Erica Sadun

The iPhone™ Developer's Cookbook

Building Applications with the iPhone SDK

Developer's Library

Library of Congress Cataloging-in-Publication Data:

Sadun, Erica.
The iPhone developer's cookbook : building mobile applications with the iPhone SDK / Erica Sadun.
p. cm.
ISBN-10: 0-321-55545-7 (pbk. : alk. paper)
ISBN-13: 978-0-321-55545-8 (pbk. : alk. paper) 1. iPhone (Smartphone)–Programming. 2. Computer software–Development. 3. Mobile computing. I. Title.
QA76.8.I64S33 2009
005.26—dc22

2008030294

ISBN-13: 978-0-321-55545-8
ISBN-10: 0-321-55545-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Second printing, October 2008

Editor-in-Chief
Karen Gettman

Senior Acquisitions Editor
Chuck Toporek

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Chelsey Marti

Copy Editor
Keith Cline

Indexers
Cheryl Lenser, Erika Millen

Proofreader
San Dee Phillips

Technical Reviewers
Tim Burks, Daniel Pasco, Alex C. Schaefer

Publishing Coordinator
Romny French

Cover Designer
Gary Adair

Composition
Nonie Ratcliff

Preface

Few platforms match the iPhone's unique developer technologies. It combines OS X-based mobile computing with an innovative multitouch screen, location awareness, an onboard accelerometer, and more. When Apple introduced the iPhone Cocoa Touch SDK beta in early March 2008, developers responded in numbers that brought Apple's servers to its knees. Apple delivered more than one hundred thousand SDK downloads in less than one week. The *iPhone Developer's Cookbook* was written to address this demand, providing an accessible resource for those new to iPhone programming.

Who This Book Is For

This book is written for new iPhone developers with projects to get done and a new unfamiliar SDK in their hands. Although each programmer brings different goals and experiences to the table, most developers end up solving similar tasks in their development work: "How do I build a table?"; "How do I create a secure keychain entry?"; "How do I search the Address Book?"; "How do I move between views?"; and "How do I use Core Location?"

The *iPhone Developer's Cookbook* is aimed squarely at anyone just getting started with iPhone programming. With its clear, fully documented examples, it will get you up to speed and working productively. It presents already tested ready-to-use solutions, letting programmers focus on the specifics of their application rather than on boilerplate tasks.

How This Book Is Structured

This book offers single-task recipes for the most common issues new iPhone developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. The cookbook approach delivers cut-and-paste convenience. Programmers can add source recipes into their projects and then customize them to their needs. Each chapter groups related tasks together. Readers can jump directly to the kind of solution they're looking for without having to decide which class or framework best matches that problem.

Here's a rundown of what you'll find in this book's chapters:

- **Chapter 1: Getting Started with the iPhone SDK**

  Chapter 1 introduces the iPhone SDK and explores the iPhone as a delivery platform, limitations and all. It explains the breakdown of the standard iPhone application and enables you to build your first Hello World style samples.

- **Chapter 2: Views**

  Chapter 2 introduces iPhone views, objects that live on your screen. You see how to lay out, create, and order your views to create backbones for your iPhone applications. You read about view hierarchies, geometries, and animations as well as how users can interact with views through touch.

- **Chapter 3: View Controllers**

  The iPhone paradigm in a nutshell is this: small screen, big virtual worlds. In Chapter 3, you discover the various `UIViewController` classes that enable you to enlarge and order the virtual spaces your users interact with. You learn how to let these powerful objects perform all the heavy lifting when navigating between iPhone application screens.

- **Chapter 4: Alerting Users**

  The iPhone offers many ways to provide users with a heads up, from pop-up dialogs and progress bars to audio pings and status bar updates. Chapter 4 shows how to build these indications into your applications and expand your user-alert vocabulary.

- **Chapter 5: Basic Tables**

  Tables provide an interaction class that works particularly well on a small, cramped device. Many, if not most, apps that ship with the iPhone and iPod touch center on tables, including Settings, YouTube, Stocks, and Weather. Chapter 5 shows how iPhone tables work, what kinds of tables are available to you as a developer, and how you can use table features in your own programs.

- **Chapter 6: Advanced Tables**

  iPhone tables do not begin and end with simple scrolling lists. You can build tables with titled sections, with multiple scrolling columns, and more. You can add controls such as switches, create translucent cell backgrounds, and include custom fonts. Chapter 6 starts from where "Basic Tables" left off. It introduces advanced table recipes for you to use in your iPhone programs.

- **Chapter 7: Media**

  As you'd expect, the iPhone can load and display media from a wide variety of formats. It does music; it does movies. It handles images and Web pages. You can present PDF documents and photo albums and more. Chapter 7 shows way after way that you can import or download data into your program and display that data using the iPhone's multitouch interface.

- **Chapter 8: Control**

  The `UIControl` class provides the basis for many iPhones interactive elements, including buttons, text fields, sliders, and switches. Chapter 8 introduces controls and their use, both through well-documented SDK calls and through less-documented ones.

- **Chapter 9: People, Places, and Things**

  In addition to standard user interface controls and media components that you'd see on any computer, the iPhone SDK provides a number of tightly focused developer solutions specific to iPhone and iPod touch delivery. Chapter 9 introduces the most useful of these, including Address Book access ("people"), core location ("places"), and sensors ("things").

- **Chapter 10: Connecting to Services**

  As an Internet-connected device, the iPhone is particularly suited to subscribing to Web-based services. Apple has lavished the platform with a solid grounding in all kinds of network computing services and their supporting technologies. The iPhone SDK handles sockets, password keychains, SQL access, XML processing, and more. Chapter 10 surveys common techniques for network computing and offering recipes that simplify day-to-day tasks.

- **Chapter 11: One More Thing: Programming Cover Flow**

  Although Cover Flow is not officially included in the iPhone SDK, it offers one of the nicest and most beautiful features of the iPhone experience. With Cover Flow, you can offer your users a gorgeously intense visual selection experience that puts standard scrolling lists to shame. Chapter 11 introduces Cover Flow and shows how you can use it in your applications.

Prerequisites

Here are basics you need on hand to begin programming for the iPhone or iPod touch:

- **A copy of Apple's iPhone SDK**. Download your copy of the iPhone SDK from Apple's iPhone Dev Center (http://developer.apple.com/iphone/). You must join Apple's (free) developer program before you download.
- **An iPhone or iPod touch.** Although Apple supplies a simulator as part of its SDK, you really do need to have an actual unit to test on if you're going to develop any serious software. You'll be able to use the cable that shipped with your iPhone or iPod touch to tether your unit to the computer and install the software you've built.
- **An Apple iPhone Developer License.** You will not be able to test your software on an actual iPhone or iPod touch until you join Apple's iPhone Developer program (http://developer.apple.com/iphone/program). Members receive a certificate that allows them to sign their applications and download them to the platforms in question for testing and debugging. The program costs $99/year for individuals and companies, $299/year for in-house enterprise development.
- **An Intel-based Macintosh running Leopard.** The SDK requires a Macintosh running Leopard OS X 10.5.3 or later. Apple requires an Intel-based computer in 32-bit mode. Many features do not work properly on PPC-based Macs or Intel Macs in 64-bit mode. Reserve plenty of disk space and at least 1GB of RAM.

- **At least one available USB 2.0 port.** This enables you to tether your development iPhone or iPod touch to your computer for file transfer and testing.
- **An Internet connection.** This connection enables you to test your programs with a live WiFi connection as well as with EDGE.
- **Familiarity with Objective-C.** The SDK is built around Objective-C 2.0. The language is based on standard C with object-oriented extensions. If you have any object-oriented and C background, making the move to Objective-C is both quick and simple. Consult any Objective-C/Cocoa reference book to get up to speed.

Note

Although the SDK supports development for the iPhone and iPod touch, as well as possible yet-to-be-announced platforms, this book refers to the target platform as iPhone for the sake of simplicity. When developing for the touch, most material is applicable. This excludes certain obvious features such as telephony and onboard speakers. This book attempts to note such exceptions in the manuscript.

Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at erica@ericasadun.com or stop by www.ericasadun.com. My Web site hosts many of the applications discussed in this book. Please feel free to visit, download software, read documentation, and leave your comments.

2
Views

Pretty much everything that appears on the iPhone's screen is a view. Views act like little canvases that you can draw on with colors, pictures, and buttons. You can drag them around the screen. You can resize them. You can layer them. In this chapter, you discover how to design and build screen content using Cocoa Touch and `UIViews`. You learn about view hierarchy, geometry, and animation, and find out how to combine event feedback from `UITouches` into meaningful `UIView` responses. There's so much that `UIViews` can do that a single chapter has no hope of covering the entire class with the thoroughness it deserves. Instead, this chapter introduces essential functionality and recipes that you can use as a starting point for your own `UIView` exploration.

`UIView` and `UIWindow`

The iPhone rule goes like this: one window, many views. If you keep that idea in mind, the iPhone interface design scenario simplifies. Metaphorically speaking, `UIWindow` is the TV set, and `UIViews` are the actors on your favorite show. They can move around the screen, appear, and disappear, and may change the way they look and behave over time.

The TV set, on the other hand, normally stays still. It has a set screen size that doesn't change even if the virtual world you see through it is practically unlimited. You may even own several TVs in the same household (just like you can create several `UIWindow` instances in the same application), but you can watch just one at a time.

`UIViews` are GUI building blocks. They provide visual elements that are shown onscreen and invite user interaction. Every iPhone user interface is built from `UIViews` displayed within one `UIWindow`, which is itself a specialized kind of `UIView`. The window acts a container; it is the root of the display hierarchy. It holds all the visible application components within itself. The following sections will give you just a taste of the kind of ways you can control and manipulate views, their hierarchy, and their geometry.

Hierarchy

A tree-based hierarchy orders what you see on your iPhone screen. Starting with the main window, views are laid out in a specifically hierarchical way. All views may have children, called subviews. Each view, including the root window, owns an ordered list of these subviews. Views might own many subviews; they might own none. Your application determines how views are laid out and who owns whom.

Subviews display onscreen in order, always from back to front. And because the iPhone supports view transparency, this works exactly like a stack of animation cells—those transparent sheets used to create cartoons. Only the parts of the sheets that have been painted are shown. The clear parts allow any visual elements behind that sheet to be seen.

Figure 2-1 shows a little of the layering used in a typical window. Here you see the window that owns a `UINavigationController`-based window. The window (represented by the clear, rightmost element) owns a Navigation subview, which in turn owns two subview buttons (one left and one right) and a table. These items stack together to build the GUI.

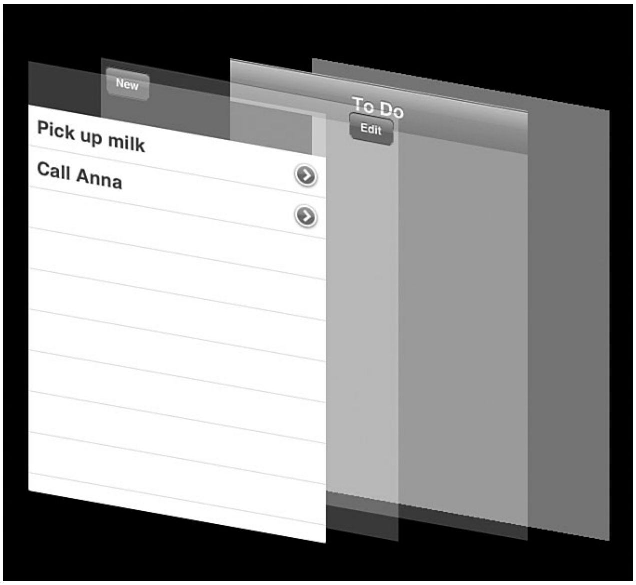

Figure 2-1 Adding subview hierarchies allows you to build complex GUIs.

Notice how the buttons appear over the navigation bar and how the table is sized so that it won't obscure either the buttons or bar. The button frames are small, taking up very little space onscreen. The table frame is large, occupying the majority of screen space. Here are some ways you can manage subviews in your programs:

- To add a subview, use a call to `[parentView addSubview:child]`. Newly added subviews are always frontmost on your screen.
- Query any view for its children by asking it for `[parentView subviews]`. This returns an array of views, ordered from back to front.

- Remove a subview from its parent with `[childView removeFromSuperview]`.
- Reorder subviews using `[parentView exchangeSubviewAtIndex:i withSubviewAtIndex:j]`. Move subviews to the front or back using `bringSubviewToFront:` or `sendSubviewToBack:`.
- Tag your subviews using `setTag:`. This identifies views by tagging them with a number. Retrieve that view from the child hierarchy by calling `viewWithTag:` on the parent.

Note

You can tag any instance that is a child of `UIView`, including windows and controls. So if you have many onscreen buttons and switches, for example, add tags so that you can tell them apart when users trigger them.

Geometry and Traits

Every view uses a frame to define its boundaries. The frame specifies the outline of the view: its location, width, and height. You define the frame rectangle using Core Graphics structures. For frames, this usually means a `CGRect` rectangle made up of an origin (a `CGPoint`, x and y) and a size (a `CGSize`, width and height). Here are some quick facts about these types.

`CGRect`

The `CGRect` structure defines an onscreen rectangle. It contains an origin (rect.origin) and a size (rect.size). These are `CGRect` functions you'll want to be aware of:

- `CGRectMake(origin.x, origin.y, size.width, size.height)` defines rectangles in your code.
- `NSStringFromCGRect(someCGRect)` converts a `CGRect` structure to a formatted string.
- `CGRectFromString(aString)` recovers a rectangle from its string representation.
- `CGRectInset(aRect)` enables you to create a smaller or larger a rectangle that's centered on the same point. Use a positive inset for smaller rectangles, negative for larger ones.
- `CGRectIntersectsRect(rect1, rect2)` lets you know whether rectangle structures intersect. Use this function to know when two rectangular onscreen objects overlap.
- `CGRectZero` is a rectangle constant located at `(0,0)` whose width and height are zero. You can use this constant when you're required to create a frame but you're still unsure what that frame size or location will be at the time of creation.

`CGPoint` and `CGSize`

Points refer to locations defined with x and y coordinates; sizes have width and height. Use `CGPointMake(x, y)` to create points. `CGSizeMake(width, height)` creates sizes. Although these two structures appear to be the same (two floating-point values), the iPhone SDK differentiates between them. Points refer to locations. Sizes refer to extents. You cannot set `myFrame.origin` to a size.

As with rectangles, you can convert them to and from strings: `NSStringFromCGPoint()`, `NSStringFromCGSize()`, `CGSizeFromString()`, and `CGPointFromString()` perform these functions.

Defining Locations

You can define a view's location by setting its center (which is a `CGPoint`) or bounds (`CGRect`). Unlike the frame, a view's bounds reflect the view's frame in its *own* coordinate system. In practical terms, that means the origin of the bounds is (0.0, 0.0), and its size is its width and height.

When you want to move or resize a view, update its frame's origin, center, or size. You don't need to worry about things such as rectangular sections that have been exposed or hidden. The iPhone takes care of the redrawing. This lets you treat your views like tangible objects and delegate the rendering issues to Cocoa Touch. For example

```
[myView setFrame:CGRectMake(0.0f, 50.0f, mywidth, myheight)];
```

Transforms

Standard Core Graphics calls transform views in real time. For example, you can apply clipping, rotation, or other 2D geometric effects. Cocoa Touch supports an entire suite of affine transforms (translate, rotate, scale, skew, and so on). The `drawRect:` method for any `UIView` subclass provides the entry point for drawing views through low-level Core Graphics calls.

> **Note**
>
> When calling Core Graphics functions, keep in mind that Quartz lays out its coordinate system from the bottom left, whereas `UIViews` have their origin at the top left.

Other View Traits

In addition to the physical screen layout, you can set the following view traits among others:

- Every view has a translucency factor (alpha) that ranges between opaque and transparent. Adjust this by issuing `[myView setAlpha:value]`, where the alpha values falls between 0.0 (fully transparent) and 1.0 (fully opaque).
- You can assign a color to the background of your view. `[myView setBackgroundColor:[UIColor redColor]]` colors your view red.

View Layout

Figure 2-2 shows the layout of a typical iPhone application screen. For current releases of the iPhone, the screen size is 320x480 pixels in portrait mode, 480x320 pixels in landscape. At the top of the screen, whether in landscape or portrait mode, a standard status bar occupies 20 pixels of height. To query the status bar frame, call `[[UIApplication sharedApplication] statusBarFrame]`.

If you'd rather free up those 20 pixels of screen space for other use, you can hide the status bar entirely. Use this `UIApplication` call: `[UIApplication sharedApplication] setStatusBarHidden:YES animated:NO]`. Alternatively, set the `UIStatusBarHidden` key to `<true/>` in your application Info.plist file.

To run your application in landscape-only mode, set the status bar orientation to landscape. Do this even if you plan to hide the status bar (that is, `[[UIApplication sharedApplication] setStatusBarOrientation: UIInterfaceOrientationLandscapeRight]`). This forces windows to display side to side and produces a proper landscape keyboard.

The `UIScreen` object acts as a stand in for the iPhone's physical screen (`[UIScreen mainScreen]`). The screen object maps view layout boundaries into pixel space. It returns either the full screen size (`bounds`) or just the rectangle that applies to your application (`applicationFrame`). This latter takes the size of your status bar and, if used, any toolbars/navigation bars into account.

By default, `UINavigationBar`, `UIToolbar`, and `UITabBar` objects are 44 pixels in height each. Use these numbers to calculate the available space on your iPhone screen and lay out your application views when not using Interface Builder's layout tools.

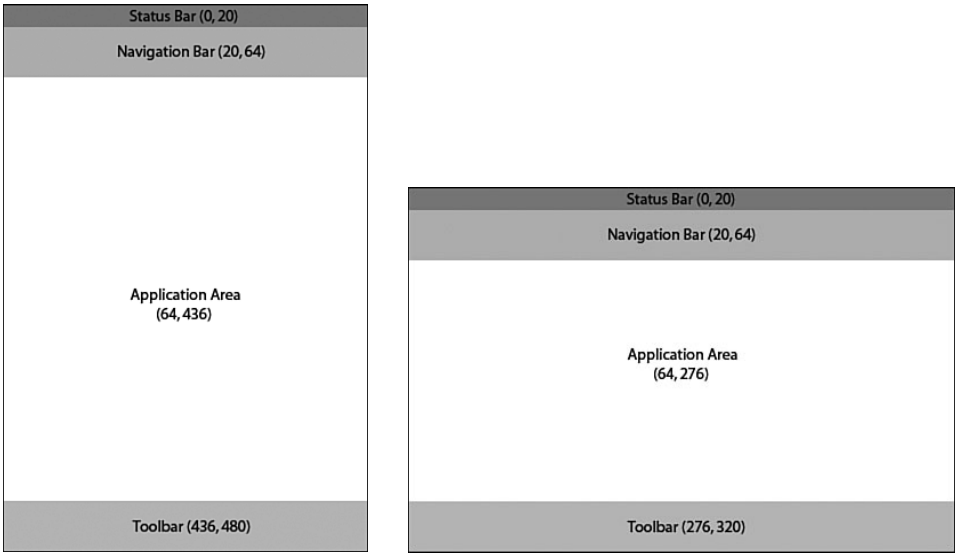

Figure 2-2 On current generations of the iPhone, the status bar is 20 pixels high, often followed below by a 44-pixel-high navigation bar. If you use a toolbar at the bottom of your screen, that will also occupy 44 pixels. It helps to use Photoshop or some other image layout program to design your screens taking these geometries into account.

Gestures

Views intercept user touches. This integration between the way things look and the way things react enables you to add a meaningful response to taps, drags, and what have you. Adding touch handlers like `touchesBegan: withEvent:` to your views allows you to intercept user touches, determine the phase of the touch (the equivalent of mouse down, mouse dragged, mouse up), and produce feedback based on those touches.

The `UITouch` class tells you where the event took place (`locationInView:`) and the tap count (`tapCount`), which is vital for distinguishing between single- and double-taps. Several recipes in this chapter demonstrate how to use these gesture responses and how to integrate view geometry and hierarchy into your applications for enticing, layered, direct-manipulation interfaces.

Recipe: Adding Stepwise Subviews

Expand your view hierarchy by calling `addSubview:`. This adds a subview to some other view. Recipe 2-1 shows a simple `UIViewController`'s `loadView` method that defines a series of stepped subviews. It demonstrates the basics of allocating, framing, and adding views.

These subviews are not nested, in that they all belong to the same parent. They're indented so that you can see them all at once. The indentation uses the handy `CGRectInset()` function. Pass it a rectangle (using the `CGRect` structure) and two insets—horizontal and vertical—and it returns the inset, centered rectangle. Here, each subview is inset from its parent or sibling's frame by 32 pixels on each side.

In their simplest form, views are little more than transparent placeholders. Coloring the view backgrounds distinguishes one view from another in the absence of meaningful content (see Figure 2-3). It's a useful trick when trying to test layouts before committing to an actual design.

Always keep your coordinate system in mind. When working with view hierarchy, you must define a view's frame in its parent's coordinate system. The example in Recipe 2-1 requests the application frame to lay out the main view and then resets its origin to (0, 0). Resetting the origin updates the frame from the screen's to the main view's coordinate system. This reset forms the basis for the view layout that follows.

Recipe 2-1 **Adding Nested Subviews**

```
- (void)loadView
{
    // Create the main view
    CGRect appRect = [[UIScreen mainScreen] applicationFrame];
    contentView = [[UIView alloc] initWithFrame:appRect];
    contentView.backgroundColor = [UIColor whiteColor];

    // Provide support for autorotation and resizing
    contentView.autoresizesSubviews = YES;
    contentView.autoresizingMask = (UIViewAutoresizingFlexibleWidth |
    ➥UIViewAutoresizingFlexibleHeight);
```

Recipe 2-1 **Continued**

```
    self.view = contentView;
    [contentView release];

    // reset the origin point for subviews. The new origin is 0,0
    appRect.origin = CGPointMake(0.0f, 0.0f);

    // Add the subviews, each stepped by 32 pixels on each side
    UIView *subview = [[UIView alloc] initWithFrame:CGRectInset(appRect, 32.0f,
    ➥32.0f)];
    subview.backgroundColor = [UIColor lightGrayColor];
    [contentView addSubview:subview];
    [subview release];

    subview = [[UIView alloc] initWithFrame:CGRectInset(appRect, 64.0f, 64.0f)];
    subview.backgroundColor = [UIColor darkGrayColor];
    [contentView addSubview:subview];
    [subview release];

    subview = [[UIView alloc] initWithFrame:CGRectInset(appRect, 96.0f, 96.0f)];
    subview.backgroundColor = [UIColor blackColor];
    [contentView addSubview:subview];
    [subview release];
}
```

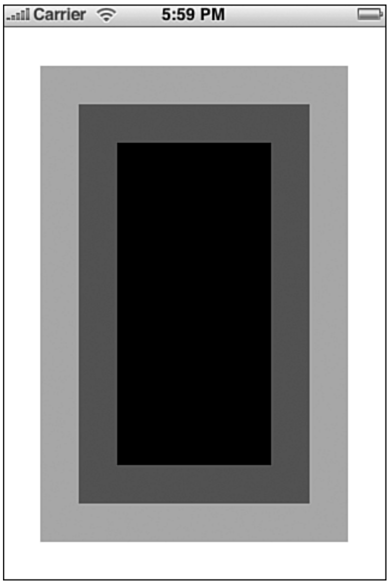

Figure 2-3 The code in Recipe 2-1 defines a `UIView` controller's main view with three colored, nested subviews.

Reorienting

Extending Recipe 2-1 to enable orientation changes takes thought. You cannot just swap each subview's heights and widths as you might assume. That's because the shapes of horizontal and vertical applications on the iPhone use different aspect ratios. Assuming a 20-pixel status bar, portrait view areas are 320 pixels wide by 460 pixels high; landscapes are 480 pixels wide by 300 pixels high (refer to Figure 2-2). This difference throws off interfaces that depend solely on rotation to reorient.

To rotate this example, add code that distinguishes landscape orientations from portrait ones and adjust the frames accordingly. This is shown in Recipe 2-2.

Avoid reorientation schemes that rely on toggling (for example, "I was just in portrait mode so, if the orientation changed, I must be in landscape mode.") It's entirely possible to switch from left-landscape to right-landscape without hitting a portrait state in-between. Orientation is all about sensors and feedback, and the iPhone is not guaranteed to catch any middle state between two orientations. Fortunately, `UIKit` provides a `UIViewController` callback that alerts you to new orientations and that specifies what that orientation will be.

Recipe 2-2 **Adding Reorientation Support to the Preceding Subview Example**

```
- (void)willRotateToInterfaceOrientation:
    (UIInterfaceOrientation)orientation
        duration:(NSTimeInterval)duration  {

    CGRect apprect;
    apprect.origin = CGPointMake(0.0f, 0.0f);

    // adjust the frame size based on actual orientation
    if ((orientation == UIInterfaceOrientationLandscapeLeft) ||
    ➥(orientation == UIInterfaceOrientationLandscapeRight))
        apprect.size = CGSizeMake(480.0f, 300.0f);
    else
        apprect.size = CGSizeMake(320.0f, 460.0f);

    // resize each subview accordingly
    float offset = 32.0f;
    for (UIView *subview in [contentView subviews])    {
        CGRect frame = CGRectInset(apprect, offset, offset);
        [subview setFrame:frame];
        offset += 32.0f;
    }
}

// Allow the view to respond to iPhone Orientation changes
-(BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
```

Recipe 2-2 **Continued**

```
{
    return YES;
}
```

Recipe: Dragging Views

Cocoa Touch simplifies direct view manipulation. When dealing with many onscreen views, the iPhone takes charge of deciding which view the user touched and passes any touch events to the proper view for you. This helps you write concrete direct-manipulation interfaces where users touch, drag, and interact with onscreen objects.

Recipe 2-3 centers on touches in action. This example creates a child of `UIImageView` called `DragView` that enables users to drag the view around the iPhone screen. Being an image view, it's important to enable its user interaction, via `[dragger setUserInteractionEnabled:YES]`. This holds true for backdrops as well as direct-interaction views. Whenever working with `UIImageView` in direct-manipulation interfaces, make sure to enable interaction, no matter what role in the view hierarchy. With image views, the user interaction toggle affects all the view's children as well as the view itself.

When a user first touches any `DragView` (see the flowers in Figure 2-4), the object stores the start location as an offset from the view's origin. As the user drags, the view moves along with the finger—always maintaining the same origin offset so that the movement feels natural.

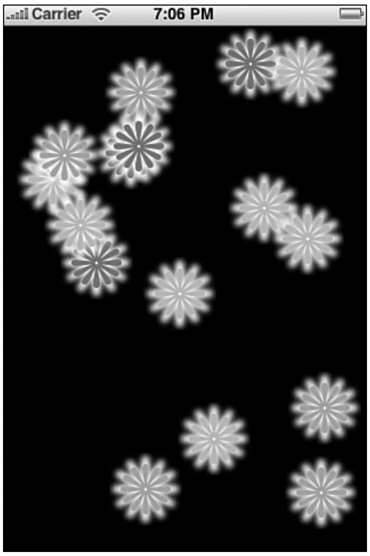

Figure 2-4 The code in Recipe 2-3 creates an interface with 16 flowers that can be dragged around the iPhone screen.

Note

The way the example in Figure 2-4 is built, you can use multiple fingers to drag more than one flower around the screen at once. It's not multitouch per se because each flower (`UIView`) responds to only one touch at a time. A discussion of true multitouch interaction follows later in this chapter.

Touching an object also does one more thing in this code: It pops that object to the front of the parent view. This means any dragged object always floats *over* any other object onscreen. Do this by telling the view's parent (its `superview`) to bring the view to its front.

`UITouch`

The `UITouch` class defines how fingers move across the iPhone screen. Touches are sent while invoking the standard began, moved, and ended handlers. You can also query user events (of the `UIEvent` class) to return touches affecting a given view through `touchesForView:` and `touchesForWindow:`. These calls return an unordered set (`NSSet`) of touches.

Note

Send `allObjects` to any `NSSet` to return an array of those objects.

A touch tells you several things: where the touch took place (both the current and most recent previous location), what stage of the touch was used (essentially mouse down, mouse moved, mouse up), a tap count (for example, single-tap/double-tap), when the touch took place (through a time stamp), and so forth.

For nonmultitouch interaction styles, assume that you're dealing with a single touch at any time. The code in Recipe 2-3 recovers the first available touch for each event by calling `anyObject` on the returned touch set.

Recipe 2-3 **Building Multiple Draggable Views**

```
/*
 *   DragView: Draggable views
 */

@interface DragView : UIImageView
{
    CGPoint startLocation;
}
@end

@implementation DragView

// Note the touch point and bring the touched view to the front
- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
```

Recipe 2-3 **Continued**

```
{
    CGPoint pt = [[touches anyObject] locationInView:self];
    ➥startLocation = pt;
    [[self superview] bringSubviewToFront:self];
}

// As the user drags, move the flower with the touch
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    CGPoint pt = [[touches anyObject] locationInView:self];
    CGRect frame = [self frame];

    frame.origin.x += pt.x - startLocation.x;
    frame.origin.y += pt.y - startLocation.y;
    [self setFrame:frame];
}
@end

/*
 *  Hello Controller: The primary view controller
 */

@interface HelloController : UIViewController
{
    UIView *contentView;
}
@end

@implementation HelloController

#define MAXFLOWERS 16

CGPoint randomPoint() {return CGPointMake(random() % 256, random() % 396);}

- (void)loadView
{
    // Create the main view with a black background
    CGRect apprect = [[UIScreen mainScreen] applicationFrame];
    contentView = [[UIView alloc] initWithFrame:apprect];
    contentView.backgroundColor = [UIColor blackColor];
    self.view = contentView;
    [contentView release];

    // Add the flowers to random points on the screen
    for (int i = 0; i < MAXFLOWERS; i++)
    {
```

Recipe 2-3 **Continued**

```
        CGRect dragRect = CGRectMake(0.0f, 0.0f, 64.0f, 64.0f);
        dragRect.origin = randomPoint();
        DragView *dragger = [[DragView alloc] initWithFrame:dragRect];
        [dragger setUserInteractionEnabled:YES];

        // select random flower color
        NSString *whichFlower = [[NSArray arrayWithObjects:@"blueFlower.png",
        ➥@"pinkFlower.png", @"orangeFlower.png", nil] objectAtIndex:(random() %
        ➥3)];
        [dragger setImage:[UIImage imageNamed:whichFlower]];

        // add the new subview
        [contentView addSubview:dragger];
        [dragger release];
    }
}

-(void) dealloc
{
    [contentView release];
    [super dealloc];
}
@end
```

Adding Persistence

Persistence represents a key iPhone design touch point. After users leave a program, Apple strongly recommends that they return to a state that matches as closely to where they left off as possible. Adding persistence to this sample code involves several steps:

1. Storing the data
2. Resuming from a saved session
3. Providing a startup image that matches the last session

Storing State

Every view knows its position because you can query its `frame`. This enables you to recover and store positions for each onscreen flower. The flower type (green, pink, or blue) is another matter. For each view to report its current flower, the `DragView` class must store that value, too. Adding a string instance variable enables the view to return the image name used. Listing 2-1 shows the extended `DragView` class definition.

Listing 2-1 **The Updated `DragView` Class Includes a String to Store the Flower Type**

```
@interface DragView : UIImageView
{
    CGPoint startLocation;
```

Listing 2-1 **Continued**

```
    NSString *whichFlower;
}
@property (nonatomic, retain) NSString *whichFlower;
@end
```

Adding this extra variable enables the `HelloController` class to store both a list of colors and a list of locations to its defaults file. A simple loop collects both values from each draggable view and then stores them. Listing 2-2 presents an `updateDefaults` method, as defined in `HelloController`. This method saves the current state to disk. It should be called in the application delegate's `applicationWillTerminate:` method, just before the program ends.

Notice the use here of `NSStringFromCGRect()`. It provides a tight way to store frame information as a string. To recover the rectangle, issue `CGRectFromString()`. Each call takes one argument: a `CGRect` in the first case, an `NSString *` in the second. The `UIKit` framework provides calls that translate points and sizes as well as rectangles to and from strings.

Defaults, as you can see, work like a dictionary. Just assign an object to a key and the iPhone "automagically" updates the preferences file associated with your application ID. Your application ID is defined in Info.plist. Defaults are stored in Library/Preferences inside your application's sandbox. Calling the `synchronize` function updates those defaults immediately instead of waiting for the program to terminate.

Listing 2-2 **Storing Flower Locations via User Defaults**

```
// Collect all the colors and locations and save them for the next use
- (void) updateDefaults
{
    NSMutableArray *colors = [[NSMutableArray alloc] init];
    NSMutableArray *locs = [[NSMutableArray alloc] init];

    for (DragView *dv in [contentView subviews]) {
        [colors addObject:[dv whichFlower]];
        [locs addObject:NSStringFromCGRect([dv frame])];
    }

    [[NSUserDefaults standardUserDefaults] setObject:colors forKey:@"colors"];
    [[NSUserDefaults standardUserDefaults] setObject:locs forKey:@"locs"];
    [[NSUserDefaults standardUserDefaults] synchronize];
    [colors release];
    [locs release];
}
```

Recovering State

Persistence awareness generally resides in the view controller's `init` or `loadView` (for example, before the view actually appears). These methods should find any previous state information and, for this example, match the flowers to that state. When querying user defaults, this code checks whether state data is unavailable (for example, the value returned is nil). When state data goes missing, the method creates random flowers at random points. Listing 2-3 shows a state-aware version of `loadView`.

> **Note**
>
> When working with large data sources, you may want to initialize and populate your saved object array in the `UIViewController`'s `init` method, and then draw them in `loadView`. Where possible, use threading when working with many objects to avoid blocking.

Listing 2-3 **Checking for Previous State**

```
- (void)loadView
{
    // Create the main view
    CGRect apprect = [[UIScreen mainScreen] applicationFrame];
    contentView = [[UIView alloc] initWithFrame:apprect];
    contentView.backgroundColor = [UIColor blackColor];
    self.view = contentView;

    // Attempt to read in previous colors and locations
    NSMutableArray *colors, *locs;
    colors = [[NSUserDefaults standardUserDefaults] objectForKey:@"colors"];
    locs = [[NSUserDefaults standardUserDefaults] objectForKey:@"locs"];

    for (int i = 0; i < MAXFLOWERS; i++)
    {
        // Use a random point unless there's a previous location
        CGRect dragRect = CGRectMake(0.0f, 0.0f, 64.0f, 64.0f);
        dragRect.origin = randomPoint();
        if (locs && ([locs count] == MAXFLOWERS))
           ➥dragRect = CGRectFromString([locs objectAtIndex:i]);
        DragView *dragger = [[DragView alloc] initWithFrame:dragRect];
        [dragger setUserInteractionEnabled:YES];

        // Use a random color unless there's a previous color
        NSString *whichFlower = [[NSArray arrayWithObjects:@"blueFlower.png",
        ➥@"pinkFlower.png", @"orangeFlower.png", nil] objectAtIndex:(random()
        ➥% 3)];
        if (colors && ([colors count] == MAXFLOWERS))
           ➥whichFlower = [colors objectAtIndex:i];
        [dragger setWhichFlower:whichFlower];
        [dragger setImage:[UIImage imageNamed:whichFlower]];
```

Listing 2-3 **Continued**

```
        // Add the subview
        [contentView addSubview:dragger];
        [dragger release];
    }

}
```

Startup Image

Apple has not yet included persistence screenshot capabilities into its official SDK release, although the functionality is partially available in the `UIKit` framework as an undocumented call. To access the `_writeApplicationSnapshot` feature shown in Listing 2-4, you must add it by hand to the `UIApplicationClass` interface. Once added, you can build a cached shot of your screen before ending the application.

> **Note**
>
> See Chapter 1, "Introducing the iPhone SDK," for further discussion about using undocumented calls and features in your programs.

The idea is this: When you leave the application, you snap a picture of the screen. Then when your application starts up (presumably returning you to the same state you left with), the cached image acts as the Default.png image, giving the illusion that you're jumping directly back without any startup sequence.

Apple has yet to enable this feature with the iPhone SDK, and at the time of writing, applications cannot check in to find updated snapshots. Hopefully, Apple will provide this functionality in a future firmware release.

Listing 2-4 **Screenshotting Before Application Termination**

```
@interface UIApplication (Extended)
-(void) _writeApplicationSnapshot;
@end

[[UIApplication sharedApplication] _writeApplicationSnapshot];
```

Recipe: Clipped Views

When working with direct-manipulation interfaces, it's unlikely that you'll want to deal solely with rectangular views. Soft borders, rounded corners, and other visual enhancements are easily added to `UIView` instances.

Clipping creates view shapes that fill only part of a view's frame. You can produce clipping with Core Graphics using the `drawRect:` method of a `UIView` object, just as you would on a Macintosh. Core Graphics enables you to build paths from sources including points, lines, standard shapes (such as ellipses), and Bézier curves. Clipping your views to these paths creates the illusion of nonrectangular onscreen objects. Figure 2-5 shows a number of onscreen circular clipped views, clearly overlapping with each other. These views were created by the code shown in Listing 2-5. This code creates a path, performs the clipping, and then draws into the clipped view.

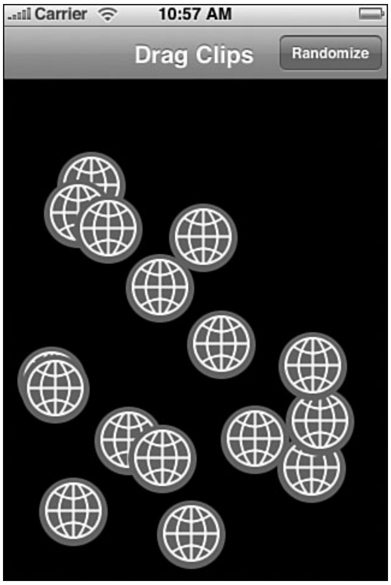

Figure 2-5 Clipping enables you to create nonrectangular views onscreen from rectangular source material, using rectangular `UIView` frames.

Listing 2-5 **Clipping a View to a Circular Path**

```
- (void) drawRect: (CGRect) aRect
{

    CGRect bounds = CGRectMake(0.0f, 0.0f, SIDELENGTH, SIDELENGTH);

    // Create a new path
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGMutablePathRef path = CGPathCreateMutable();

    // Add circle to path
    CGPathAddEllipseInRect(path, NULL, bounds);
    CGContextAddPath(context, path);
```

Listing 2-5 **Continued**

```
    // Clip to the circle and draw the logo
    CGContextClip(context);
    [logo drawInRect:bounds];
    CFRelease(path);
}
```

Balancing Touches with Clipping

Visual clipping does not affect how `UIViews` respond to touches. The iPhone senses user taps throughout the entire view frame. This includes the undrawn area such as the corners of the frame outside the actual circles of Figure 2-5 just as much as the clipped presentation. That means that unless you add some sort of hit test, users may attempt to tap through to a view that's "obscured" by the clear portion of the `UIView` frame.

Listing 2-6 adds a simple hit test to the clipped views, determining whether touches fall within the clipping path. I implemented circular clipping and circular hit tests to provide the simplest example. Use any computable test method you like to determine whether a user touch intersects the view. Add `pointInside:withEvent:` to your `UIView` subclass and return `YES` when the touch has properly hit your view or `NO` when it does not.

Listing 2-6 **Checking Circular Views against Touches**

```
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    CGPoint pt;
    float HALFSIDE = SIDELENGTH / 2.0f;

    // normalize with centered origin
    pt.x = (point.x - HALFSIDE) / HALFSIDE;
    pt.y = (point.y - HALFSIDE) / HALFSIDE;

    // x^2 + y^2 = hypoteneus length
    float xsquared = pt.x * pt.x;
    float ysquared = pt.y * pt.y;

    // If the length < 1, the point is within the clipped circle
    if ((xsquared + ysquared) < 1.0) return YES;
    return NO;
}
```

Accessing Pixel-by-Pixel Values

There are many ways to test user touches against views. Listing 2-6 computed whether a touch fell within a circle's radius. With hit masks and variable transparency images, you can test against a point's alpha value. Translucency controls whether you trigger a response. Listing 2-7 extends the `UIImageView` class to add an image's bitmap representation. It tests touches against alpha values in the bitmap, point by point. Pixels whose alpha levels fall below 0.5 will not respond to touches using this code.

> **Note**
>
> The code in this listing returns a bitmap context, and its bitmap data is based on Apple sample code.

Listing 2-7 **Testing Touch Hits Against a Bitmap**

```
// Return a bitmap context using alpha/red/green/blue byte values
CGContextRef CreateARGBBitmapContext (CGImageRef inImage)
{
    CGContextRef    context = NULL;
    CGColorSpaceRef colorSpace;
    void *          bitmapData;
    int             bitmapByteCount;
    int             bitmapBytesPerRow;

    size_t pixelsWide = CGImageGetWidth(inImage);
    size_t pixelsHigh = CGImageGetHeight(inImage);
    bitmapBytesPerRow   = (pixelsWide * 4);
    bitmapByteCount     = (bitmapBytesPerRow * pixelsHigh);
    colorSpace = CGColorSpaceCreateDeviceRGB();

    if (colorSpace == NULL)
    {
        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

    // allocate the bitmap & create context
    bitmapData = malloc( bitmapByteCount );
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Memory not allocated!");
        CGColorSpaceRelease( colorSpace );
        return NULL;
    }
```

Listing 2-7 **Continued**

```
    context = CGBitmapContextCreate (bitmapData, pixelsWide, pixelsHigh, 8,
    ➥bitmapBytesPerRow, colorSpace, kCGImageAlphaPremultipliedFirst);
    if (context == NULL)
    {
        free (bitmapData);
        fprintf (stderr, "Context not created!");
    }
    CGColorSpaceRelease( colorSpace );
    return context;
}

// Return Image Pixel data as an ARGB bitmap
unsigned char *RequestImagePixelData(UIImage *inImage)
{
    CGImageRef img = [inImage CGImage];
    CGSize size = [inImage size];

    CGContextRef cgctx = CreateARGBBitmapContext(img, size);
    if (cgctx == NULL) return NULL;

    CGRect rect = {{0,0},{size.width, size.height}};
    CGContextDrawImage(cgctx, rect, img);
    unsigned char *data = CGBitmapContextGetData (cgctx);
    CGContextRelease(cgctx);

    return data;
}

// Create an Image View that stores a copy of its image as an addressable bitmap
@interface BitMapView : UIImageView
{
    unsigned char *bitmap;
    CGSize size;
    UIView *colorView;
}
@end
@implementation BitMapView

// Hit test relies on the alpha level of the touched pixel
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    long startByte = (int)((point.y * size.width) + point.x) * 4;
    int alpha = (unsigned char) bitmap[startByte];
    return (alpha > 0.5);
}
```

Listing 2-7 **Continued**

```
-(void) setImage:(UIImage *) anImage
{
    [super setImage:anImage];
    bitmap = RequestImagePixelData(anImage);
    size = [anImage size];
}
@end
```

Recipe: Detecting Multitouch

By enabling multitouch interaction in your `UIViews`, the iPhone enables you to recover and respond to multifinger interaction. This recipe, shown in Recipe 2-4, demonstrates how to add multitouch to your iPhone applications.

To begin, set `multipleTouchEnabled` to `YES` or override `isMultipleTouchEnabled` for your view. This tells your application to poll for more than one `UITouch` at a time. Now when you call `touchesForView:`, the returned set may contain several touches. Use `NSSet`'s `allObjects` method to convert that set into an addressable `NSArray`. When the array's count exceeds one, you know you're dealing with multitouch.

In theory, the iPhone could support an arbitrary number of touches. In practice, multitouch is limited to five finger touches at a time. Even five at a time goes beyond what most developers need. There aren't many meaningful gestures you can make with five fingers at once. This particularly holds true when you grasp the iPhone with one hand and touch with the other. Perhaps it's a comfort to know that if you need to, the extra finger support has been built in. Unfortunately, when you are using three or more touches at a time, the screen has a tendency to lose track of one or more of those fingers. It's hard to programmatically track smooth gestures when you go beyond two finger touches.

Touches are not grouped. If, for example, you touch the screen with two fingers from each hand, there's no way to determine which touches belong to which hand. The touch order is arbitrary. Although grouped touches retain the same finger order for the lifetime of a single touch event (down, move, up), the order may change the next time your user touches the screen. When you need to distinguish touches from each other, build a touch dictionary indexed by the touch objects.

Note

The `drawRect:` routine in Recipe 2-4 clears its context each time it is called. This removes previous circles and lines from the display. Comment out this line if you want to see an event trail.

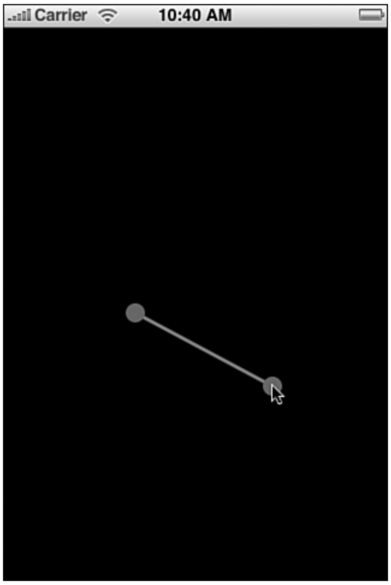

Figure 2-6 The iPhone enables you to capture multitouch events as well as single-touch ones. In this example, two circles mark the points at which the user has touched the screen.

Recipe 2-4 **Visualizing Multitouch**

```
@interface MultiTouchView : UIView
{
    CGPoint loc1, loc2;
}
@property (nonatomic) CGPoint loc1;
@property (nonatomic) CGPoint loc2;
@end

@implementation MultiTouchView
@synthesize loc1;
@synthesize loc2;

- (BOOL) isMultipleTouchEnabled {return YES;}

- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    NSArray *allTouches = [touches allObjects];
    int count = [allTouches count];
    if (count > 0) loc1 = [[allTouches objectAtIndex:0] locationInView:self];
    if (count > 1) loc2 = [[allTouches objectAtIndex:1] locationInView:self];
```

Recipe 2-4 **Continued**

```
    [self setNeedsDisplay];
}

// React to moved touches the same as to "began"
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    [self touchesBegan:touches withEvent:event];
}

- (void) drawRect: (CGRect) aRect
{
    // Get the current context
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextClearRect(context, aRect);

    // Set up the stroke and fill characteristics
    CGContextSetLineWidth(context, 3.0f);
    CGFloat gray[4] = {0.5f, 0.5f, 0.5f, 1.0f};
    CGContextSetStrokeColor(context, gray);
    CGFloat red[4] = {0.75f, 0.25f, 0.25f, 1.0f};
    CGContextSetFillColor(context, red);

    // Draw a line between the two location points
    CGContextMoveToPoint(context, loc1.x, loc1.y);
    CGContextAddLineToPoint(context, loc2.x, loc2.y);
    CGContextStrokePath(context);

    CGRect p1box = CGRectMake(loc1.x, loc1.y, 0.0f, 0.0f);
    CGRect p2box = CGRectMake(loc2.x, loc2.y, 0.0f, 0.0f);
    float offset = -8.0f;

    // circle point 1
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddEllipseInRect(path, NULL, CGRectInset(p1box, offset, offset));
    CGContextAddPath(context, path);
    CGContextFillPath(context);
    CFRelease(path);

    // circle point 2
    path = CGPathCreateMutable();
    CGPathAddEllipseInRect(path, NULL, CGRectInset(p2box, offset, offset));
    CGContextAddPath(context, path);
    CGContextFillPath(context);
    CFRelease(path);
}
@end
```

Note

Apple provides many Core Graphics/Quartz 2D resources on its developer Web site. Although these forums, mailing lists, and source code samples are not iPhone specific, they offer an invaluable resource for expanding your iPhone Core Graphics knowledge.

`UIView` Animations

`UIView` animation provides one of the odd but lovely perks of working with the iPhone as a development platform. It enables you to slow down changes when updating views, producing smooth animated results that enhance the user experience. Best of all, this all occurs without you having to do much work.

`UIView` animations are perfect for building a visual bridge between a view's current and changed states. With them, you emphasize visual change and create an animation that links those changes together. Animatable changes include the following:

- Changes in location—moving a view around the screen
- Changes in size—updating the view's frame
- Changes in transparency—altering the view's alpha value
- Changes in rotation or any other affine transforms that you apply to a view

Building `UIView` Animation Blocks

`UIView` animations work as blocks, a complete transaction that progresses at once. Start the block by issuing `beginAnimations:context:`. End the block with `commitAnimations`. These class methods are sent to `UIView` and not to individual views. In the block between these two calls, you define the way the animation works and perform the actual view updates. The animation controls you'll use are as follows:

- **`beginAnimations:context.`** Marks the start of the animation block.
- **`setAnimationCurve.`** Defines the way the animation accelerates and decelerates. Use ease-in/ease-out (`UIViewAnimationCurveEaseInOut`) unless you have some compelling reason to select another curve. The other curve types are ease in (accelerate into the animation), linear (no animation acceleration), and ease out (accelerate out of the animation). Ease-in/ease-out provides the most natural-feeling animation style.
- **`setAnimationDuration.`** Specifies the length of the animation, in seconds. This is really the cool bit. You can stretch out the animation for as long as you need it to run. Be aware of straining your user's patience and keep your animations below a second or two in length.
- **`commitAnimations.`** Marks the end of the animation block.

Sandwich your actual view change commands after setting up the animation details and before ending the animation. Listing 2-8 shows `UIView` animations in action by setting an animation curve and the animation duration (here, one second). The actual change being animated is a transparency update. The alpha value of the content view goes to zero, making it invisible. Instead of the view simply disappearing, this animation block slows down the change and fades it out of sight.

Note

Apple often uses two animation blocks one after another to add bounce to their animations. For example, they might zoom into a view a bit more than needed and then use a second animation to bring that enlarged view down to its final size. Use "bounces" to add a little more life to your animation blocks. Be sure that the animations do not overlap. Either add a delay so that the second animation does not start until the first ends (`performSelector: withObject: afterDelay:`) or assign an animation delegate callback (`animationDidStop: finished:`) to catch the end of the first animation and start the second.

Listing 2-8 **Using `UIView` Animation Calls**

```
[UIView beginAnimations:nil context:context];
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
[UIView setAnimationDuration:1.0];
[contentView setAlpha:0.0f];
[UIView commitAnimations];
```

Recipe: Fading a View In and Out

There are times you'll want to add information to your screen that overlays your view but does not of itself do anything. For example, you might show a top scores list or some instructions or provide a context-sensitive tool tip. Recipe 2-5 demonstrates how to use a `UIView` animation block to slowly fade a noninteractive overlay view into and out of sight.

This is done by creating a custom `ToggleView`. As defined by this code, `ToggleViews` are `UIViews` with one child, an image view. When tapped, the animation block toggles the alpha setting from off to on or on to off. The key bits for making this happen well and reliably are as follows:

- Make sure the child does not look for interaction events. Cocoa Touch does not allow transparent views to catch touches. So you must allow the parent, the `ToggleView`, to handle all user interactions instead. When creating the child, the method sets the child's property `userInteractionEnabled` to `NO`.

- Make sure to catch only mouse down events. For simple on-off-on-off toggles, catch and respond only to presses for the most natural user feedback. Otherwise, user taps will hide and then immediately show your image view again.
- Pick a reasonable animation time. If you lengthen the animation beyond what your user is willing to handle, you'll end up handling new taps before the first animation has completed. The one-second animation shown here is just about the longest time you'll want to use. Half- or quarter-second animations are better for common interface changes.

Recipe 2-5 **Using `UIView` Animations with Transparency Changes**

```
@interface ToggleView: UIView
{
    BOOL isVisible;
    UIImageView *imgView;
}
@end

@implementation ToggleView
- (id) initWithFrame: (CGRect) aFrame;
{
    self = [super initWithFrame:aFrame];
    isVisible = YES;
    imgView = [[UIImageView alloc] initWithFrame:[[UIScreen mainScreen]
    ➥applicationFrame]];
    [imgView setImage:[UIImage imageNamed:@"alphablend.png"]];
    imgView.userInteractionEnabled = NO;
    [self addSubview:imgView];
    [imgView release];
    return self;
}

- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // only respond to mouse down events
    UITouch *touch = [touches anyObject];
    if ([touch phase] != UITouchPhaseBegan) return;

    isVisible = !isVisible;

    CGContextRef context = UIGraphicsGetCurrentContext();
    [UIView beginAnimations:nil context:context];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    [UIView setAnimationDuration:1.0];
    [imgView setAlpha:(float)isVisible];
```

Recipe 2-5 **Continued**

```
    [UIView commitAnimations];
}
- (void) dealloc
{
    [imgView release];
    [super dealloc];
}
@end
```

Recipe: Swapping Views

The `UIView` animation block doesn't limit you to a single change. Recipe 2-6 combines frame size updates with transparency changes to create a more compelling animation. You do this by adding several directives at once to the animation block. Recipe 2-6 performs four actions at a time. It zooms and fades one view into place, while zooming out and fading away another. Figure 2-7 provides a preview of this animation in action.

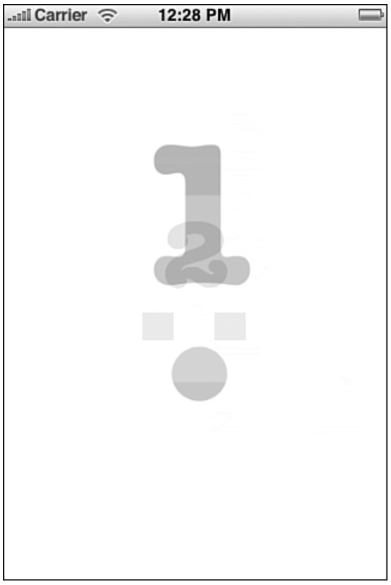

Figure 2-7 Issuing several view changes within a single `UIView` animation block can create complex visual effects.

Recipe 2-6 **Combining Multiple View Changes in Animation Blocks**

```
@interface ToggleView: UIView
{
      BOOL isOne;
      UIImageView *imgView1, *imgView2;
}
@end

@implementation ToggleView

#define BIGRECT CGRectMake(0.0f, 0.0f, 320.0f, 435.0f)
#define SMALLRECT CGRectMake(130.0f, 187.0f, 60.0f, 60.0f)

- (id) initWithFrame: (CGRect) aFrame;
{
    self = [super initWithFrame:aFrame];

    // Load both views, make them noninteractive
    imgView1 = [[UIImageView alloc] initWithFrame:BIGRECT];
    imgView2 = [[UIImageView alloc] initWithFrame:SMALLRECT];
    [imgView1 setImage:[UIImage imageNamed:@"one.png"]];
    [imgView2 setImage:[UIImage imageNamed:@"two.png"]];
    imgView1.userInteractionEnabled = NO;
    imgView2.userInteractionEnabled = NO;

    // image 1 is in front of image 2 to begin
    [self addSubview:imgView2];
    [self addSubview:imgView1];
    isOne = YES;
    [imgView1 release];
    [imgView2 release];
    return self;
}
- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Determine which view occupies which role
    UIImageView *big = isOne ? imgView1 : imgView2;
    UIImageView *little = isOne ? imgView2 : imgView1;
    isOne = !isOne;

    // Pack all the changes into the animation block
    CGContextRef context = UIGraphicsGetCurrentContext();
    [UIView beginAnimations:nil context:context];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    [UIView setAnimationDuration:1.0];

    [big setFrame:SMALLRECT];
```

Recipe 2-6 **Continued**

```
    [big setAlpha:0.5];
    [little setFrame:BIGRECT];
    [little setAlpha:1.0];

    [UIView commitAnimations];

    // Hide the shrunken "big" image.
    [big setAlpha:0.0f];
    [[big superview] bringSubviewToFront:big];
}
-(void) dealloc
{
      [imgView1 release];
      [imgView2 release];
      [super dealloc];
}
@end
```

Recipe: Flipping Views

Transitions enable you to extend your `UIView` animation blocks to add even more visual flair. Two transitions—`UIViewAnimationTransitionFlipFromLeft` and `UIViewAnimationTransitionFlipFromRight`—enable you to do just what their names suggest. At this time, you can flip views left or flip views right. These are the only two official transitions available for `UIViews`.

Note

During the SDK beta period, Apple promised additional animations that were never realized, specifically `UIViewAnimationTransitionCurlUp` and `UIViewAnimationTransitionCurlDown`. These extra animations may appear at some future time.

To use transitions in `UIView` animation blocks, you need to do two things. First, you must add the transition as a block parameter. Use `setAnimationTransition:` to assign the transition to the enclosing `UIView` animation block. Second, you should rearrange the view order while inside the block. This is best done with `exchangeSubviewAtIndex: withSubviewAtIndex:`. Recipe 2-7 demonstrates how to create a simple flip view using these techniques. When tapped, the views use the animation to flip from one side to the next, as shown in Figure 2-8.

Do not confuse the `UIView` animation blocks with the Core Animation `CATransition` class. Unfortunately, you cannot assign a `CATransition` to your `UIView` animation. To use a `CATransition`, you must apply it to a `UIView`'s layer, which is shown in the next recipe.

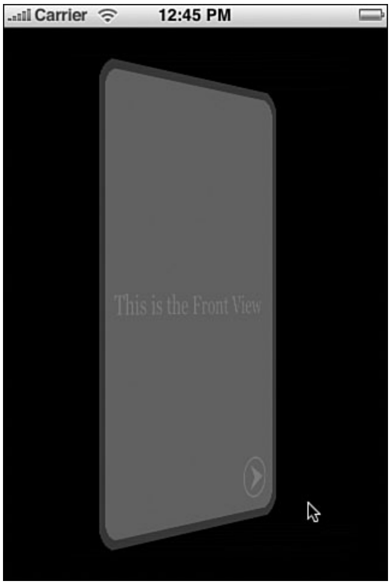

Figure 2-8 Use `UIView`'s built-in transition animations to flip your way from one view to the next.

Recipe 2-7 **Using Transitions with `UIView` Animation Blocks**

```
@interface FlipView : UIImageView
@end

@implementation FlipView
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Start Animation Block
    CGContextRef context = UIGraphicsGetCurrentContext();
    [UIView beginAnimations:nil context:context];
    [UIView setAnimationTransition: UIViewAnimationTransitionFlipFromLeft
    ➥forView:[self superview] cache:YES];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    [UIView setAnimationDuration:1.0];

    // Animations
    [[self superview] exchangeSubviewAtIndex:0 withSubviewAtIndex:1];

    // Commit Animation Block
    [UIView commitAnimations];
}
@end
```

Recipe: Applying `CATransitions` to Layers

Core Animation Transitions expand your `UIView` animation vocabulary with just a few small differences in implementation. `CATransitions` work on layers rather than on views. Layers are the Core Animation rendering surfaces associated with each `UIView`. When working with Core Animation, you apply `CATransitions` to a view's default layer (`[myView layer]`) rather than the view itself.

You don't set your parameters through `UIView` the way you do with `UIView` animation. You create a Core Animation object, set *its* parameters, and then add the parameterized transition to the layer. Listing 2-9 shows a simple `pushFromLeft` method that you might swap out for the `flip` method shown in Recipe 2-7.

Animations use both a *type* and a *subtype.* The type specifies the kind of transition used. The subtype sets its direction. Together the type and subtype tell how the views should act when you apply the animation to them.

Core Animation Transitions are distinct from the two `UIView` flips discussed in the previous recipe. Cocoa Touch offers four types of Core Animation. These available types include cross fades, pushes (used in Listing 2-9), reveals (where one view slides off another), and covers (where one view slides onto another). The last three types enable you to specify the direction of motion for the transition through subtypes. For obvious reasons, cross fades do not have a direction and they do not use subtypes.

Core Animation is part of the Quartz Core framework. To use this sample code, you must add the Quartz Core framework to your project and import `<QuartzCore/QuartzCore.h>` into your code.

Note

Apple's Core Animation features 2D and 3D routines built around Objective-C classes. These classes provide graphics rendering and animation for your iPhone and Macintosh applications. Core Animation avoids many low-level development details associated with, for example, direct OpenGL while retaining the simplicity of working with hierarchical views.

Listing 2-9 **Adding a Core Animation Transition to a `UIView` Layer**

```
@implementation PushView
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event
{

      CATransition *animation = [CATransition animation];
      [animation setDelegate:self];
      [animation setDuration:1.0f];
      [animation setTimingFunction:UIViewAnimationCurveEaseInOut];
      [animation setType: kCATransitionPush];
      [animation setSubtype: kCATransitionFromLeft];

      [[self superview] exchangeSubviewAtIndex:0 withSubviewAtIndex:1];
```

Listing 2-9 **Continued**

```
    [[[self superview] layer] addAnimation:animation
    ➥forKey:@"transitionViewAnimation"];
}
@end
```

Undocumented Animation Types

The iPhone actually implements more animation types than official documents would suggest. As Listing 2-10 shows, the iPhone is perfectly capable of handling map curls à la the Google Maps application. This code, which works on the iPhone but not the Simulator, relies on extracting animation names from the `UIKit` binary framework file.

Like all undocumented calls, this is not without risk. Apple may change or delete these animations at any time. Other animation types include `pageCurl`, `pageUnCurl`, `suckEffect`, `spewEffect`, `cameraIris` (from the Photos application), `cameraIrisHollowOpen`, `cameraIrisHollowClose`, `genieEffect` (typically used for deleting garbage), `unGenieEffect`, `rippleEffect`, `twist`, `tubey`, `swirl`, `charminUltra`, `zoomyIn`, `zoomyOut`, and `oglFlip`.

Note the use of `setRemovedOnCompletion: NO`. This freezes the animation at its end, allowing the curled map to remain visible, as shown in Figure 2-9.

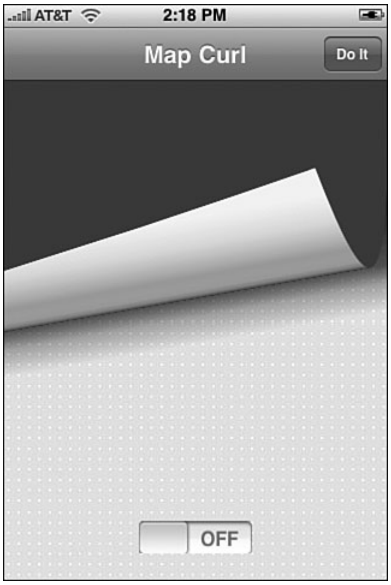

Figure 2-9 This eye-catching effect uses an undocumented Core Animation type called `mapCurl`.

Listing 2-10 **Calling Undocumented Animation Types**

```
- (void) performCurl
{
    // Curl the image up or down
    CATransition *animation = [CATransition animation];
    [animation setDelegate:self];
    [animation setDuration:1.0f];
    [animation setTimingFunction:UIViewAnimationCurveEaseInOut];
    [animation setType:(notCurled ? @"mapCurl" : @"mapUnCurl")];
    [animation setRemovedOnCompletion:NO];
    [animation setFillMode: @"extended"];
    [animation setRemovedOnCompletion: NO];
    notCurled = !notCurled;

    [[topView layer] addAnimation:animation forKey:@"pageFlipAnimation"];
}
```

General Core Animation Calls

The iPhone provides partial support for Core Animation calls. By *partial,* I mean that many standard classes are missing in action. `CIFilter` is one such class. It's not included in Cocoa Touch, although the `CALayer` and `CATransition` classes are both filter-aware. If you're willing to work through these limits, you can freely use standard Core Animation calls in your programs.

Listing 2-11 shows iPhone native Core Animation code based on a sample from Lucas Newman (http://lucasnewman.com). When run, this method scales down and fades away the contents of a `UIImageView`. The source adds a translucent reflection layer, which follows the view.

This code remains virtually unchanged from the Mac OS X sample it was based on. More complex Core Animation samples may offer porting challenges, but for simple reflections, shadows, and transforms, all the functionality you need can be had at the native iPhone level.

Listing 2-11 **Native iPhone Core Animation Calls**

```
// Adapted from http://lucasnewman.com/animationsamples.zip
- (void) scaleAndFade
{
    // create the reflection layer
    CALayer *reflectionLayer = [CALayer layer];

    // share the contents image with the screen layer
    reflectionLayer.contents = [contentView layer].contents;
    reflectionLayer.opacity = 0.4;
    reflectionLayer.frame = CGRectOffset([contentView layer].frame, 0.5,
    ➥416.0f + 0.5);
```

Listing 2-11 **Continued**

```
    // flip the y-axis
    reflectionLayer.transform = CATransform3DMakeScale(1.0, -1.0, 1.0);
    reflectionLayer.sublayerTransform = reflectionLayer.transform;
    [[contentView layer] addSublayer:reflectionLayer];

#define ANIMATION_DURATION (4.0)

    [CATransaction begin];
    [CATransaction setValue:[NSNumber numberWithFloat:ANIMATION_DURATION]
    ➥forKey:kCATransactionAnimationDuration];

    // scale it down
    CABasicAnimation *shrinkAnimation = [CABasicAnimation
    ➥animationWithKeyPath:@"transform.scale"];
    shrinkAnimation.timingFunction = [CAMediaTimingFunction
    ➥functionWithName:kCAMediaTimingFunctionEaseIn];
    shrinkAnimation.toValue = [NSNumber numberWithFloat:0.0];
    [[contentView layer] addAnimation:shrinkAnimation forKey:@"shrinkAnimation"];

    // fade it out
    CABasicAnimation *fadeAnimation = [CABasicAnimation
    ➥animationWithKeyPath:@"opacity"];
    fadeAnimation.toValue = [NSNumber numberWithFloat:0.0];
    fadeAnimation.timingFunction = [CAMediaTimingFunction functionWithName:
    ➥kCAMediaTimingFunctionEaseIn];
    [[contentView layer] addAnimation:fadeAnimation forKey:@"fadeAnimation"];

    [CATransaction commit];
}
```

Recipe: Swiping Views

Swipes are a convenient but often-overlooked iPhone interaction style. When a user quickly drags his or her finger across the screen, the `UITouch` objects returned for that gesture include an `info` property. This property defines the direction in which the user swiped the screen, (for example, up, down, left, or right). This behavior is best seen in the iPhone's Photos application, when users swipe left or right to move between album pictures.

Early versions of the iPhone SDK offered swipe detection as a standard part of the `UITouch` object, but later releases dropped that capability. Instead, Apple offered workaround code in its *iPhone Developers Guide.* Recipe 2-8 is based on that code. It ensures that a user continues finger movement in one direction by defining a safety zone around the movement. If the user strays diagonally more than 6 pixels off course, the swipe cancels. Stay on-course for at least 12 pixels and the swipe is set.

Recipe 2-8 applies a Core Animation Transition on completion of a successful swipe. It uses the swipe direction to set the animation's subtype. Subtypes are used in Core Animation to specify the overall movement of the animation, whether up, down, or sideways.

This sample mimics the interaction style used for browsing through album pictures in Photos but allows you to move up and down as well as left and right. If you comment out the `kCATransitionPush` animation type and replace it with the undocumented `oglFlip` in the line that immediately follows it, you'll receive an even nicer surprise. Far from being limited to the two core flip directions, the iPhone actually supports a full four-way flip style, albeit one that Apple has not included in its public SDK.

Note

In early releases of the iPhone SDK, swipes didn't work in the Simulator. In later versions, they did. Should you encounter platform limitations while developing (for example, when working with the Camera), you can easily add workarounds based on testing the platform. Add compiler directives such as `#if defined(TARGET_IPHONE_SIMULATOR)` to your source.

Recipe 2-8 **Detecting and Responding to User Swipes in Your Views**

```
- (CATransition *) getAnimation:(NSString *) direction
{
    CATransition *animation = [CATransition animation];
    [animation setDelegate:self];
    [animation setType:kCATransitionPush];
    // [animation setType:@"oglFlip"];
    [animation setSubtype:direction];
    [animation setDuration:1.0f];
    [animation setTimingFunction:[CAMediaTimingFunction
    ➥functionWithName:kCAMediaTimingFunctionEaseInEaseOut]];
    return animation;
}

#define HORIZ_SWIPE_DRAG_MIN 12
#define VERT_SWIPE_DRAG_MAX 4

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    startTouchPosition = [touch locationInView:self];
    dirString = NULL;
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
```

Recipe 2-8 **Continued**

```
    UITouch *touch = touches.anyObject;
    CGPoint currentTouchPosition = [touch locationInView:self];


    if (fabsf(startTouchPosition.x - currentTouchPosition.x) >=
        HORIZ_SWIPE_DRAG_MIN &&
        fabsf(startTouchPosition.y - currentTouchPosition.y) <=
        VERT_SWIPE_DRAG_MAX)
    {
        // Horizontal Swipe
        if (startTouchPosition.x < currentTouchPosition.x) {
            dirString = kCATransitionFromLeft;
        }
        else
            dirString = kCATransitionFromRight;
    }
    else if (fabsf(startTouchPosition.y - currentTouchPosition.y) >=
          HORIZ_SWIPE_DRAG_MIN &&
          fabsf(startTouchPosition.x - currentTouchPosition.x) <=
          VERT_SWIPE_DRAG_MAX)
    {
        // Vertical Swipe
        if (startTouchPosition.y < currentTouchPosition.y)
            dirString = kCATransitionFromBottom;
        else
            dirString = kCATransitionFromTop;
    } else
    {
        // Process a non-swipe event.
        // dirString = NULL;
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (dirString)
    {
        CATransition *animation = [self getAnimation:dirString];
        [[self superview] exchangeSubviewAtIndex:0 withSubviewAtIndex:1];
        [[[self superview] layer] addAnimation:animation forKey:kAnimationKey];

    }
}
```

Recipe: Transforming Views

Affine transforms enable you to change an object's geometry by mapping that object from one view coordinate system into another. The iPhone SDK fully supports standard affine 2D transforms. With them, you can scale, translate, rotate, and skew your views however your heart desires and your application demands.

Transforms are defined in Core Graphics, and consist of calls such as `CGAffineTransformMakeRotation` and `CGAffineTransformScale`. These build and modify the 3-by-3 transform matrices. Once built, use `UIView`'s `setTransform:` call to apply 2D affine transformations to `UIView` objects.

Recipe 2-9 demonstrates how to build and apply an affine transform of a `UIView`. To create the sample, I kept things simple. I build an `NSTimer` that ticks every 1/30th of a second. On ticking, it rotates a view by 1% of pi and scales over a cosine curve. I use the cosine's absolute value for two reasons. It keeps the view visible at all times, and it provides a nice bounce effect when the scaling changes direction. This produces a rotating and undamped bounce animation.

This is one of those samples that it's best to build and view as you read through the code. You'll be better able to see how the `handleTimer:` method correlates to the visual effects you're looking at.

Recipe 2-9 **Example of an Affine Transform of a `UIView`**

```
#import "math.h"
#define PI 3.14159265

@interface HelloController : UIViewController
{
    UIView *contentView;
    UIImageView *rotateView;
    int theta;
}
@end

@implementation HelloController
- (id)init
{
    if (self = [super init]) self.title = @"Affine Demo";
    return self;
}

- (void) handleTimer: (NSTimer *) timer
{
    // Rotate each iteration by 1% of PI
    float angle = theta * (PI / 100);
    CGAffineTransform transform = CGAffineTransformMakeRotation(angle);
    theta = (theta + 1) % 200;
```

Recipe 2-9 **Continued**

```
    // For fun, scale by the absolute value of the cosine
    float degree = cos(angle);
    if (degree < 0.0) degree *= -1.0f;
    degree += 0.5f;
    CGAffineTransform scaled = CGAffineTransformScale(transform, degree, degree);

    // Apply the affine transform
    [rotateView setTransform:scaled];
}

- (void)loadView
{
    theta = 0;

    contentView = [[UIView alloc] initWithFrame:[[UIScreen mainScreen]
    ➥applicationFrame]];
    rotateView = [[UIImageView alloc] initWithFrame:CGRectMake(0.0f, 0.0f, 240.0f,
    ➥240.0f)];
    [rotateView setImage:[UIImage imageNamed:@"rotateart.png"]];
    [rotateView setCenter:CGPointMake(160.0f, 208.0f)];
    [contentView addSubview:rotateView];
    [rotateView release];

    self.view = contentView;
    [contentView release];

    [NSTimer scheduledTimerWithTimeInterval: 0.03f target: self selector:
    ➥@selector(handleTimer:)
                                   userInfo: nil repeats: YES];
}

// Allow the view to respond to iPhone Orientation changes
-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation
{
    return NO;
}
-(void) dealloc
{
    [contentView release];
    [rotateView release];
    [super dealloc];
}
@end
```

Centering Landscape Views

Use the same affine transform approach to center landscape-oriented views. Listing 2-12 creates a 480-by-320 pixel view, centers it at [160, 240] (using portrait view coordinates), and then rotates it into place. Half of pi corresponds to 90 degrees, creating a landscape-right rotation. Centering keeps the entire view onscreen. All subviews, including text fields, labels, switches, and so on rotate into place along with the parent view.

If you want to work with a landscape keyboard for this view, make sure to call `[[UIApplication sharedApplication] setStatusBarOrientation: UIInterfaceOrientationLandscapeRight]`. This sets the status bar orientation, which controls the keyboard regardless of whether the status bar is hidden or shown.

Listing 2-12 **Rotating Landscape Views into Place**

```
@implementation HelloController
- (void)loadView
{
    contentView = [[UIView alloc] initWithFrame:CGRectMake(0.0f, 0.0f, 480.0f,
    ➥320.0f)];
    [contentView setCenter:CGPointMake(160.0f, 240.0f)];
    [contentView setBackgroundColor:[UIColor blackColor]];
    [contentView setTransform:CGAffineTransformMakeRotation(3.141592f / 2.0f)];
    self.view = contentView;
    [contentView release];
}

-(void) dealloc
{
    [contentView release];
    [super dealloc];
}
@end
```

Summary

`UIViews` provide the onscreen components your users see and interact with. As this chapter has shown, even in their most basic form, they offer incredible flexibility and power. You've discovered how to use views to build up elements on a screen, create multiple interaction objects, and introduce eye-catching animation. Here's a collection of thoughts about the recipes you've seen in this chapter that you might want to ponder before moving on:

- When dealing with multiple onscreen views, hierarchy should always remain front-most in your mind—no pun! Use your view hierarchy vocabulary (`bringSubviewToFront:`, `sendSubviewToBack:`, `exchangeSubviewAtIndex:withSubviewAtIndex:`) to take charge of your views and always present the proper visual context to your users.

- You're not limited to rectangles. Use `UIView / Core` Graphics clipping to create compelling interaction objects that don't necessarily have right corners.
- Be concrete. The iPhone has a perfectly good touch screen. Why not let your users drag items around the screen with their fingers? It adds to the reality and the platform's interactive nature.
- Animate everything. Animations don't have to be loud, splashy, or bad design. The iPhone's strong animation support enables you to add smooth transitions between user tasks. Short, smooth, focused changes are the iPhone's bread and butter.
- Users typically have five fingers per hand. Don't limit yourself to a one-finger interface when it makes sense to expand your interaction into multitouch territory.
- A solid grounding in Quartz graphics and Core Animation will be your friend. Using `drawRect:`, you can build any kind of custom `UIView` presentation you'd like, including text, Bézier curves, scribbles, and so forth.
- Explore! This chapter has only touched lightly on the ways you can use `UIViews` in your applications. Use this material as a jumping-off point to explore the full vocabulary of the `UIView` class.