the actual tap and touch responses, as well as the controller that initially configures the `UIScrollView`.

Scrolling Using Paging Mode

The `UIScrollView` class supports a paging mode, which restricts a user initiated scroll action to scrolling a single screens worth of content at a time. This mode is used when displaying sequential content, such as an eBook or a series of instructions.

Configuring Paging Mode

Configuring a scroll view to support paging mode requires that code be implemented in the scroll view's controller class.

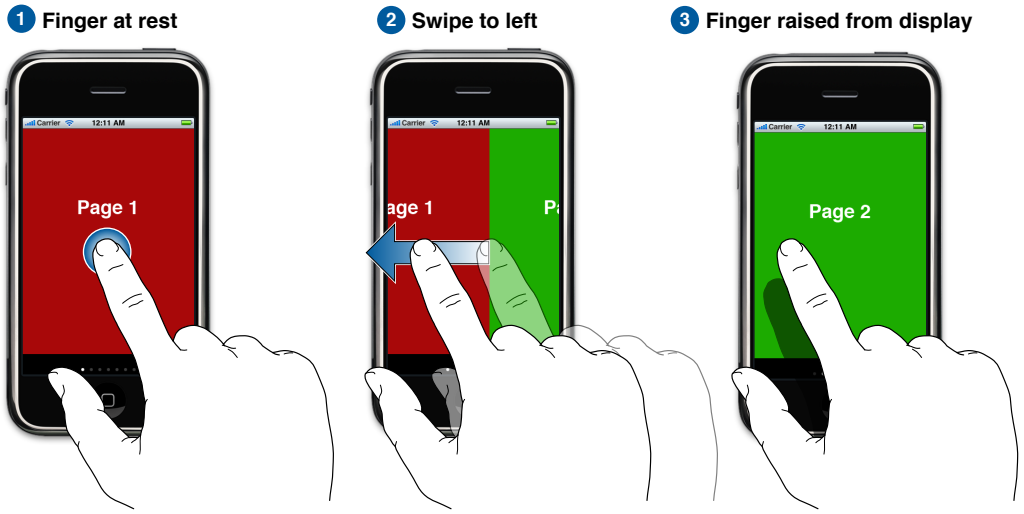
Aside from the standard scroll view initialization described in "Creating and Configuring Scroll Views" (page 8), you must also set the `pagingMode` property to `YES`.

The `contentSize` property of a paging scroll view is set so that it fills the height of the screen and that the width is a multiple of the width of the device screen multiplied by the number of pages to be displayed.

Additionally, the scroll indicators should be disabled, because the relative location as the user is touching the screen is irrelevant, or is shown using a `UIPageControl`.

Figure 5-1 shows an example of a scroll view configured in paging mode. The implementation of the shown application is available in the *PageControl* sample code.

Figure 5-1 A scroll view in paging mode and the results of a scrolling action



Configuring Subviews of a Paging Scroll View

The subviews of a paging scroll view can be configured in one of two ways. If the content is small, you could draw the entire contents at once, in a single view that is the size of the scroll view's `contentSize`. While this is the easiest to implement, it is not efficient when dealing with large content areas, or page content that takes time to draw.

When your application needs to display a large number of pages or drawing the page content can take some time, your application should use multiple views to display the content, one view for each page. This is more complicated, but can greatly increase performance and allows your application to support much larger display sets. The *PageControl* example uses this multiple view technique. By examining the sample code, you can see exactly how this technique can be implemented.

Supporting a large number of pages in a paging scroll view can be accomplished using only three view instances, each the size of the device screen: one view displays current page, another displays the previous page, and third displays the next page. The views are reused as the user scrolls through the pages.

When the scroll view controller is initialized, all three views are created and initialized. Typically the views are a custom subclass of UIView, although an application could use instances of `UIImageView` if appropriate. The views are then positioned relative to each so that when the user scrolls, the next or previous page is always in place and the content is ready for display. The controller is responsible for keeping track of which page is the current page.

To determine when the pages need to be reconfigured because the user is scrolling the content, the scroll view requires a delegate that implements the `scrollViewDidScroll:` method. The implementation of this method should track the `contentOffset` of the scroll view, and when it passes the mid point of the current view's width, the views should be reconfigured, moving the view that is no longer visible on the screen to the position that represents the next and previous page (depending on the direction the user scrolled). The delegate should then inform the view that it should draw the content appropriate for the new location it the represents.

By using this technique, you can display a large amount of content using a minimum of resources.

If drawing the page content is time consuming, your application could add additional views to the view pool, positioning those as pages on either side of the next and previous pages as scrolling occurs, and then draw the page content of those additional pages when the current content scrolls.

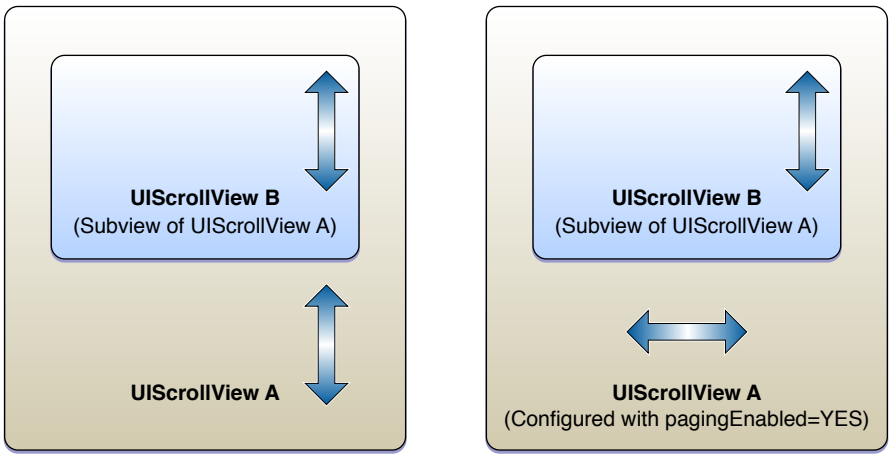
Nesting Scroll Views

To create a rich user experience, you may want to nest scroll views in your application. Before iOS 3.0 it was difficult, if not impossible to accomplish this. In iOS 3.0, this functionality is fully supported and works automatically.

Same-Direction Scrolling

Same direction scrolling occurs when a `UIScrollView` that is a subview of a `UIScrollView` both scroll in the same direction. This is shown in the left image in Figure 6-1.

Figure 6-1 Same direction scroll views and cross-directional scroll views



Note: This same-direction scrolling is supported, and there is a distinct behavior defined for the functionality, however that behavior is subject to change in future versions of iOS.

Cross-Directional Scrolling

Cross-directional scrolling is the term used when a scroll view that is a subview of another scroll view scrolls at a 90 degree angle as shown in the right image in Figure 6-1.

An example of cross directional scrolling can be found in the Stocks application. The top view is a table view, but the bottom view is a horizontal scroll view configured using paging mode. While two of its three subviews are custom views, the third view (that contains the news articles) is a `UITableView` (a subclass of `UIScrollView`) that is a subview of the horizontal scroll view. After you scroll horizontally to the news view, you can then scroll its contents vertically.

As mentioned earlier, your application does not need to do anything to support nesting scrolling. It is supported and provided by default.

Document Revision History

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

Date	Notes
2011-06-06	Corrected the content size in Listing 1-2. Corrected an error in "Setting the scroll view contentInset and scrollIndicatorInsets properties" (page 16). Other edits made to improve clarity.
2010-07-10	Corrected color definition in Basic Zooming Using the Pinch Gestures.
2010-07-07	Changed the title from "Scroll View Programming Guide for iPhone OS." Changed the title from "Scroll View Programming Guide for iPhone OS."
2010-06-14	Corrected typos.
2010-03-24	Rewrote introduction. Corrected diagrams and small typographic errors.
2010-02-24	New document that describes how to use scroll views to implement scrollable and zoomable user interfaces.

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