Drawing and Printing Guide for iOS

Developer

Contents

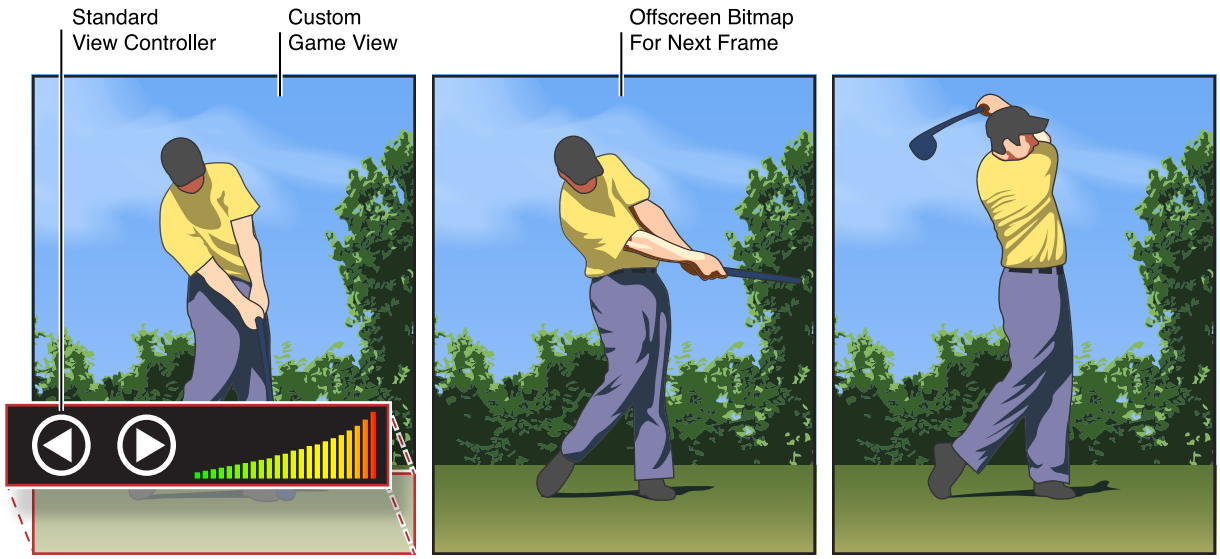
Figures, Tables, and Listings

About Drawing and Printing in iOS

This document covers three related subjects:

- Drawing custom UI views. Custom UI views allow you to draw content that cannot easily be drawn with standard UI elements. For example, a drawing program might use a custom view for the user's drawing, or an arcade game might use a custom view into which it draws sprites.
- Drawing into offscreen bitmap and PDF content. Whether you plan to display the images later, export them to a file, or print the images to an AirPrint-enabled printer, offscreen drawing lets you do so without interrupting the user's workflow.
- Adding AirPrint support to your app. The iOS printing system lets you draw your content differently to fit on the page.

Figure I-1 You can combine custom views with standard views, and even draw things offscreen.



At a Glance

The iOS native graphics system combines three major technologies: UIKit, Core Graphics, and Core Animation. UIKit provides views and some high-level drawing functionality within those views, Core Graphics provides additional (lower-level) drawing support within UIKit views, and Core Animation provides the ability to apply transformations and animation to UIKit views. Core Animation is also responsible for view compositing.

Custom UI Views Allow Greater Drawing Flexibility

This document describes how to draw into custom UI views using native drawing technologies. These technologies, which include the Core Graphics and UIKit frameworks, support 2D drawing.

Before you consider using a custom UI view, you should make certain that you really need to do so. Native drawing is suitable for handling more complex 2D layout needs. However, because custom views are processor-intensive, you should limit the amount of drawing you do using native drawing technologies.

As an alternative to custom drawing, an iOS app can draw things onscreen in several other ways.

- **Using standard (built-in) views.** Standard views let you draw common user-interface primitives, including lists, collections, alerts, images, progress bars, tables, and so on without the need to explicitly draw anything yourself. Using built-in views not only ensures a consistent user experience between iOS apps, but also saves you programming effort. If built-in views meet your needs, you should read *View Programming Guide for iOS*.
- **Using Core Animation layers.** Core Animation lets you create complex, layered 2D views with animation and transformations. Core Animation is a good choice for animating standard views, or for combining views in complex ways to present the illusion of depth, and can be combined with custom-drawn views as described in this document. To learn more about Core Animation, read *Core Animation Overview*.
- **Using OpenGL ES in a GLKit view or a custom view.** The OpenGL ES framework provides a set of open-standard graphics libraries geared primarily toward game development or apps that require high frame rates, such as virtual prototyping apps and mechanical and architectural design apps. It conforms to the OpenGL ES 2.0 and OpenGL ES v1.1 specifications. To learn more about OpenGL drawing, read *OpenGL ES Programming Guide for iOS*.
- **Using web content.** The `UIWebView` class lets you display web-based user interfaces in an iOS app. To learn more about displaying web content in a web view, read *Using UIWebView to display select document types* and *UIWebView Class Reference*.

Depending on the type of app you are creating, it may be possible to use little or no custom drawing code. Although immersive apps typically make extensive use of custom drawing code, utility and productivity apps can often use standard views and controls to display their content.

The use of custom drawing code should be limited to situations where the content you display needs to change dynamically. For example, a drawing app typically needs to use custom drawing code to track the user's drawing commands, and an arcade-style game may need to update the screen constantly to reflect the changing game environment. In those situations, you should choose an appropriate drawing technology and create a custom view class to handle events and update the display appropriately.

On the other hand, if the bulk of your app's interface is fixed, you can render the interface in advance to one or more image files and display those images at runtime using the `UIImageView` class. You can layer image views with other content as needed to build your interface. You can also use the `UILabel` class to display configurable text and include buttons or other controls to provide interactivity. For example, an electronic version of a board game can often be created with little or no custom drawing code.

Because custom views are generally more processor-intensive (with less help from the GPU), if you can do what you need to do using standard views, you should always do so. Also, you should make your custom views as small as possible, containing only content that you cannot draw in any other way, use use standard views for everything else. If you need to combine standard UI elements with custom drawing, consider using a Core Animation layer to superimpose a custom view with a standard view so that you draw as little as possible.

A Few Key Concepts Underpin Drawing With the Native Technologies

When you draw content with UIKit and Core Graphics, you should be familiar with a few concepts in addition to the view drawing cycle.

- For the `drawRect:` method, UIKit creates a **graphics context** for rendering to the display. This graphics context contains the information the drawing system needs to perform drawing commands, including attributes such as fill and stroke color, the font, the clipping area, and line width. You can also create and draw into custom graphics context for bitmap images and PDF content.
- UIKit has a **default coordinate system** where the origin of drawing is at the top-left of a view; positive values extend downward and to the right of that origin. You can change the size, orientation, and position of the default coordinate system relative to the underlying view or window by modifying the current transformation matrix, which maps a view's coordinate space to the device screen.
- In iOS, the **logical coordinate space**, which measures distances in points, is not equal to the device coordinate space, which measures in pixels. For greater precision, points are expressed in floating-point values.

Relevant Chapter: "iOS Drawing Concepts" (page 12)

UIKit, Core Graphics, and Core Animation Give Your App Many Tools For Drawing

The UIKit and Core Graphics have many complementary graphics capabilities that encompass graphics contexts, Bézier paths, images, bitmaps, transparency layers, colors, fonts, PDF content, and drawing rectangles and clipping areas. In addition, Core Graphics has functions related to line attributes, color spaces, pattern colors, gradients, shadings, and image masks. The Core Animation framework enables you to create fluid animations by manipulating and displaying content created with other technologies.

Relevant Chapters: "iOS Drawing Concepts" (page 12), "Drawing Shapes Using Bézier Paths" (page 30), "Drawing and Creating Images" (page 40), "Generating PDF Content" (page 45)

Apps Can Draw Into Offscreen Bitmaps or PDFs

It is often useful for an app to draw content offscreen:

- Offscreen bitmap contexts are often used when scaling down photographs for upload, rendering content into an image file for storage purposes, or using Core Graphics to generate complex images for display.
- Offscreen PDF contexts are often used when drawing user-generated content for printing purposes.

After you create an offscreen context, you can draw into it just as you would draw within the `drawRect:` method of a custom view.

Relevant Chapters: "Drawing and Creating Images" (page 40), "Generating PDF Content" (page 45)

Apps Have a Range of Options for Printing Content

As of iOS 4.2, apps can print content wirelessly to supported printers using AirPrint. When assembling a print job, they have three ways to give UIKit the content to print:

- They can give the framework one or more objects that are directly printable; such objects require minimal app involvement. These are instances of the `NSData`, `NSURL`, `UIImage`, or `ALAsset` classes containing or referencing image data or PDF content.
- They can assign a print formatter to the print job. A print formatter is an object that can lay out content of a certain type (such as plain text or HTML) over multiple pages.
- They can assign a page renderer to the print job. A page renderer is usually an instance of a custom subclass of `UIPrintPageRenderer` that draws the content to be printed in part or in full. A page renderer can use one or more print formatters to help it draw and format its printable content.

Relevant Chapter: "Printing" (page 52)

It's Easy to Update Your App for High-Resolution Screens

Some iOS devices feature high-resolution screens, so your app must be prepared to run on these devices and on devices with lower-resolution screens. iOS handles much of the work required to handle the different resolutions, but your app must do the rest. Your tasks include providing specially named high-resolution images and modifying your layer- and image-related code to take the current scale factor into account.

Relevant Appendix: "Supporting High-Resolution Screens In Views" (page 82)

See Also

For complete examples of printing, see the *PrintPhoto*, *Recipes and Printing*, and *UIKit Printing with UIPrintInteractionController and UIViewPrintFormatter* sample code projects.

iOS Drawing Concepts

High-quality graphics are an important part of your app's user interface. Providing high-quality graphics not only makes your app look good, but it also makes your app look like a natural extension to the rest of the system. iOS provides two primary paths for creating high-quality graphics in your system: OpenGL or native rendering using Quartz, Core Animation, and UIKit. This document describes native rendering. (To learn about OpenGL drawing, see *OpenGL ES Programming Guide for iOS* .)

Quartz is the main drawing interface, providing support for path-based drawing, anti-aliased rendering, gradient fill patterns, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. UIKit provides Objective-C wrappers for line art, Quartz images, and color manipulations. Core Animation provides the underlying support for animating changes in many UIKit view properties and can also be used to implement custom animations.

This chapter provides an overview of the drawing process for iOS apps, along with specific drawing techniques for each of the supported drawing technologies. You will also find tips and guidance on how to optimize your drawing code for the iOS platform.

Important: Not all UIKit classes are thread safe. Be sure to check the documentation before performing drawing-related operations on threads other than your app's main thread.

The UIKit Graphics System

In iOS, all drawing to the screen—regardless of whether it involves OpenGL, Quartz, UIKit, or Core Animation—occurs within the confines of an instance of the `UIView` class or a subclass thereof. Views define the portion of the screen in which drawing occurs. If you use system-provided views, this drawing is handled for you automatically. If you define custom views, however, you must provide the drawing code yourself. If you use Quartz, Core Animation, and UIKit to draw, you use the drawing concepts described in the following sections.

In addition to drawing directly to the screen, UIKit also allows you to draw into offscreen bitmap and PDF graphics contexts. When you draw in an offscreen context, you are not drawing in a view, which means that concepts such as the view drawing cycle do not apply (unless you then obtain that image and draw it in an image view or similar).

The View Drawing Cycle

The basic drawing model for subclasses of the `UIView` class involves updating content on demand. The `UIView` class makes the update process easier and more efficient; however, by gathering the update requests you make and delivering them to your drawing code at the most appropriate time.

When a view is first shown or when a portion of the view needs to be redrawn, iOS asks the view to draw its content by calling the view's `drawRect:` method.

There are several actions that can trigger a view update:

- Moving or removing another view that was partially obscuring your view
- Making a previously hidden view visible again by setting its `hidden` property to `NO`
- Scrolling a view off of the screen and then back onto the screen
- Explicitly calling the `setNeedsDisplay` or `setNeedsDisplayInRect:` method of your view

System views are redrawn automatically. For custom views, you must override the `drawRect:` method and perform all your drawing inside it. Inside your `drawRect:` method, use the native drawing technologies to draw shapes, text, images, gradients, or any other visual content you want. The first time your view becomes visible, iOS passes a rectangle to the view's `drawRect:` method that contains your view's entire visible area. During subsequent calls, the rectangle includes only the portion of the view that actually needs to be redrawn. For maximum performance, you should redraw only affected content.

After calling your `drawRect:` method, the view marks itself as updated and waits for new actions to arrive and trigger another update cycle. If your view displays static content, then all you need to do is respond to changes in your view's visibility caused by scrolling and the presence of other views.

If you want to change the contents of the view, however, you must tell your view to redraw its contents. To do this, call the `setNeedsDisplay` or `setNeedsDisplayInRect:` method to trigger an update. For example, if you were updating content several times a second, you might want to set up a timer to update your view. You might also update your view in response to user interactions or the creation of new content in your view.

> **Important:** Do not call your view's `drawRect:` method yourself. That method should be called *only* by code built into iOS during a screen repaint. At other times, no graphics context exists, so drawing is not possible. (Graphics contexts are explained in the next section.)

Coordinate Systems and Drawing in iOS

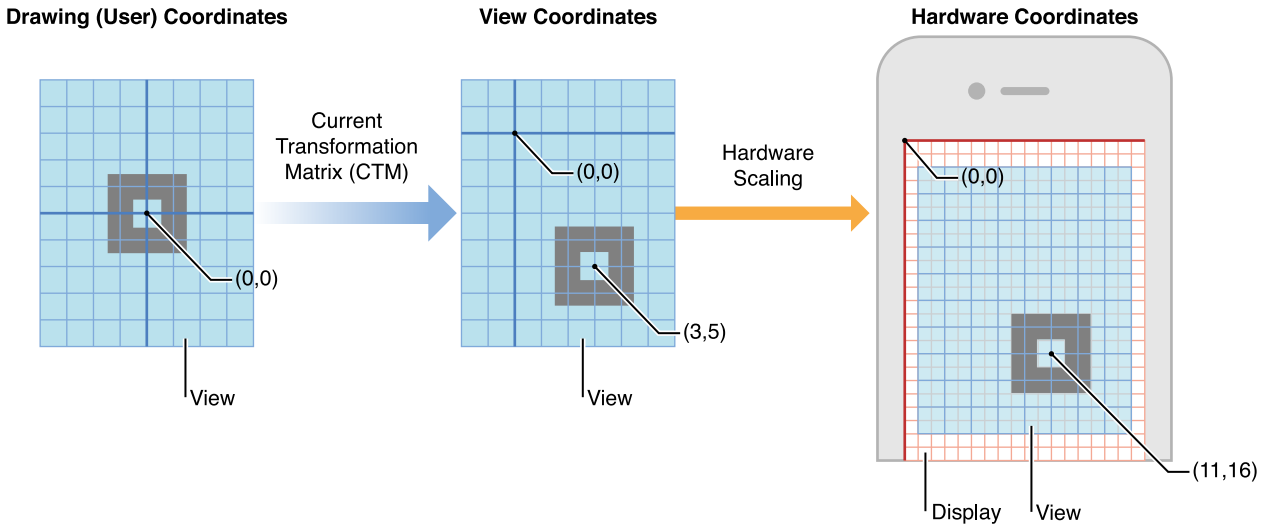
When an app draws something in iOS, it has to locate the drawn content in a two-dimensional space defined by a coordinate system. This notion might seem straightforward at first glance, but it isn't. Apps in iOS sometimes have to deal with different coordinate systems when drawing.

In iOS, all drawing occurs in a graphics context. Conceptually, a graphics context is an object that describes where and how drawing should occur, including basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and so on.

In addition, as shown in Figure 1-1 (page 14), each graphics context has a coordinate system. More precisely, each graphics context has three coordinate systems:

- The drawing (user) coordinate system. This coordinate system is used when you issue drawing commands.
- The view coordinate system (base space). This coordinate system is a fixed coordinate system relative to the view.
- The (physical) device coordinate system. This coordinate system represents pixels on the physical screen.

Figure 1-1 The relationship between drawing coordinates, view coordinates, and hardware coordinates



The drawing frameworks of iOS create graphics contexts for drawing to specific destinations—the screen, bitmaps, PDF content, and so on—and these graphics contexts establish the initial drawing coordinate system for that destination. This initial drawing coordinate system is known as the **default coordinate system**, and is a 1:1 mapping onto the view's underlying coordinate system.

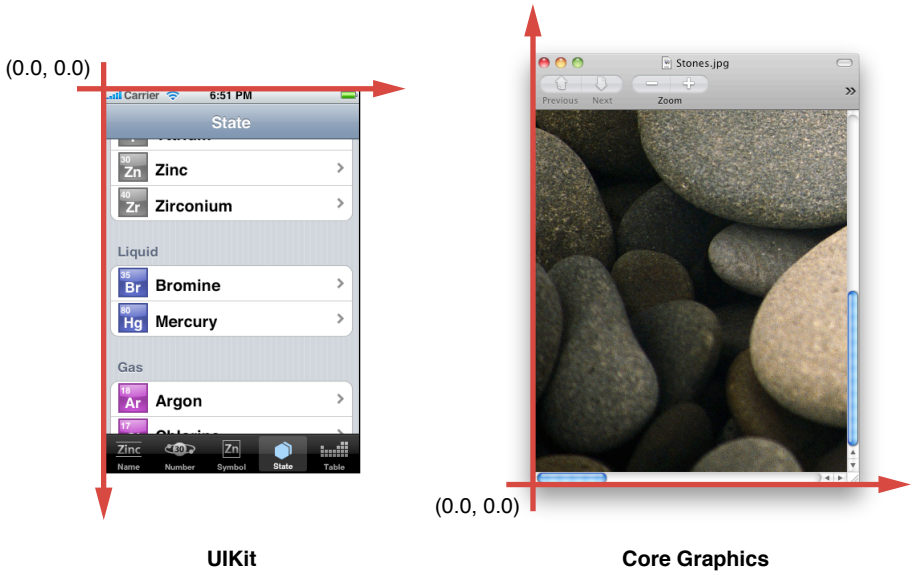
Each view also has a **current transformation matrix (CTM)**, a mathematical matrix that maps the points in the current drawing coordinate system to the (fixed) view coordinate system. The app can modify this matrix (as described later) to change the behavior of future drawing operations.

Each of the drawing frameworks of iOS establishes a default coordinate system based on the current graphics context. In iOS, there are two main types of coordinate systems:

- An upper-left-origin coordinate system (ULO), in which the origin of drawing operations is at the upper-left corner of the drawing area, with positive values extending downward and to the right. The default coordinate system used by the UIKit and Core Animation frameworks is ULO-based.
- A lower-left-origin coordinate system (LLO), in which the origin of drawing operations is at the lower-left corner of the drawing area, with positive values extending upward and to the right. The default coordinate system used by Core Graphics framework is LLO-based.

These coordinate systems are shown in Figure 1-2.

Figure 1-2 Default coordinate systems in iOS



Note: The default coordinate system in OS X is LLO-based. Although the drawing functions and methods of the Core Graphics and AppKit frameworks are perfectly suited to this default coordinate system, AppKit provides programmatic support for flipping the drawing coordinate system to have an upper-left origin.

Before calling your view's `drawRect:` method, UIKit establishes the default coordinate system for drawing to the screen by making a graphics context available for drawing operations. Within a view's `drawRect:` method, an app can set graphics-state parameters (such as fill color) and draw to the current graphics context without needing to refer to the graphics context explicitly. This implicit graphics context establishes a ULO default coordinate system.

Points Versus Pixels

In iOS there is a distinction between the coordinates you specify in your drawing code and the pixels of the underlying device. When using native drawing technologies such as Quartz, UIKit, and Core Animation, the drawing coordinate space and the view's coordinate space are both **logical coordinate spaces**, with distances measured in **points**. These logical coordinate systems are decoupled from the device coordinate space used by the system frameworks to manage the pixels onscreen.

The system automatically maps points in the view's coordinate space to pixels in the device coordinate space, but this mapping is not always one-to-one. This behavior leads to an important fact that you should always remember:

One point does not necessarily correspond to one physical pixel.

The purpose of using points (and the logical coordinate system) is to provide a consistent size of output that is device independent. For most purposes, the actual size of a point is irrelevant. The goal of points is to provide a relatively consistent scale that you can use in your code to specify the size and position of views and rendered content. How points are actually mapped to pixels is a detail that is handled by the system frameworks. For example, on a device with a high-resolution screen, a line that is one point wide may actually result in a line that is two physical pixels wide. The result is that if you draw the same content on two similar devices, with only one of them having a high-resolution screen, the content appears to be about the same size on both devices.

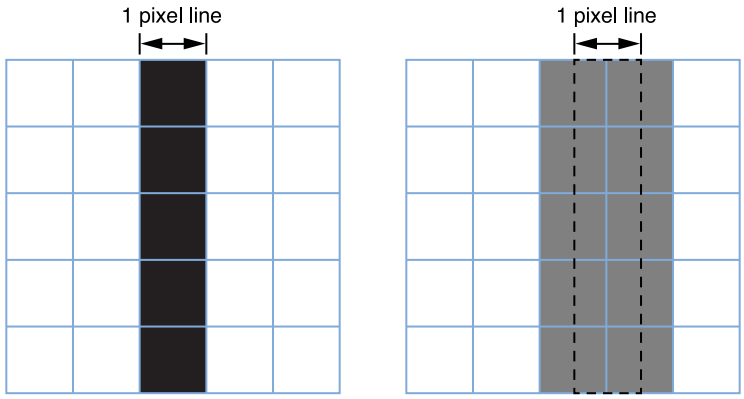
In iOS, the `UIScreen`, `UIView`, `UIImage`, and `CALayer` classes provide properties to obtain (and, in some cases, set) a **scale factor** that describes the relationship between points and pixels for that particular object. For example, every UIKit view has a `contentScaleFactor` property. On a standard-resolution screen, the scale factor is typically 1.0. On a high-resolution screen, the scale factor is typically 2.0. In the future, other scale factors may also be possible. (In iOS prior to version 4, you should assume a scale factor of 1.0.)

Native drawing technologies, such as Core Graphics, take the current scale factor into account for you. For example, if one of your views implements a `drawRect:` method, UIKit automatically sets the scale factor for that view to the screen's scale factor. In addition, UIKit automatically modifies the current transformation matrix of any graphics contexts used during drawing to take into account the view's scale factor. Thus, any content you draw in your `drawRect:` method is scaled appropriately for the underlying device's screen.

Because of this automatic mapping, when writing drawing code, pixels *usually* don't matter. However, there are times when you might need to change your app's drawing behavior depending on how points are mapped to pixels—to download higher-resolution images on devices with high-resolution screens or to avoid scaling artifacts when drawing on a low-resolution screen, for example.

In iOS, when you draw things onscreen, the graphics subsystem uses a technique called antialiasing to approximate a higher-resolution image on a lower-resolution screen. The best way to explain this technique is by example. When you draw a black vertical line on a solid white background, if that line falls exactly on a pixel, it appears as a series of black pixels in a field of white. If it appears exactly between two pixels, however, it appears as two grey pixels side-by-side, as shown in Figure 1-3.

Figure 1-3 A one-point line centered at a whole-numbered point value

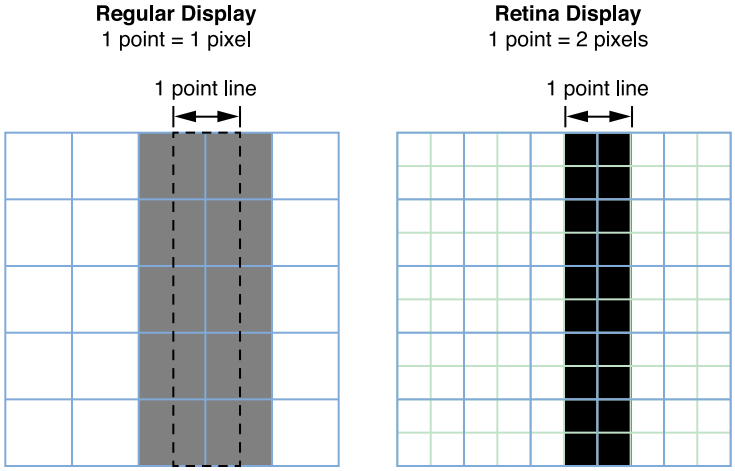


Positions defined by whole-numbered points fall at the midpoint between pixels. For example, if you draw a one-pixel-wide vertical line from (1.0, 1.0) to (10.0, 10.0), you get a fuzzy grey line. If you draw a two-pixel-wide line, you get a solid black line because it fully covers two pixels (one on either side of the specified point). As a rule, lines that are an odd number of physical pixels wide appear softer than lines with widths measured in even numbers of physical pixels unless you adjust their position to make them cover pixels fully.

Where the scale factor comes into play is when determining how many pixels are covered by a one-point-wide line.

On a low-resolution display (with a scale factor of 1.0), a one-point-wide line is one pixel wide. To avoid antialiasing when you draw a one-point-wide horizontal or vertical line, if the line is an odd number of pixels in width, you must offset the position by 0.5 points to either side of a whole-numbered position. If the line is an even number of points in width, to avoid a fuzzy line, you must *not* do so.

Figure 1-4 Appearance of one-point-wide lines on standard and retina displays



On a high-resolution display (with a scale factor of 2.0), a line that is one point wide is not antialiased at all because it occupies two full pixels (from -0.5 to +0.5). To draw a line that covers only a single physical pixel, you would need to make it 0.5 points in thickness and offset its position by 0.25 points. A comparison between the two types of screens is shown in Figure 1-4.

Of course, changing drawing characteristics based on scale factor may have unexpected consequences. A 1-pixel-wide line might look nice on some devices but on a high-resolution device might be so thin that it is difficult to see clearly. It is up to you to determine whether to make such a change.

Obtaining Graphics Contexts

Most of the time, graphics contexts are configured for you. Each view object automatically creates a graphics context so that your code can start drawing immediately as soon as your custom `drawRect:` method is called. As part of this configuration, the underlying `UIView` class creates a graphics context (a `CGContextRef` opaque type) for the current drawing environment.

If you want to draw somewhere other than your view (for example, to capture a series of drawing operations in a PDF or bitmap file), or if you need to call Core Graphics functions that require a context object, you must take additional steps to obtain a graphics context object. The sections below explain how.

For more information about graphics contexts, modifying the graphics state information, and using graphics contexts to create custom content, see *Quartz 2D Programming Guide*. For a list of functions used in conjunction with graphics contexts, see *CGContext Reference*, *CGBitmapContext Reference*, and *CGPDFContext Reference*.

Drawing to the Screen

If you use Core Graphics functions to draw to a view, either in the `drawRect:` method or elsewhere, you'll need a graphics context for drawing. (The first parameter of many of these functions must be a `CGContextRef` object.) You can call the function `UIGraphicsGetCurrentContext` to get an explicit version of the same graphics context that's made implicit in `drawRect:`. Because it's the same graphics context, the drawing functions should also make reference to a ULO default coordinate system.

If you want to use Core Graphics functions to draw in a UIKit view, you should use the ULO coordinate system of UIKit for drawing operations. Alternatively, you can apply a flip transform to the CTM and then draw an object in the UIKit view using Core Graphics native LLO coordinate system. "Flipping the Default Coordinate System" (page 25) discusses flip transforms in detail.

The `UIGraphicsGetCurrentContext` function always returns the graphics context currently in effect. For example, if you create a PDF context and then call `UIGraphicsGetCurrentContext`, you'd receive that PDF context. You must use the graphics context returned by `UIGraphicsGetCurrentContext` if you use Core Graphics functions to draw to a view.

> **Note:** The `UIPrintPageRenderer` class declares several methods for drawing printable content. In a manner similar to `drawRect:`, UIKit installs an implicit graphics context for implementations of these methods. This graphics context establishes a ULO default coordinate system.

Drawing to Bitmap Contexts and PDF Contexts

UIKit provides functions for rendering images in a bitmap graphics context and for generating PDF content by drawing in a PDF graphics context. Both of these approaches require that you first call a function that creates a graphics context—a bitmap context or a PDF context, respectively. The returned object serves as the current (and implicit) graphics context for subsequent drawing and state-setting calls. When you finish drawing in the context, you call another function to close the context.

Both the bitmap context and the PDF context provided by UIKit establish a ULO default coordinate system. Core Graphics has corresponding functions for rendering in a bitmap graphics context and for drawing in a PDF graphics context. The context that an app directly creates through Core Graphics, however, establishes a LLO default coordinate system.

> **Note:** In iOS, it is recommended that you use the UIKit functions for drawing to bitmap contexts and PDF contexts. However, if you do use the Core Graphics alternatives and intend to display the rendered results, you will have to adjust your code to compensate for the difference in default coordinate systems. See "Flipping the Default Coordinate System" (page 25) for more information.

For details, see "Drawing and Creating Images" (page 40) (for drawing to bitmap contexts) and "Generating PDF Content" (page 45) (for drawing to PDF contexts).

Color and Color Spaces

iOS supports the full range of color spaces available in Quartz; however, most apps should need only the RGB color space. Because iOS is designed to run on embedded hardware and display graphics onscreen, the RGB color space is the most appropriate one to use.

The `UIColor` object provides convenience methods for specifying color values using RGB, HSB, and grayscale values. When creating colors in this way, you never need to specify the color space. It is determined for you automatically by the `UIColor` object.

You can also use the `CGContextSetRGBStrokeColor` and `CGContextSetRGBFillColor` functions in the Core Graphics framework to create and set colors. Although the Core Graphics framework includes support for creating colors using other color spaces, and for creating custom color spaces, using those colors in your drawing code is not recommended. Your drawing code should always use RGB colors.

Drawing with Quartz and UIKit

Quartz is the general name for the native drawing technology in iOS. The Core Graphics framework is at the heart of Quartz, and is the primary interface you use for drawing content. This framework provides data types and functions for manipulating the following:

- Graphics contexts
- Paths
- Images and bitmaps
- Transparency layers
- Colors, pattern colors, and color spaces
- Gradients and shadings
- Fonts
- PDF content

UIKit builds on the basic features of Quartz by providing a focused set of classes for graphics-related operations. The UIKit graphics classes are not intended as a comprehensive set of drawing tools—Core Graphics already provides that. Instead, they provide drawing support for other UIKit classes. UIKit support includes the following classes and functions:

- `UIImage`, which implements an immutable class for displaying images
- `UIColor`, which provides basic support for device colors
- `UIFont`, which provides font information for classes that need it
- `UIScreen`, which provides basic information about the screen
- `UIBezierPath`, which enables your app to draw lines, arcs, ovals, and other shapes.
- Functions for generating a JPEG or PNG representation of a `UIImage` object
- Functions for drawing to a bitmap graphics context
- Functions for generating PDF data by drawing to a PDF graphics context
- Functions for drawing rectangles and clipping the drawing area
- Functions for changing and getting the current graphics context

For information about the classes and methods that comprise UIKit, see *UIKit Framework Reference*. For more information about the opaque types and functions that comprise the Core Graphics framework, see *Core Graphics Framework Reference*.

Configuring the Graphics Context

Before calling your `drawRect:` method, the view object creates a graphics context and sets it as the current context. This context exists only for the lifetime of the `drawRect:` call. You can retrieve a pointer to this graphics context by calling the `UIGraphicsGetCurrentContext` function. This function returns a reference to a `CGContextRef` type, which you pass to Core Graphics functions to modify the current graphics state. Table 1-1 lists the main functions you use to set different aspects of the graphics state. For a complete list of functions, see *CGContext Reference*. This table also lists UIKit alternatives where they exist.

Table 1-1 Core graphics functions for modifying graphics state

Graphics state	Core Graphics functions	UIKit alternatives

Current transformation matrix (CTM)	`CGContextRotateCTM`
	`CGContextScaleCTM`
	`CGContextTranslateCTM`
	`CGContextConcatCTM`
		None

Graphics state	Core Graphics functions	UIKit alternatives
Clipping area	`CGContextClipToRect`	`UIRectClip` function

Line: Width, join, cap, dash, miter limit	`CGContextSetLineWidth`
	`CGContextSetLineJoin`
	`CGContextSetLineCap`
	`CGContextSetLineDash`
	`CGContextSetMiterLimit`
		None

Accuracy of curve estimation	`CGContextSetFlatness`	None
Anti-aliasing setting	`CGContextSetAllowsAntialiasing`	None
Color: Fill and stroke settings	`CGContextSetRGBFillColor` `CGContextSetRGBStrokeColor`	`UIColor` class
Alpha global value (transparency)	`CGContextSetAlpha`	None
Rendering intent	`CGContextSetRenderingIntent`	None
Color space: Fill and stroke settings	`CGContextSetFillColorSpace` `CGContextSetStrokeColorSpace`	`UIColor` class

Text: Font, font size, character spacing, text drawing mode	`CGContextSetFont`
	`CGContextSetFontSize`
	`CGContextSetCharacterSpacing`
		`UIFont` class

Blend mode	`CGContextSetBlendMode`	The `UIImage` class and various drawing functions let you specify which blend mode to use.

The graphics context contains a stack of saved graphics states. When Quartz creates a graphics context, the stack is empty. Using the `CGContextSaveGState` function pushes a copy of the current graphics state onto the stack. Thereafter, modifications you make to the graphics state affect subsequent drawing operations but do not affect the copy stored on the stack. When you are done making modifications, you can return to the previous graphics state by popping the saved state off the top of the stack using the

`CGContextRestoreGState` function. Pushing and popping graphics states in this manner is a fast way to return to a previous state and eliminates the need to undo each state change individually. It is also the only way to restore some aspects of the state, such as the clipping path, back to their original settings.

For general information about graphics contexts and using them to configure the drawing environment, see "Graphics Contexts" in *Quartz 2D Programming Guide*.

Creating and Drawing Paths

A path is a vector-based shapes created from a sequence of lines and Bézier curves. UIKit includes the `UIRectFrame` and `UIRectFill` functions (among others) for drawing simple paths such as rectangles in your views. Core Graphics also includes convenience functions for creating simple paths such as rectangles and ellipses.

For more complex paths, you must create the path yourself using the `UIBezierPath` class of UIKit, or using the functions that operate on the `CGPathRef` opaque type in the Core Graphics framework. Although you can construct a path without a graphics context using either API, the points in the path still must refer to the current coordinate system (which either has a ULO or LLO orientation), and you still need a graphics context to actually render the path.

When drawing a path, you must have a current context set. This context can be a custom view's context (in `drawRect:`), a bitmap context, or a PDF context. The coordinate system determines how the path is rendered. `UIBezierPath` assumes a ULO coordinate system. Thus, if your view is flipped (to use LLO coordinates), the resulting shape may render differently than intended. For best results, you should always specify points relative to the origin of the current coordinate system of the graphics context used for rendering.

> **Note:** Arcs are an aspect of paths that require additional work even if this "rule" is followed. If you create a path using Core Graphic functions that locate points in a ULO coordinate system, and then render the path in a UIKit view, the direction an arc "points" is different. See "Side Effects of Drawing with Different Coordinate Systems" (page 25) for more on this subject.

For creating paths in iOS, it is recommended that you use `UIBezierPath` instead of `CGPath` functions unless you need some of the capabilities that only Core Graphics provides, such as adding ellipses to paths. For more on creating and rendering paths in UIKit, see "Drawing Shapes Using Bézier Paths" (page 30).

For information on using `UIBezierPath` to draw paths, see "Drawing Shapes Using Bézier Paths" (page 30). For information on how to draw paths using Core Graphics, including information about how you specify the points for complex path elements, see "Paths" in *Quartz 2D Programming Guide*. For information on the functions you use to create paths, see *CGContext Reference* and *CGPath Reference*.

Creating Patterns, Gradients, and Shadings

The Core Graphics framework includes additional functions for creating patterns, gradients, and shadings. You use these types to create non monochrome colors and use them to fill the paths you create. Patterns are created from repeating images or content. Gradients and shadings provide different ways to create smooth transitions from color to color.

The details for creating and using patterns, gradients, and shadings are all covered in *Quartz 2D Programming Guide*.

Customizing the Coordinate Space

By default, UIKit creates a straightforward current transformation matrix that maps points onto pixels. Although you can do all of your drawing without modifying that matrix, sometimes it can be convenient to do so.

When your view's `drawRect:` method is first called, the CTM is configured so that the origin of the coordinate system matches the your view's origin, its positive X axis extends to the right, and its positive Y axis extends down. However, you can change the CTM by adding scaling, rotation, and translation factors to it and thereby change the size, orientation, and position of the default coordinate system relative to the underlying view or window.

Using Coordinate Transforms to Improve Drawing Performance

Modifying the CTM is a standard technique for drawing content in a view because it allows you to reuse paths, which potentially reduces the amount of computation required while drawing. For example, if you want to draw a square starting at the point (20, 20), you could create a path that moves to (20, 20) and then draws the needed set of lines to complete the square. However, if you later decide to move that square to the point (10, 10), you would have to recreate the path with the new starting point. Because creating paths is a relatively expensive operation, it is preferable to create a square whose origin is at (0, 0) and to modify the CTM so that the square is drawn at the desired origin.

In the Core Graphics framework, there are two ways to modify the CTM. You can modify the CTM directly using the CTM manipulation functions defined in *CGContext Reference*. You can also create a `CGAffineTransform` structure, apply any transformations you want, and then concatenate that transform onto the CTM. Using an affine transform lets you group transformations and then apply them to the CTM all at once. You can also evaluate and invert affine transforms and use them to modify point, size, and rectangle values in your code. For more information on using affine transforms, see *Quartz 2D Programming Guide* and *CGAffineTransform Reference*.

Flipping the Default Coordinate System

Flipping in UIKit drawing modifies the backing `CALayer` to align a drawing environment having a LLO coordinate system with the default coordinate system of UIKit. If you only use UIKit methods and function for drawing, you shouldn't need to flip the CTM. However, if you mix Core Graphics or Image I/O function calls with UIKit calls, flipping the CTM might be necessary.

Specifically, if you draw an image or PDF document by calling Core Graphics functions directly, the object is rendered upside-down in the view's context. You must flip the CTM to display the image and pages correctly.

To flip a object drawn to a Core Graphics context so that it appears correctly when displayed in a UIKit view, you must modify the CTM in two steps. You translate the origin to the upper-left corner of the drawing area, and then you apply a scale translation, modifying the y-coordinate by -1. The code for doing this looks similar to the following:

```
CGContextSaveGState(graphicsContext);
CGContextTranslateCTM(graphicsContext, 0.0, imageHeight);
CGContextScaleCTM(graphicsContext, 1.0, -1.0);
CGContextDrawImage(graphicsContext, image, CGRectMake(0, 0, imageWidth,
imageHeight));
CGContextRestoreGState(graphicsContext);
```

If you create a `UIImage` object initialized with a Core Graphics image object, UIKit performs the flip transform for you. Every `UIImage` object is backed by a `CGImageRef` opaque type. You can access the Core Graphics object through the `CGImage` property and do some work with the image. (Core Graphics has image-related facilities not available in UIKit.) When you are finished, you can recreate the `UIImage` object from the modified `CGImageRef` object.

> **Note:** You can use the Core Graphics function `CGContextDrawImage` to draw an image to any rendering destination. This function has two parameters, the first for a graphics context and the second for a rectangle that defines both the size of the image and its location in a drawing surface such as a view. When drawing an image with `CGContextDrawImage`, if you don't adjust the current coordinate system to a LLO orientation, the image appears inverted in a UIKit view. Additionally, the origin of the rectangle passed into this function is relative to the origin of the coordinate system that is current when the function is called.

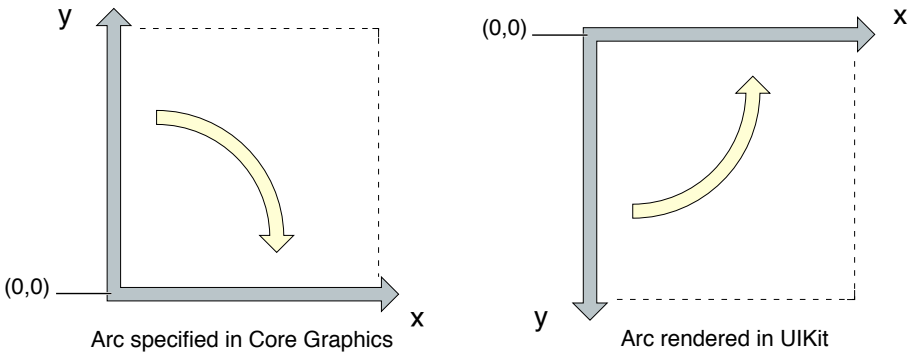
Side Effects of Drawing with Different Coordinate Systems

Some rendering oddities are brought to light when you draw an object with with reference to the default coordinate system of one drawing technology and then render it in a graphics context of the other. You may want to adjust your code to account for these side effects.

Arcs and Rotations

If you draw a path with functions such as `CGContextAddArc` and `CGPathAddArc` and assume a LLO coordinate system, then you need to flip the CTM to render the arc correctly in a UIKit view. However, if you use the same function to create an arc with points located in a ULO coordinate system and then render the path in a UIKit view, you'll notice that the arc is an altered version of its original. The terminating endpoint of the arc now points in the opposite direction of what that endpoint would do were the arc created using the `UIBezierPath` class. For example, a downward-pointing arrow now points upward (as shown in Figure 1-5), and the direction in which the arc "bends" is also different. You must change the direction of Core Graphics-drawn arcs to account for the ULO-based coordinate system; this direction is controlled by the `startAngle` and `endAngle` parameters of those functions.

Figure 1-5 Arc rendering in Core Graphics versus UIKit



You can observe the same kind of mirroring effect if you rotate an object (for example, by calling `CGContextRotateCTM`). If you rotate an object using Core Graphics calls that make reference to a ULO coordinate system, the direction of the object when rendered in UIKit is reversed. You must account for the different directions of rotation in your code; with `CGContextRotateCTM`, you do this by inverting the sign of the `angle` parameter (so, for example, a negative value becomes a positive value).

Shadows

The direction a shadow falls from its object is specified by an offset value, and the meaning of that offset is a convention of a drawing framework. In UIKit, positive x and y offsets make a shadow go down and to the right of an object. In Core Graphics, positive x and y offsets make a shadow go up and to the right of an object. Flipping the CTM to align an object with the default coordinate system of UIKit does not affect the object's shadow, and so a shadow does not correctly track its object. To get it to track correctly, you must modify the offset values appropriately for the current coordinate system.

Note: Prior to iOS 3.2, Core Graphics and UIKit shared the same convention for shadow direction: positive offset values make the shadow go down and to the right of an object.

Applying Core Animation Effects

Core Animation is an Objective-C framework that provides infrastructure for creating fluid, real-time animations quickly and easily. Core Animation is not a drawing technology itself, in the sense that it does not provide primitive routines for creating shapes, images, or other types of content. Instead, it is a technology for manipulating and displaying content that you created using other technologies.

Most apps can benefit from using Core Animation in some form in iOS. Animations provide feedback to the user about what is happening. For example, when the user navigates through the Settings app, screens slide in and out of view based on whether the user is navigating further down the preferences hierarchy or back up to the root node. This kind of feedback is important and provides contextual information for the user. It also enhances the visual style of an app.

In most cases, you may be able to reap the benefits of Core Animation with very little effort. For example, several properties of the `UIView` class (including the view's frame, center, color, and opacity—among others) can be configured to trigger animations when their values change. You have to do some work to let UIKit know that you want these animations performed, but the animations themselves are created and run automatically for you. For information about how to trigger the built-in view animations, see "Animating Views" in *UIView Class Reference*.

When you go beyond the basic animations, you must interact more directly with Core Animation classes and methods. The following sections provide information about Core Animation and show you how to work with its classes and methods to create typical animations in iOS. For additional information about Core Animation and how to use it, see *Core Animation Programming Guide*.

About Layers

The key technology in Core Animation is the layer object. Layers are lightweight objects that are similar in nature to views, but that are actually model objects that encapsulate geometry, timing, and visual properties for the content you want to display. The content itself is provided in one of three ways:

- You can assign a `CGImageRef` to the `contents` property of the layer object.
- You can assign a delegate to the layer and let the delegate handle the drawing.
- You can subclass `CALayer` and override one of the display methods.

When you manipulate a layer object's properties, what you are actually manipulating is the model-level data that determines how the associated content should be displayed. The actual rendering of that content is handled separately from your code and is heavily optimized to ensure it is fast. All you must do is set the layer content, configure the animation properties, and then let Core Animation take over.

For more information about layers and how they are used, see *Core Animation Programming Guide*.

About Animations

When it comes to animating layers, Core Animation uses separate animation objects to control the timing and behavior of the animation. The `CAAnimation` class and its subclasses provide different types of animation behaviors that you can use in your code. You can create simple animations that migrate a property from one value to another, or you can create complex keyframe animations that track the animation through the set of values and timing functions you provide.

Core Animation also lets you group multiple animations together into a single unit, called a transaction. The `CATransaction` object manages the group of animations as a unit. You can also use the methods of this class to set the duration of the animation.

For examples of how to create custom animations, see *Animation Types and Timing Programming Guide*.

Accounting for Scale Factors in Core Animation Layers

Apps that use Core Animation layers directly to provide content may need to adjust their drawing code to account for scale factors. Normally, when you draw in your view's `drawRect:` method, or in the `drawLayer:inContext:` method of the layer's delegate, the system automatically adjusts the graphics context to account for scale factors. However, knowing or changing that scale factor might still be necessary when your view does one of the following:

- Creates additional Core Animation layers with different scale factors and composites them into its own content
- Sets the `contents` property of a Core Animation layer directly

Core Animation's compositing engine looks at the `contentsScale` property of each layer to determine whether the contents of that layer need to be scaled during compositing. If your app creates layers without an associated view, each new layer object's scale factor is initially set to 1.0. If you do not change that scale factor, and if you subsequently draw the layer on a high-resolution screen, the layer's contents are scaled automatically to compensate for the difference in scale factors. If you do not want the contents to be scaled, you can change the layer's scale factor to 2.0 by setting a new value for the `contentsScale` property, but if you do so without providing high-resolution content, your existing content may appear smaller than you were expecting. To fix that problem, you need to provide higher-resolution content for your layer.

Important: The `contentsGravity` property of the layer plays a role in determining whether standard-resolution layer content is scaled on a high-resolution screen. This property is set to the value `kCAGravityResize` by default, which causes the layer content to be scaled to fit the layer's bounds. Changing the gravity to a nonresizing option eliminates the automatic scaling that would otherwise occur. In such a situation, you may need to adjust your content or the scale factor accordingly.

Adjusting the content of your layer to accommodate different scale factors is most appropriate when you set the `contents` property of a layer directly. Quartz images have no notion of scale factors and therefore work directly with pixels. Therefore, before creating the `CGImageRef` object you plan to use for the layer's contents, check the scale factor and adjust the size of your image accordingly. Specifically, load an appropriately sized image from your app bundle or use the `UIGraphicsBeginImageContextWithOptions` function to create an image whose scale factor matches the scale factor of your layer. If you do not create a high-resolution bitmap, the existing bitmap may be scaled as discussed previously.

For information on how to specify and load high-resolution images, see "Loading Images into Your App" (page 83). For information about how to create high-resolution images, see "Drawing to Bitmap Contexts and PDF Contexts" (page 19).

Drawing Shapes Using Bézier Paths

In iOS 3.2 and later, you can use the `UIBezierPath` class to create vector-based paths. The `UIBezierPath` class is an Objective-C wrapper for the path-related features in the Core Graphics framework. You can use this class to define simple shapes, such as ovals and rectangles, as well as complex shapes that incorporate multiple straight and curved line segments.

You can use path objects to draw shapes in your app's user interface. You can draw the path's outline, fill the space it encloses, or both. You can also use paths to define a clipping region for the current graphics context, which you can then use to modify subsequent drawing operations in that context.

Bézier Path Basics

A `UIBezierPath` object is a wrapper for a `CGPathRef` data type. **Paths** are vector-based shapes that are built using line and curve segments. You can use line segments to create rectangles and polygons, and you can use curve segments to create arcs, circles, and complex curved shapes. Each segment consists of one or more points (in the current coordinate system) and a drawing command that defines how those points are interpreted.

Each set of connected line and curve segments form what is referred to as a **subpath**. The end of one line or curve segment in a subpath defines the beginning of the next. A single `UIBezierPath` object may contain one or more subpaths that define the overall path, separated by `moveToPoint:` commands that effectively raise the drawing pen and move it to a new location.

The processes for building and using a path object are separate. Building the path is the first process and involves the following steps:

1. Create the path object.
2. Set any relevant drawing attributes of your `UIBezierPath` object, such as the `lineWidth` or `lineJoinStyle` properties for stroked paths or the `usesEvenOddFillRule` property for filled paths. These drawing attributes apply to the entire path.
3. Set the starting point of the initial segment using the `moveToPoint:` method.
4. Add line and curve segments to define a subpath.
5. Optionally, close the subpath by calling `closePath`, which draws a straight line segment from the end of the last segment to the beginning of the first.
6. Optionally, repeat the steps 3, 4, and 5 to define additional subpaths.

When building your path, you should arrange the points of your path relative to the origin point (0, 0). Doing so makes it easier to move the path around later. During drawing, the points of your path are applied as-is to the coordinate system of the current graphics context. If your path is oriented relative to the origin, all you have to do to reposition it is apply an affine transform with a translation factor to the current graphics context. The advantage of modifying the graphics context (as opposed to the path object itself) is that you can easily undo the transformation by saving and restoring the graphics state.

To draw your path object, you use the `stroke` and `fill` methods. These methods render the line and curve segments of your path in the current graphics context. The rendering process involves rasterizing the line and curve segments using the attributes of the path object. The rasterization process does not modify the path object itself. As a result, you can render the same path object multiple times in the current context or in another context.

Adding Lines and Polygons to Your Path

Lines and polygons are simple shapes that you build point-by-point using the `moveToPoint:` and `addLineToPoint:` methods. The `moveToPoint:` method sets the starting point of the shape you want to create. From that point, you create the lines of the shape using the `addLineToPoint:` method. You create the lines in succession, with each line being formed between the previous point and the new point you specify.

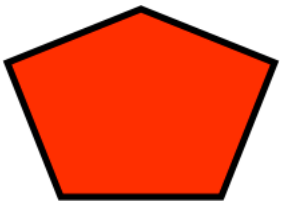
Listing 2-1 shows the code needed to create a pentagon shape using individual line segments. (Figure 2-1 shows the result of drawing this shape with appropriate stroke and fill color settings, as described in "Rendering the Contents of a Bézier Path Object" (page 36).) This code sets the initial point of the shape and then adds four connected line segments. The fifth segment is added by the call to the `closePath` method, which connects the last point (0, 40) with the first point (100, 0).

Listing 2-1 Creating a pentagon shape

```
UIBezierPath *aPath = [UIBezierPath bezierPath];

// Set the starting point of the shape.
[aPath moveToPoint:CGPointMake(100.0, 0.0)];

// Draw the lines.
[aPath addLineToPoint:CGPointMake(200.0, 40.0)];
[aPath addLineToPoint:CGPointMake(160, 140)];
[aPath addLineToPoint:CGPointMake(40.0, 140)];
[aPath addLineToPoint:CGPointMake(0.0, 40.0)];
```

```
[aPath closePath];
```

Figure 2-1 Shape drawn with methods of the `UIBezierPath` class



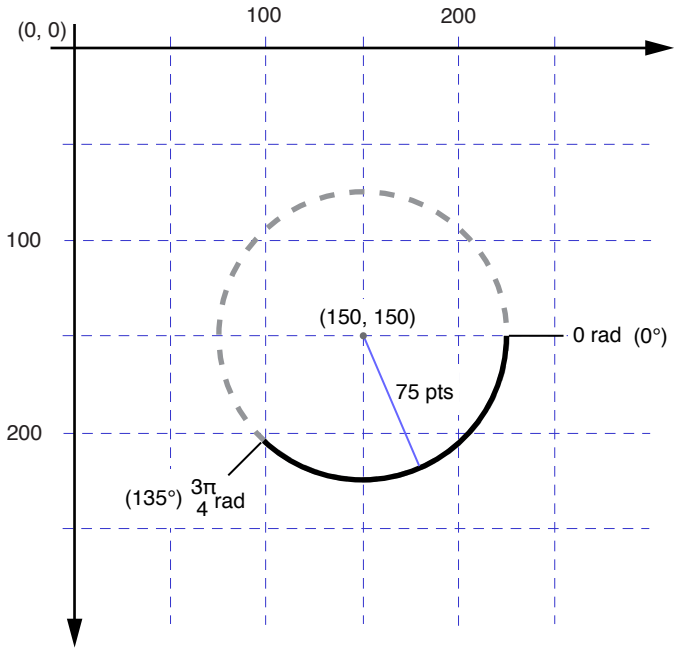
Using the `closePath` method not only ends the subpath describing the shape, it also draws a line segment between the first and last points. This is a convenient way to finish a polygon without having to draw the final line.

Adding Arcs to Your Path

The `UIBezierPath` class provides support for initializing a new path object with an arc segment. The parameters of the `bezierPathWithArcCenter:radius:startAngle:endAngle:clockwise:` method define the circle that contains the desired arc and the start and end points of the arc itself. Figure 2-2 shows the components that go into creating an arc, including the circle that defines the arc and the angle measurements used to

specify it. In this case, the arc is created in the clockwise direction. (Drawing the arc in the counterclockwise direction would paint the dashed portion of the circle instead.) The code for creating this arc is shown in Listing 2-2 (page 33).

Figure 2-2 An arc in the default coordinate system



Listing 2-2 Creating a new arc path

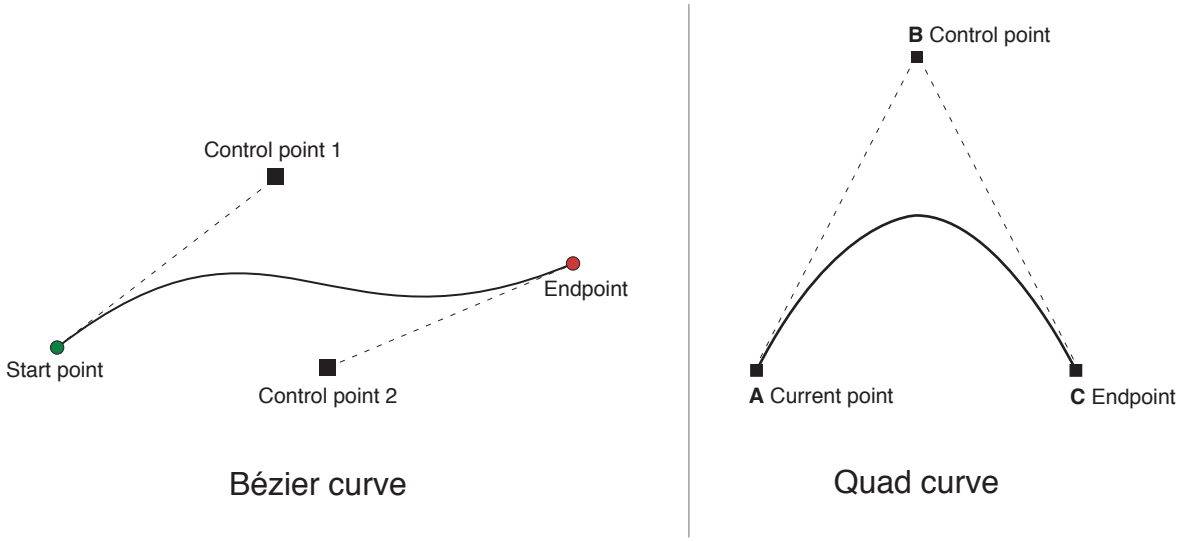
```
// pi is approximately equal to 3.14159265359.
#define   DEGREES_TO_RADIANS(degrees)  ((pi * degrees)/ 180)


- (UIBezierPath *)createArcPath
{
   UIBezierPath *aPath = [UIBezierPath bezierPathWithArcCenter:CGPointMake(150,
150)
                          radius:75
                          startAngle:0
                          endAngle:DEGREES_TO_RADIANS(135)
                          clockwise:YES];
   return aPath;
}
```

If you want to incorporate an arc segment into the middle of a path, you must modify the path object's `CGPathRef` data type directly. For more information about modifying the path using Core Graphics functions, see "Modifying the Path Using Core Graphics Functions" (page 35).

Adding Curves to Your Path

The `UIBezierPath` class provides support for adding cubic and quadratic Bézier curves to a path. Curve segments start at the current point and end at the point you specify. The shape of the curve is defined using tangent lines between the start and end points and one or more control points. Figure 2-3 shows approximations of both types of curve and the relationship between the control points and the shape of the curve. The exact curvature of each segment involves a complex mathematical relationship between all of the points and is well documented online and at Wikipedia.

Figure 2-3 Curve segments in a path



To add curves to a path, you use the following methods:

- **Cubic curve:**`addCurveToPoint:controlPoint1:controlPoint2:`
- **Quadratic curve:**`addQuadCurveToPoint:controlPoint:`

Because curves rely on the current point of the path, you must set the current point before calling either of the preceding methods. Upon completion of the curve, the current point is updated to the new end point you specified.

Creating Oval and Rectangular Paths

Ovals and rectangles are common types of paths that are built using a combination of curve and line segments. The `UIBezierPath` class includes the `bezierPathWithRect:` and `bezierPathWithOvalInRect:` convenience methods for creating paths with oval or rectangular shapes. Both of these methods create a new path object and initialize it with the specified shape. You can use the returned path object right away or add more shapes to it as needed.

If you want to add a rectangle to an existing path object, you must do so using the `moveToPoint:`, `addLineToPoint:`, and `closePath` methods as you would for any other polygon. Using the `closePath` method for the final side of the rectangle is a convenient way to add the final line of the path and also mark the end of the rectangle subpath.

If you want to add an oval to an existing path, the simplest way to do so is to use Core Graphics. Although you can use the `addQuadCurveToPoint:controlPoint:` to approximate an oval surface, the `CGPathAddEllipseInRect` function is much simpler to use and more accurate. For more information, see "Modifying the Path Using Core Graphics Functions" (page 35).

Modifying the Path Using Core Graphics Functions

The `UIBezierPath` class is really just a wrapper for a `CGPathRef` data type and the drawing attributes associated with that path. Although you normally add line and curve segments using the methods of the `UIBezierPath` class, the class also exposes a `CGPath` property that you can use to modify the underlying path data type directly. You can use this property when you would prefer to build your path using the functions of the Core Graphics framework.

There are two ways to modify the path associated with a `UIBezierPath` object. You can modify the path entirely using Core Graphics functions, or you can use a mixture of Core Graphics functions and `UIBezierPath` methods. Modifying the path entirely using Core Graphics calls is easier in some ways. You create a mutable `CGPathRef` data type and call whatever functions you need to modify its path information. When you are done you assign your path object to the corresponding `UIBezierPath` object, as shown in Listing 2-3.

Listing 2-3 Assigning a new `CGPathRef` to a `UIBezierPath` object

```
// Create the path data.
CGMutablePathRef cgPath = CGPathCreateMutable();
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(0, 0, 300, 300));
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(50, 50, 200, 200));

// Now create the UIBezierPath object.
```

```
UIBezierPath *aPath = [UIBezierPath bezierPath];
aPath.CGPath = cgPath;
aPath.usesEvenOddFillRule = YES;

// After assigning it to the UIBezierPath object, you can release
// your CGPathRef data type safely.
CGPathRelease(cgPath);
```

If you choose to use a mixture of Core Graphics functions and `UIBezierPath` methods, you must carefully move the path information back and forth between the two. Because a `UIBezierPath` object owns its underlying `CGPathRef` data type, you cannot simply retrieve that type and modify it directly. Instead, you must make a mutable copy, modify the copy, and then assign the copy back to the `CGPath` property as shown in Listing 2-4.

Listing 2-4 Mixing Core Graphics and `UIBezierPath` calls

```
UIBezierPath *aPath = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(0, 0, 300,
 300)];

// Get the CGPathRef and create a mutable version.
CGPathRef cgPath = aPath.CGPath;
CGMutablePathRef  mutablePath = CGPathCreateMutableCopy(cgPath);

// Modify the path and assign it back to the UIBezierPath object.
CGPathAddEllipseInRect(mutablePath, NULL, CGRectMake(50, 50, 200, 200));
aPath.CGPath = mutablePath;

// Release both the mutable copy of the path.
CGPathRelease(mutablePath);
```

Rendering the Contents of a Bézier Path Object

After creating a `UIBezierPath` object, you can render it in the current graphics context using its `stroke` and `fill` methods. Before you call these methods, though, there are usually a few other tasks to perform to ensure your path is drawn correctly:

- Set the desired stroke and fill colors using the methods of the `UIColor` class.
- Position the shape where you want it in the target view.

  If you created your path relative to the point (0, 0), you can apply an appropriate affine transform to the current drawing context. For example, to draw your shape starting at the point (10, 10), you would call the `CGContextTranslateCTM` function and specify `10` for both the horizontal and vertical translation values. Adjusting the graphics context (as opposed to the points in the path object) is preferred because you can undo the change more easily by saving and restoring the previous graphics state.
- Update the drawing attributes of the path object. The drawing attributes of your `UIBezierPath` instance override the values associated with the graphics context when rendering the path.

Listing 2-5 shows a sample implementation of a `drawRect:` method that draws an oval in a custom view. The upper-left corner of the oval's bounding rectangle is located at the point (50, 50) in the view's coordinate system. Because fill operations paint right up to the path boundary, this method fills the path before stroking it. This prevents the fill color from obscuring half of the stroked line.

Listing 2-5 Drawing a path in a view

```
- (void)drawRect:(CGRect)rect
{
    // Create an oval shape to draw.
    UIBezierPath *aPath = [UIBezierPath bezierPathWithOvalInRect:
                                CGRectMake(0, 0, 200, 100)];

    // Set the render colors.
    [[UIColor blackColor] setStroke];
    [[UIColor redColor] setFill];

    CGContextRef aRef = UIGraphicsGetCurrentContext();

    // If you have content to draw after the shape,
    // save the current state before changing the transform.
    //CGContextSaveGState(aRef);

    // Adjust the view's origin temporarily. The oval is
    // now drawn relative to the new origin point.
    CGContextTranslateCTM(aRef, 50, 50);
```

```
    // Adjust the drawing options as needed.
    aPath.lineWidth = 5;

    // Fill the path before stroking it so that the fill
    // color does not obscure the stroked line.
    [aPath fill];
    [aPath stroke];

    // Restore the graphics state before drawing any other content.
    //CGContextRestoreGState(aRef);
}
```

Doing Hit-Detection on a Path

To determine whether a touch event occurred on the filled portion of a path, you can use the `containsPoint:` method of `UIBezierPath`. This method tests the specified point against all closed subpaths in the path object and returns `YES` if it lies on or inside any of those subpaths.

> **Important:** The `containsPoint:` method and the Core Graphics hit-testing functions operate only on closed paths. These methods always return `NO` for hits on open subpaths. If you want to do hit detection on an open subpath, you must create a copy of your path object and close the open subpaths before testing points.

If you want to do hit-testing on the stroked portion of the path (instead of the fill area), you must use Core Graphics. The `CGContextPathContainsPoint` function lets you test points on either the fill or stroke portion of the path currently assigned to the graphics context. Listing 2-6 shows a method that tests to see whether the specified point intersects the specified path. The `inFill` parameter lets the caller specify whether the point should be tested against the filled or stroked portion of the path. The path passed in by the caller must contain one or more closed subpaths for the hit detection to succeed.

Listing 2-6 Testing points against a path object

```
- (BOOL)containsPoint:(CGPoint)point onPath:(UIBezierPath *)path
inFillArea:(BOOL)inFill
{
   CGContextRef context = UIGraphicsGetCurrentContext();
```

```
    CGPathRef cgPath = path.CGPath;
    BOOL    isHit = NO;

    // Determine the drawing mode to use. Default to
    // detecting hits on the stroked portion of the path.
    CGPathDrawingMode mode = kCGPathStroke;
    if (inFill)
    {
        // Look for hits in the fill area of the path instead.
        if (path.usesEvenOddFillRule)
            mode = kCGPathEOFill;
        else
            mode = kCGPathFill;
    }

    // Save the graphics state so that the path can be
    // removed later.
    CGContextSaveGState(context);
    CGContextAddPath(context, cgPath);

    // Do the hit detection.
    isHit = CGContextPathContainsPoint(context, point, mode);

    CGContextRestoreGState(context);

    return isHit;
}
```

Drawing and Creating Images

Most of the time, it is fairly straightforward to display images using standard views. However, there are two situations in which you may need to do additional work:

- If you want to display images as part of a custom view, you must draw the images yourself in your view's `drawRect:` method. "Drawing Images" (page 40) explains how.
- If you want to render images offscreen (to draw later, or to save into a file), you must create a bitmap image context. To learn more, read "Creating New Images Using Bitmap Graphics Contexts" (page 41).

Drawing Images

For maximum performance, if your image drawing needs can be met using the `UIImageView` class, you should use this image object to initialize a `UIImageView` object. However, if you need to draw an image explicitly, you can store the image and use it later in your view's `drawRect:` method.

The following example shows how to load an image from your app's bundle.

```
NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"myImage"
ofType:@"png"];
UIImage *myImageObj = [[UIImage alloc] initWithContentsOfFile:imagePath];


// Store the image into a property of type UIImage *
// for use later in the class's drawRect: method.
self.anImage = myImageObj;
```

To draw the resulting image explicitly in your view's `drawRect:` method, you can use any of the drawing methods available in `UIImage`. These methods let you specify where in your view you want to draw the image and therefore do not require you to create and apply a separate transform prior to drawing.

The following snippet draws the image loaded above at the point (10, 10) in the view.

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

```
    ...

    // Draw the image.
    [self.anImage drawAtPoint:CGPointMake(10, 10)];
}
```

Important: If you use the `CGContextDrawImage` function to draw bitmap images directly, the image data is inverted along the y axis by default. This is because Quartz images assume a coordinate system with a lower-left corner origin and positive coordinate axes extending up and to the right from that point. Although you can apply a transform before drawing, the simpler (and recommended) way to draw Quartz images is to wrap them in a `UIImage` object, which compensates for this difference in coordinate spaces automatically. For more information on creating and drawing images using Core Graphics, see *Quartz 2D Programming Guide*.

Creating New Images Using Bitmap Graphics Contexts

Most of the time, when drawing, your goal is to show something onscreen. However, it is sometimes useful to draw something to an offscreen buffer. For example, you might want to create a thumbnail of an existing image, draw into a buffer so that you can save it to a file, and so on. To support those needs, you can create a bitmap image context, use UIKit framework or Core Graphics functions to draw to it, and then obtain an image object from the context.

In UIKit, the procedure is as follows:

1. Call `UIGraphicsBeginImageContextWithOptions` to create a bitmap context and push it onto the graphics stack.

   For the first parameter (`size`), pass a `CGSize` value to specify the dimensions of the bitmap context (in points).

   For the second parameter (`opaque`), if your image contains transparency (an alpha channel), pass `NO`. Otherwise, pass `YES` to maximize performance.

   For the final parameter (`scale`), pass `0.0` for a bitmap that is scaled appropriately for the main screen of the device, or pass the scale factor of your choice.

   For example, the following code snippet creates a bitmap that is 200 x 200 pixels. (The number of pixels is determined by multiplying the size of the image by the scale factor.)

   ```
   UIGraphicsBeginImageContextWithOptions(CGSizeMake(100.0,100.0), NO, 2.0);
   ```

> **Note:** You should generally avoid calling the similarly named `UIGraphicsBeginImageContext` function (except as a fallback for backwards compatibility), because it always creates images with a scale factor of 1.0. If the underlying device has a high-resolution screen, an image created with `UIGraphicsBeginImageContext` might not appear as smooth when rendered.

2. Use UIKit or Core Graphics routines to draw the content of the image into the newly created graphics context.
3. Call the `UIGraphicsGetImageFromCurrentImageContext` function to generate and return a `UIImage` object based on what you drew. If desired, you can continue drawing and call this method again to generate additional images.
4. Call `UIGraphicsEndImageContext` to pop the context from the graphics stack.

The method in Listing 3-1 gets an image downloaded over the Internet and draws it into an image-based context, scaled down to the size of an app icon. It then obtains a `UIImage` object created from the bitmap data and assigns it to an instance variable. Note that the size of the bitmap (the first parameter of `UIGraphicsBeginImageContextWithOptions`) and the size of the drawn content (the size of `imageRect`) should match. If the content is larger than the bitmap, a portion of the content will be clipped and not appear in the resulting image.

Listing 3-1 Drawing a scaled-down image to a bitmap context and obtaining the resulting image

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {

    UIImage *image = [[UIImage alloc] initWithData:self.activeDownload];

    if (image != nil && image.size.width != kAppIconHeight && image.size.height
!= kAppIconHeight) {

        CGRect imageRect = CGRectMake(0.0, 0.0, kAppIconHeight, kAppIconHeight);

        UIGraphicsBeginImageContextWithOptions(itemSize, NO, [UIScreen
mainScreen].scale);

        [image drawInRect:imageRect];

        self.appRecord.appIcon = UIGraphicsGetImageFromCurrentImageContext();  //
 UIImage returned.

        UIGraphicsEndImageContext();

    } else {

        self.appRecord.appIcon = image;

    }

    self.activeDownload = nil;

    [image release];
```

```
    self.imageConnection = nil;
    [delegate appImageDidLoad:self.indexPathInTableView];
}
```

You can also call Core Graphics functions to draw the contents of the generated bitmap image; the code fragment in Listing 3-2, which draws a scaled-down image of a PDF page, gives an example of this. Note that the code flips the graphics context prior to calling `CGContextDrawPDFPage` to align the drawn image with default coordinate system of UIKit.

Listing 3-2 Drawing to a bitmap context using Core Graphics functions

```
// Other code precedes...

CGRect pageRect = CGPDFPageGetBoxRect(page, kCGPDFMediaBox);
pdfScale = self.frame.size.width/pageRect.size.width;
pageRect.size = CGSizeMake(pageRect.size.width * pdfScale, pageRect.size.height *
 pdfScale);
UIGraphicsBeginImageContextWithOptions(pageRect.size, YES, pdfScale);
CGContextRef context = UIGraphicsGetCurrentContext();

// First fill the background with white.
CGContextSetRGBFillColor(context, 1.0,1.0,1.0,1.0);
CGContextFillRect(context,pageRect);
CGContextSaveGState(context);

// Flip the context so that the PDF page is rendered right side up
CGContextTranslateCTM(context, 0.0, pageRect.size.height);
CGContextScaleCTM(context, 1.0, -1.0);

// Scale the context so that the PDF page is rendered at the
// correct size for the zoom level.
CGContextScaleCTM(context, pdfScale,pdfScale);
CGContextDrawPDFPage(context, page);
CGContextRestoreGState(context);
UIImage *backgroundImage = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
```

```
backgroundImageView = [[UIImageView alloc] initWithImage:backgroundImage];

// Other code follows...
```

If you prefer using Core Graphics entirely for drawing in a bitmap graphics context, you can use the `CGBitmapContextCreate` function to create the context and draw your image contents into it. When you finish drawing, call the `CGBitmapContextCreateImage` function to obtain a `CGImageRef` object from the bitmap context. You can draw the Core Graphics image directly or use this it to initialize a `UIImage` object. When finished, call the `CGContextRelease` function on the graphics context.

Generating PDF Content

The UIKit framework provides a set of functions for generating PDF content using native drawing code. These functions let you create a graphics context that targets a PDF file or PDF data object. You can then create one or more PDF pages and draw into those pages using the same UIKit and Core Graphics drawing routines you use when drawing to the screen. When you are done, what you are left with is a PDF version of what you drew.

The overall drawing process is similar to the process for creating any other image (described in "Drawing and Creating Images" (page 40)). It consists of the following steps:

1. Create a PDF context and push it onto the graphics stack (as described in "Creating and Configuring the PDF Context" (page 45)).
2. Create a page (as described in "Drawing PDF Pages" (page 48)).
3. Use UIKit or Core Graphics routines to draw the content of the page.
4. Add links if needed (as described in "Creating Links Within Your PDF Content" (page 50)).
5. Repeat steps 2, 3, and 4 as needed.
6. End the PDF context (as described in "Creating and Configuring the PDF Context" (page 45)) to pop the context from the graphics stack and, depending on how the context was created, either write the resulting data to the specified PDF file or store it into the specified `NSMutableData` object.

The following sections describe the PDF creation process in more detail using a simple example. For information about the functions you use to create PDF content, see *UIKit Function Reference*.

Creating and Configuring the PDF Context

You create a PDF graphics context using either the `UIGraphicsBeginPDFContextToData` or `UIGraphicsBeginPDFContextToFile` function. These functions create the graphics context and associate it with a destination for the PDF data. For the `UIGraphicsBeginPDFContextToData` function, the destination is an `NSMutableData` object that you provide. And for the `UIGraphicsBeginPDFContextToFile` function, the destination is a file in your app's home directory.

PDF documents organize their content using a page-based structure. This structure imposes two restrictions on any drawing you do:

- There must be an open page before you issue any drawing commands.

- You must specify the size of each page.

The functions you use to create a PDF graphics context allow you to specify a default page size but they do not automatically open a page. After creating your context, you must explicitly open a new page using either the `UIGraphicsBeginPDFPage` or `UIGraphicsBeginPDFPageWithInfo` function. And each time you want to create a new page, you must call one of these functions again to mark the start of the new page. The `UIGraphicsBeginPDFPage` function creates a page using the default size, while the `UIGraphicsBeginPDFPageWithInfo` function lets you customize the page size and other page attributes.

When you are done drawing, you close the PDF graphics context by calling the `UIGraphicsEndPDFContext`. This function closes the last page and writes the PDF content to the file or data object you specified at creation time. This function also removes the PDF context from the graphics context stack.

Listing 4-1 shows the processing loop used by an app to create a PDF file from the text in a text view. Aside from three function calls to configure and manage the PDF context, most of the code is related to drawing the desired content. The `textView` member variable points to the `UITextView` object containing the desired text. The app uses the Core Text framework (and more specifically a `CTFramesetterRef` data type) to handle the text layout and management on successive pages. The implementations for the custom `renderPageWithTextRange:andFramesetter:` and `drawPageNumber:` methods are shown in Listing 4-2 (page 48).

Listing 4-1 Creating a new PDF file

```
- (IBAction)savePDFFile:(id)sender
{
    // Prepare the text using a Core Text Framesetter.
    CFAttributedStringRef currentText = CFAttributedStringCreate(NULL,
(CFStringRef)textView.text, NULL);
    if (currentText) {
        CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString(currentText);
        if (framesetter) {

            NSString *pdfFileName = [self getPDFFileName];
            // Create the PDF context using the default page size of 612 x 792.
            UIGraphicsBeginPDFContextToFile(pdfFileName, CGRectZero, nil);

            CFRange currentRange = CFRangeMake(0, 0);
            NSInteger currentPage = 0;
```

```
            BOOL done = NO;

            do {
                // Mark the beginning of a new page.
                UIGraphicsBeginPDFPageWithInfo(CGRectMake(0, 0, 612, 792), nil);

                // Draw a page number at the bottom of each page.
                currentPage++;
                [self drawPageNumber:currentPage];

                // Render the current page and update the current range to
                // point to the beginning of the next page.
                currentRange = [self renderPageWithTextRange:currentRange
andFramesetter:framesetter];

                // If we're at the end of the text, exit the loop.
                if (currentRange.location ==
CFAttributedStringGetLength((CFAttributedStringRef)currentText))
                    done = YES;
            } while (!done);

            // Close the PDF context and write the contents out.
            UIGraphicsEndPDFContext();

            // Release the framewetter.
            CFRelease(framesetter);

        } else {
          NSLog(@"Could not create the framesetter needed to lay out the atrributed
 string.");
        }
        // Release the attributed string.
        CFRelease(currentText);
    } else {
            NSLog(@"Could not create the attributed string for the framesetter");
    }
```

```
}
```

Drawing PDF Pages

All PDF drawing must be done in the context of a page. Every PDF document has at least one page and many may have multiple pages. You specify the start of a new page by calling the `UIGraphicsBeginPDFPage` or `UIGraphicsBeginPDFPageWithInfo` function. These functions close the previous page (if one was open), create a new page, and prepare it for drawing. The `UIGraphicsBeginPDFPage` creates the new page using the default size while the `UIGraphicsBeginPDFPageWithInfo` function lets you customize the page size or customize other aspects of the PDF page.

After you create a page, all of your subsequent drawing commands are captured by the PDF graphics context and translated into PDF commands. You can draw anything you want in the page, including text, vector shapes, and images just as you would in your app's custom views. The drawing commands you issue are captured by the PDF context and translated into PDF data. Placement of content on the the page is completely up to you but must take place within the bounding rectangle of the page.

Listing 4-2 shows two custom methods used to draw content inside a PDF page. The `renderPageWithTextRange:andFramesetter:` method uses Core Text to create a text frame that fits the page and then lay out some text inside that frame. After laying out the text, it returns an updated range that reflects the end of the current page and the beginning of the next page. The `drawPageNumber:` method uses the `NSString` drawing capabilities to draw a page number string at the bottom of each PDF page.

> **Note:** This code snippet makes use of the Core Text framework. Be sure to add it to your project.

Listing 4-2 Drawing page-based content

```
// Use Core Text to draw the text in a frame on the page.
- (CFRange)renderPage:(NSInteger)pageNum withTextRange:(CFRange)currentRange
                andFramesetter:(CTFramesetterRef)framesetter
{
   // Get the graphics context.
   CGContextRef    currentContext = UIGraphicsGetCurrentContext();

   // Put the text matrix into a known state. This ensures
   // that no old scaling factors are left in place.
   CGContextSetTextMatrix(currentContext, CGAffineTransformIdentity);
```

```
    // Create a path object to enclose the text. Use 72 point
    // margins all around the text.
    CGRect    frameRect = CGRectMake(72, 72, 468, 648);
    CGMutablePathRef framePath = CGPathCreateMutable();
    CGPathAddRect(framePath, NULL, frameRect);

    // Get the frame that will do the rendering.
    // The currentRange variable specifies only the starting point. The framesetter
    // lays out as much text as will fit into the frame.
    CTFrameRef frameRef = CTFramesetterCreateFrame(framesetter, currentRange,
framePath, NULL);
    CGPathRelease(framePath);

    // Core Text draws from the bottom-left corner up, so flip
    // the current transform prior to drawing.
    CGContextTranslateCTM(currentContext, 0, 792);
    CGContextScaleCTM(currentContext, 1.0, -1.0);

    // Draw the frame.
    CTFrameDraw(frameRef, currentContext);

    // Update the current range based on what was drawn.
    currentRange = CTFrameGetVisibleStringRange(frameRef);
    currentRange.location += currentRange.length;
    currentRange.length = 0;
    CFRelease(frameRef);

    return currentRange;
}

- (void)drawPageNumber:(NSInteger)pageNum
{
    NSString *pageString = [NSString stringWithFormat:@"Page %d", pageNum];
    UIFont *theFont = [UIFont systemFontOfSize:12];
```

```
    CGSize maxSize = CGSizeMake(612, 72);

    CGSize pageStringSize = [pageString sizeWithFont:theFont
                                        constrainedToSize:maxSize
                                        lineBreakMode:UILineBreakModeClip];
    CGRect stringRect = CGRectMake(((612.0 - pageStringSize.width) / 2.0),
                                    720.0 + ((72.0 - pageStringSize.height) / 2.0),
                                    pageStringSize.width,
                                    pageStringSize.height);

    [pageString drawInRect:stringRect withFont:theFont];
}
```

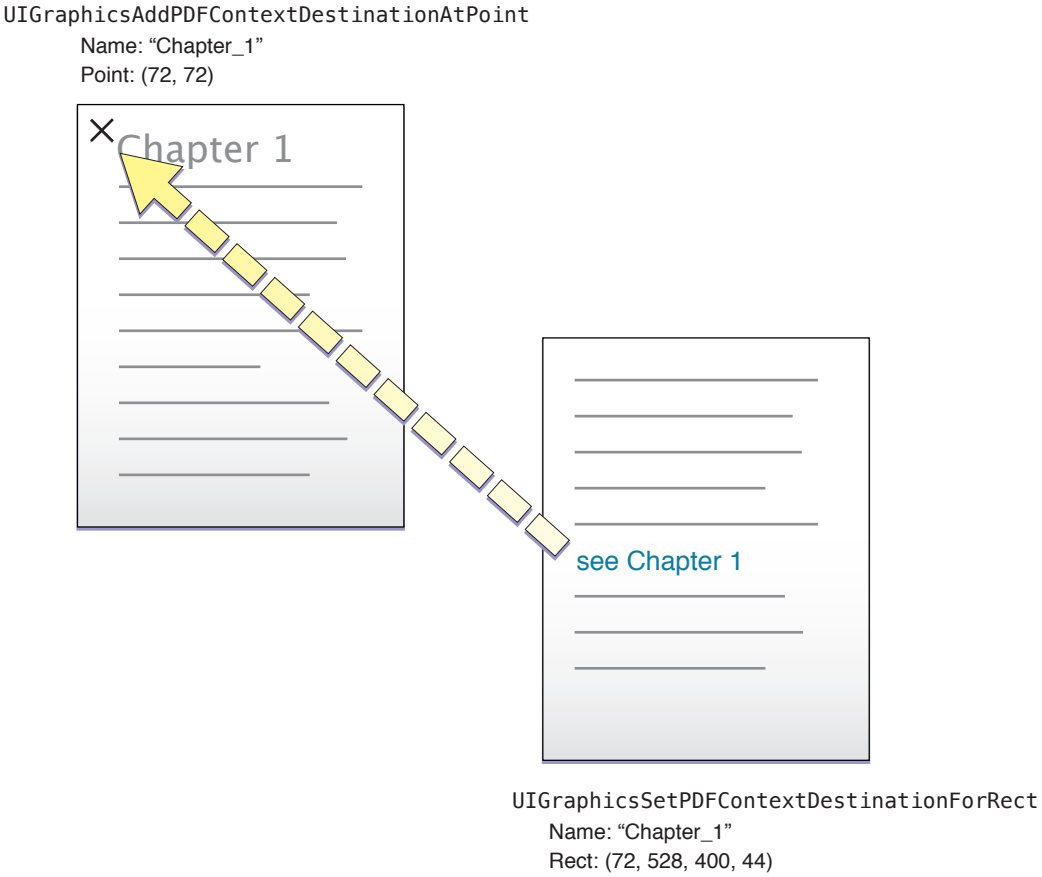
Creating Links Within Your PDF Content

Besides drawing content, you can also include links that take the user to another page in the same PDF file or to an external URL. To create a single link, you must add a source rectangle and a link destination to your PDF pages. One of the attributes of the link destination is a string that serves as the unique identifier for that link. To create a link to a specific destination, you specify the unique identifier for that destination when creating the source rectangle.

To add a new link destination to your PDF content, you use the `UIGraphicsAddPDFContextDestinationAtPoint` function. This function associates a named destination with a specific point on the current page. When you want to link to that destination point, you use `UIGraphicsSetPDFContextDestinationForRect` function to specify the source rectangle for the link.

Figure 4-1 shows the relationship between these two function calls when applied to the pages of your PDF documents. Tapping on the rectangle surrounding the "see Chapter 1" text takes the user to the corresponding destination point, which is located at the top of Chapter 1.

Figure 4-1 Creating a link destination and jump point



In addition to creating links within a document, you can also use the `UIGraphicsSetPDFContextURLForRect` function to create links to content located outside of the document. When using this function to create links, you specify the target URL and the source rectangle on the current page.

Printing

In iOS 4.2 and later, apps can add support for printing content to local AirPrint-capable printers. Although not all apps need printing support, it is often a useful feature if your app is used for creating content (such as a word processor or a drawing program), making purchases (printing order confirmations), and other tasks where the user might reasonably want a permanent record.

This chapter explains how to add printing support to your app. At a high level, your app creates a print job, providing either an array of ready-to-print images and PDF documents, a single image or PDF document, an instance of any of the built-in print formatter classes, or a custom page renderer.

> **Terminology Note:** The notion of a *print job* comes up many times in this chapter. A print job is a unit of work that includes not just the content to be printed but information used in the printing of it, such as the identity of the printer, the name of the print job, and the quality and orientation of printing.

Printing in iOS is Designed to be Simple and Intuitive

To print, users tap a button that is usually in a navigation bar or toolbar that is associated with the view or selected item the user wants to print. The app then presents a view of printing options. The user selects a printer and various options and then requests printing. The app is asked to generate printing output from its content or provide printable data or file URLs. The requested print job is spooled and control returns to the app. If the destination printer is currently not busy, printing begins immediately. If the printer is already printing or if there are jobs before it in the queue, the print job remains in the iOS print queue until it moves to the top of queue and is printed.

The Printing User Interface

The first thing a user sees related to printing is a print button. The print button is often a bar-button item on a navigation bar or a toolbar. The print button should logically apply to the content the app is presenting; if the user taps the button, the app should print that content. Although the print button can be any custom

button, it is recommended that you use the system item-action button shown in Figure 5-1. This is a `UIBarButtonItem` object, specified with the `UIBarButtonSystemItemAction` constant, that you create either in Interface Builder or by calling `initWithBarButtonSystemItem:target:action:`.

Figure 5-1 System item action button—used for printing

When a user taps the print button, a controller object of the app receives the action message. The controller responds by preparing for printing and displaying the printer-options view. The options always include the destination printer (selected from a list of discoverable printers), the number of copies, and sometimes the range of pages to print. If the selected printer is capable of duplex printing, users can choose single-sided or double-sided output. If users decide not to print, they tap outside the options view (on iPad) or tap the Cancel button (on iPhone and iPod touch) to dismiss the printer-options view.

The kind of user interface shown depends on the device. On iPad, the UIKit framework displays a popover view containing the options, as shown in Figure 5-2. An app can animate this view to appear from the print button or from an arbitrary area of the app's user interface.

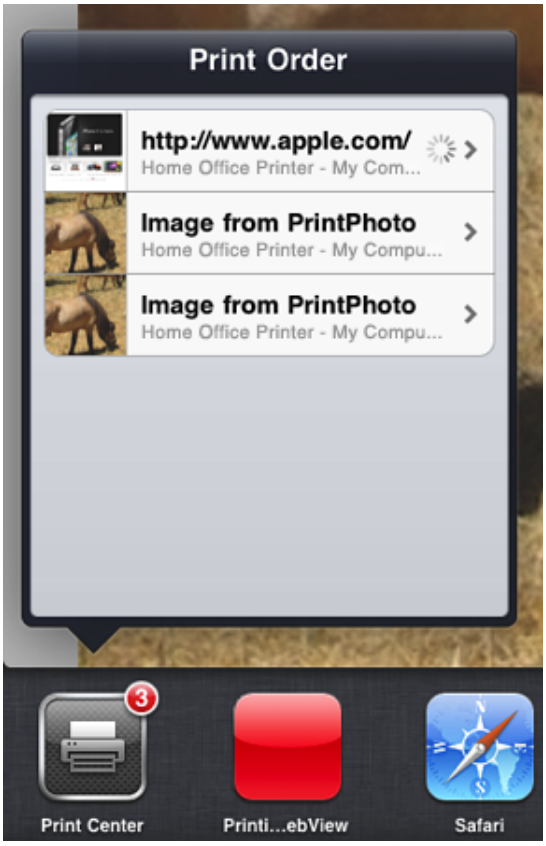
Figure 5-2 Printer-options popover view (iPad)

On iPhone and iPod touch devices, UIKit displays a sheet of printing options that an app can animate to slide up from the bottom of the screen, as shown in Figure 5-3.

Figure 5-3 Printer-options sheet (iPhone)

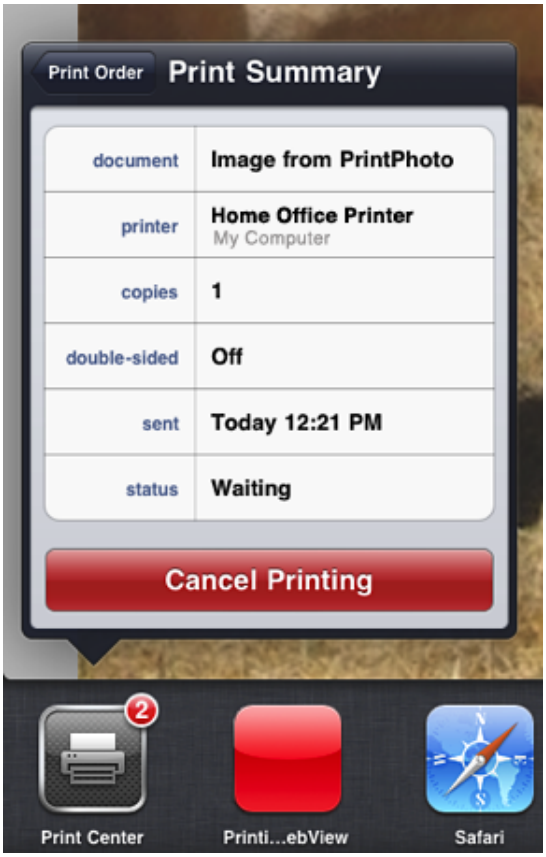
Once a print job has been submitted and is either printing or waiting in the print queue, users can check on its status by double-tapping the Home button to access the Print Center in the multitasking UI. The Print Center (shown in Figure 5-4) is a background system app that shows the order of jobs in the print queue, including those that are currently printing. It is only available while a print job is in progress.

Figure 5-4 Print Center



Users can tap a print job in the Print Center to get detailed information about it (Figure 5-5) and cancel jobs that are printing or waiting in the queue.

Figure 5-5 Print Center: detail of print job



How Printing Works in iOS

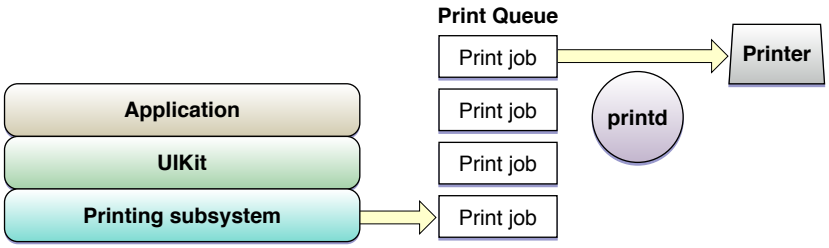
An app uses the UIKit printing API to assemble the elements of a print job, including the content to print and information related to the print job. It then presents the printer-options view described in "The Printing User Interface" (page 52). The user makes his or her choices and then taps Print. In some cases, the UIKit framework then asks the app to draw the content to be printed; UIKit records what the app draws as PDF data. UIKit then hands off the printing data to the printing subsystem.

The printing system does a few things. As UIKit passes the print data to the printing subsystem, it writes this data to storage (that is, it spools the data). It also captures information about the print job. The printing system manages the combined print data and metadata for each print job in a first-in-first-out print queue. Multiple apps on a device can submit multiple print jobs to the printing subsystem, and all of these are placed in the print queue. Each device has one queue for all print jobs regardless of originating app or destination printer.

When a print job rises to the top of the queue, the system printing daemon (`printd`) considers the destination printer's requirements and, if necessary, converts the print data to a form that is usable by the printer. The printing system reports error conditions such as "Out of Paper" to the user as alerts. It also reports the progress of print jobs programmatically to the Print Center, which displays information such as “page 2 of 5” for a print job.

This overall architecture is shown in Figure 5-6.

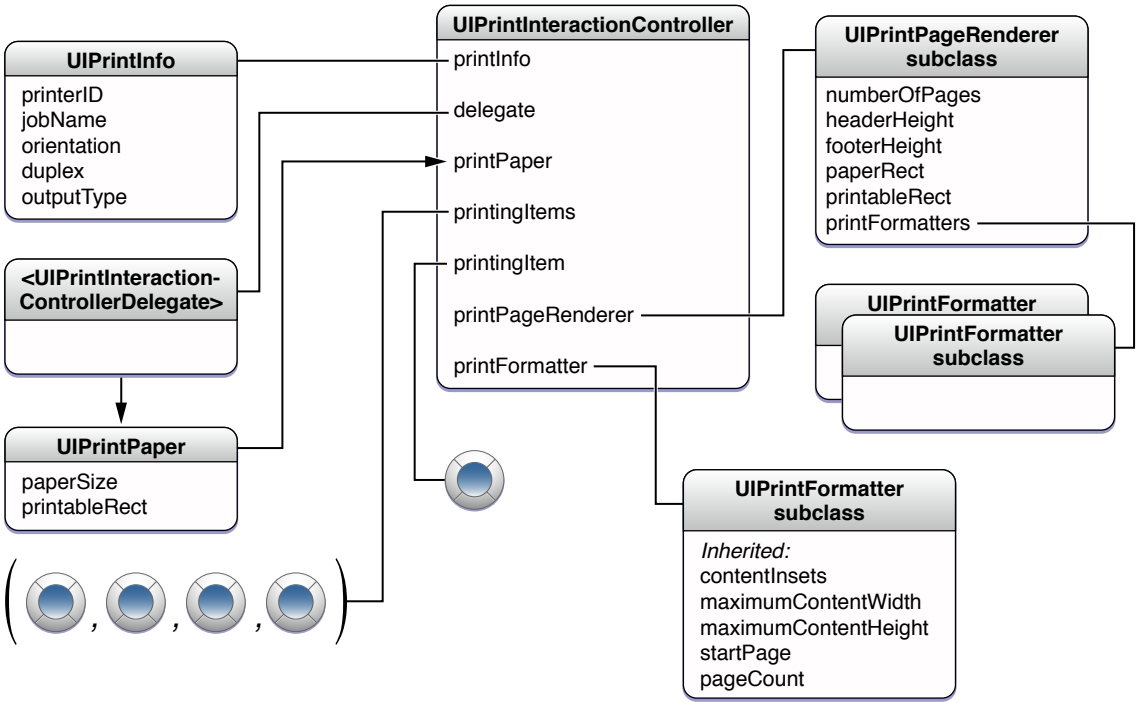
Figure 5-6 Printing architecture



The UIKit Printing API

The UIKit printing API includes eight classes and one formal protocol. Objects of these classes and the delegate implementing the protocol have runtime relationships as depicted in Figure 5-7.

Figure 5-7 Relationships of UIKit printing objects



Printing Support Overview

At a high level, there are two ways to add printing to your app. If you are using a `UIActivityViewController`, and if you do not need the ability to control whether the user can choose a page range or override the paper selection process, you can add a printing activity.

Otherwise, to add printing to your app, you must work with the `UIPrintInteractionController` class. A shared instance of the `UIPrintInteractionController` class provides your app with the ability to specify what should happen when the user tells your app to print. It contains information about the print job (`UIPrintInfo`) and the size of the paper and the area for the printed content (`UIPrintPaper`). It can also have a reference to a delegate object that adopts the `UIPrintInteractionControllerDelegate` protocol for further configuring behavior.

More importantly, the print interaction controller lets your app provide the content to be printed. The `UIPrintInteractionController` class provides three distinct ways to print content:

- Static images or PDFs. For simple content, you can use the print interaction controller's `printingItem` or `printingItems` properties to provide an image (in various formats), a PDF file, or an array of images and PDF files.
- Print formatters. If you need to print text and HTML content with automatic reflow, you can assign an instance of any of the built-in print formatter classes to the print interaction controller's `printFormatter` property.
- Page renderers. Page renderers let you provide your own drawing routines for custom content, and give you complete control over the page layout, including headers and footers. To use a page renderer, you must first write a page renderer subclass, then assign an instance of it to the print interaction controller's `printPageRenderer` property.

> **Important:** These four properties are mutually exclusive; that is, if you assign a value to one of the properties, UIKit makes sure that all the other properties are `nil`.

With such a range of options available to you, what is the best option for your app? Table 5-1 clarifies the factors involved in making this decision.

Table 5-1 Deciding how to print app content

If...	Then...
Your app has access to directly printable content (images or PDF documents).	Use the `printingItem` or `printingItems` properties.

If...	Then...
You want to print a single image or PDF document and want the user to be able to select a page range.	Use the `printingItem` property.
You want to print a plain text or HTML document (and do not want additional content such as headers and footers).	Assign a `UISimpleTextPrintFormatter` or `UIMarkupTextPrintFormatter` object to the `printFormatter` property. The print-formatter object must be initialized with the plain or HTML text.
You want to print the content of a UIKit view (and do not want additional content such as headers and footers).	Get a `UIViewPrintFormatter` object from the view and assign it to the `printFormatter` property.
You want the printed pages to have repeating headers and footers, possibly with incremented page numbers.	Assign an instance of a custom subclass of `UIPrintPageRenderer` to `printPageRenderer`. This subclass should implement the methods required for drawing headers and footers.
You have mixed content or sources that you want to print—for example, HTML and custom drawing.	Assign an instance of `UIPrintPageRenderer` (or a custom subclass thereof) to `printPageRenderer`. You can add one or more print formatters to render specific pages of content. If you are using a custom subclass of `UIPrintPageRenderer`, you also have the option of providing custom drawing code to render some or all of the pages yourself.
You want to have the greatest amount of control over what gets drawn for printing.	Assign an instance of a custom subclass of `UIPrintPageRenderer` to `printPageRenderer` and draw everything that gets printed.

Printing Workflow

The general workflow for printing an image, document, or other printable content of an app is as follows:

1. Obtain the shared instance of `UIPrintInteractionController`.
2. (Optional, but strongly recommended) Create a `UIPrintInfo` object, set attributes such as output type, job name, and print orientation; then assign the object to the `printInfo` property of the `UIPrintInteractionController` instance. (Setting the output type and job name are strongly recommended.)

   If you don't assign a print-info object, UIKit assumes default attributes for the print job (for example, the job name is the app name).

3. (Optional) Assign a custom controller object to the `delegate` property. This object must adopt the `UIPrintInteractionControllerDelegate` protocol. The delegate can do a range of tasks. It can respond appropriately when printing options are presented and dismissed, and when the print job starts and ends. It can return a parent view controller for the print interaction controller.

   Also, by default, UIKit chooses a default paper size and printable area based on the output type, which indicates the kind of content your app is printing. If your app needs more control over paper size, you can override this standard behavior. See "Specifying Paper Size, Orientation, and Duplexing Options" (page 75) for more details.

4. Assign one of the following to a property of the `UIPrintInteractionController` instance:
   - A single `NSData`, `NSURL`, `UIImage`, or `ALAsset` object containing or referencing PDF data or image data to the `printingItem` property.
   - An array of directly printable images or PDF documents to the `printingItems` property. Array elements must be `NSData`, `NSURL`, `UIImage`, or `ALAsset` objects containing, referencing, or representing PDF data or images in supported formats.
   - A `UIPrintFormatter` object to the `printFormatter` property. Print formatters perform custom layout of printable content.
   - A `UIPrintPageRenderer` object to the `printPageRenderer` property.

   Only one of these properties can be non-`nil` for any print job. See "Printing Support Overview" (page 58) for descriptions of these properties.

5. If you assigned a page renderer in the previous step, that object is typically an instance of a custom subclass of `UIPrintPageRenderer`. This object draws pages of content for printing when requested by the UIKit framework. It can also draw content in headers and footers of printed pages. A custom page renderer must override one or more of the "draw" methods and, if it is drawing at least part of the content (excluding headers and footers), it must compute and return the number of pages for the print job. (Note that you can use an instance of `UIPrintPageRenderer` "as-is" to connect a series of print formatters.)

6. (Optional) If you are using a page renderer, you can create one or more `UIPrintFormatter` objects using concrete subclasses of this class; then add the print formatters for specific pages (or page ranges) to the `UIPrintPageRenderer` instance either by calling the `addPrintFormatter:startingAtPageAtIndex:` method of `UIPrintPageRenderer` or by creating an array of one or more print formatters (each with its own starting page) and assigning that array to the `printFormatters` property of `UIPrintPageRenderer`.

7. If the current user-interface idiom is iPad, present the printing interface to the user by calling `presentFromBarButtonItem:animated:completionHandler:` or `presentFromRect:inView:animated:completionHandler:`; if the idiom is iPhone or iPod touch, call `presentAnimated:completionHandler:`. Alternatively, you can embed the printing UI into your existing UI by implementing a `printInteractionControllerParentViewController:` delegate method. If your app uses an activity sheet (in iOS 6.0 and later), you can also add a printing activity item.

From here, the process varies depending on whether you are printing using static content, a print formatter, or a page renderer.

Printing Printer-Ready Content

The iOS printing system accepts certain objects and prints their contents directly, with minimal involvement by the app. These objects are instances of the `NSData`, `NSURL`, `UIImage`, and `ALAsset` classes, and they must contain or reference image data or PDF data. Image data involves all of these object types; PDF data is either referenced by `NSURL` objects or encapsulated by `NSData` objects. There are additional requirements for these printer-ready objects:

- An image must be in a format supported by the Image I/O framework. See "Supported Image Formats" in *UIImage Class Reference* for a list of these formats.
- `NSURL` objects must use as a scheme `file:`, `assets-library:`, or anything that can return an `NSData` with a registered protocol (for example, QuickLook's `x-apple-ql-id:` scheme).
- `ALAsset` objects must be of type `ALAssetTypePhoto`.

You assign printer-ready objects either to the `printingItem` or `printingItems` property of the shared `UIPrintInteractionController` instance. You assign a single printer-ready object to `printingItem` and an array of printer-ready objects to the `printingItems` property.

> **Note:** By providing printer-ready content, you are putting the layout of that content in the hands of the printing system. Thus, settings such as printing orientation have no effect. If your app needs to control layout, you must do the drawing yourself.
>
> Also, if your app uses `printingItems` for its printable content (as opposed to `printingItem`), users cannot specify page ranges in the printer-options view, even if there are multiple pages and the `showsPageRange` property is set to YES.

Before assigning objects to the these properties, you should validate the objects by using one of the class methods of `UIPrintInteractionController`. If, for example, you have the UTI of an image and you want to verify that the image is printer-ready, you can test it first with the `printableUTIs` class method, which returns the set of UTIs that are valid for the printing system:

```
if ([[UIPrintInteractionController printableUTIs] containsObject:mysteryImageUTI])
    printInteractionController.printingItem = mysteryImage;
```

Similarly, you can apply the `canPrintURL:` and `canPrintData:` class methods of `UIPrintInteractionController` to `NSURL` and `NSData` objects prior to assigning those objects to the `printingItem` or `printingItems` properties. These methods determine if the printing system can directly print those objects. Their use is strongly recommended, especially for PDF.

Listing 5-1 shows code that prints a PDF document encapsulated in an `NSData` object. Before assigning it to `printingItem`, it tests the object's validity. It also tells the print interaction controller to include the page-range controls in the printing options presented to the user.

Listing 5-1 Printing a single PDF document with capability for page-range selection

```
- (IBAction)printContent:(id)sender {
    UIPrintInteractionController *pic = [UIPrintInteractionController
sharedPrintController];
    if  (pic && [UIPrintInteractionController canPrintData: self.myPDFData] ) {
        pic.delegate = self;

        UIPrintInfo *printInfo = [UIPrintInfo printInfo];
        printInfo.outputType = UIPrintInfoOutputGeneral;
        printInfo.jobName = [self.path lastPathComponent];
        printInfo.duplex = UIPrintInfoDuplexLongEdge;
        pic.printInfo = printInfo;
        pic.showsPageRange = YES;
        pic.printingItem = self.myPDFData;

        void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError
*) =
           ^(UIPrintInteractionController *pic, BOOL completed, NSError *error) {
             self.content = nil;
             if (!completed && error)
                  NSLog(@"FAILED! due to error in domain %@ with error code %u",
                  error.domain, error.code);
        };
        if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [pic presentFromBarButtonItem:self.printButton animated:YES
            completionHandler:completionHandler];
        } else {
```

```
        [pic presentAnimated:YES completionHandler:completionHandler];
    }
}
```

The procedure for submitting several printer-ready objects at once is identical—except (of course), you must assign an array of these objects to the `printingItems` property:

```
pic.printingItems = [NSArray arrayWithObjects:imageViewOne.image, imageViewTwo.image,
 imageViewThree.image, nil];
```

Using Print Formatters and Page Renderers

Print formatters and page renderers are objects that lay out printable content over multiple pages. They specify the beginning page of content and compute the final page of content based on the starting page, the content area, and the content they lay out. They can also specify the margins relative to the printable area of the page. The difference can be summarized as follows:

- Print formatters lay out a single piece of content (a block of text or HTML, a view, and so on).
- Page renderers let you add headers and footers, perform custom drawing, and so on. Page renderers can, if desired, use print formatters to do most of their work.

  If you create a custom subclass of `UIPrintPageRenderer`, you can draw each page of printable content partially or entirely. A page renderer can have one or more print formatters associated with specific pages or page ranges of the printable content.

Setting the Layout Properties for the Print Job

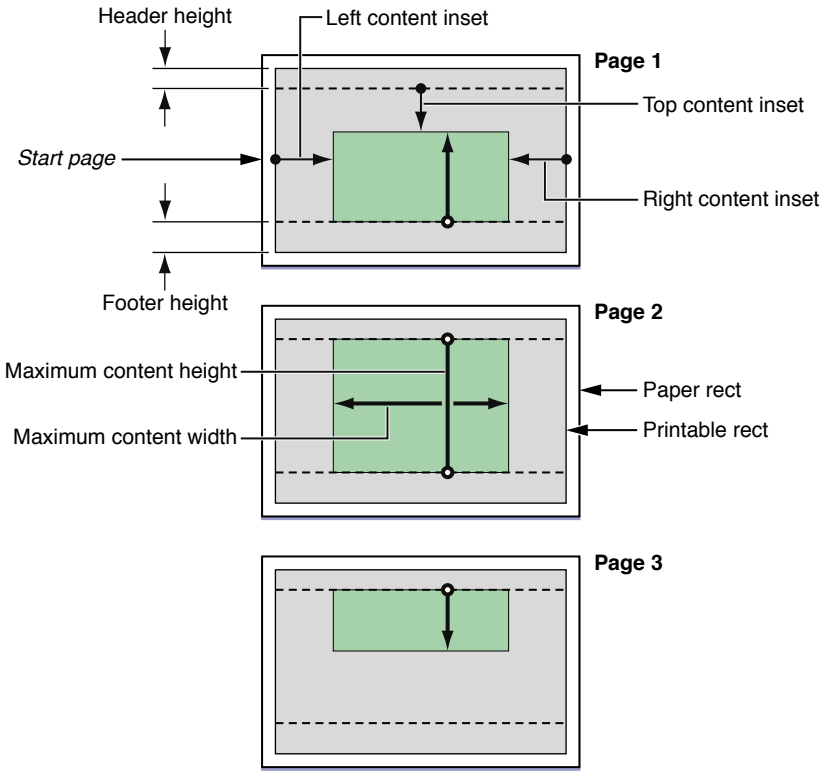
To define the areas on pages for printable content, the `UIPrintFormatter` class declares four key properties for its concrete subclasses. These properties, along with the `footerHeight` and `headerHeight` properties of `UIPrintPageRenderer` and the `paperSize` and `printableRect` properties of `UIPrintPaper`, define the layout of a multi-page print job. Figure 5-8 (page 65) depicts this layout.

Property	Description
`contentInsets`	Distances in points inset from the top, left, and right boundaries of the printable rectangle. These values set the margins of the printed content, although they can be overridden by the `maximumContentHeight` and `maximumContentWidth` values. The top inset applies only to the first page of a given formatter.

Property	Description
`maximumContentHeight`	Specifies the maximum height of the content area, which factors in any header or footer height. UIKit compares this value against the height of the content area minus the top content-inset and uses the lesser of the two values.
`maximumContentWidth`	Specifies the maximum width of the content area. UIKit compares this value against the width of the content area created by the left content-inset and right content-inset and uses the lesser of the two values.
`startPage`	The page on which this formatter should start drawing content for printing. This value is zero-based—the first page of output has a value of 0—but a formatter used by a pager renderer may begin drawing on a later page. For example, to tell a formatter to start printing on the third page, a page renderer would specify 2 for `startPage`.

Note: If you are using the `printFormatter` property of `UIPrintInteractionController`, there is no `UIPrintPageRenderer` object, which means you cannot specify headers or footers on the printed pages.

Figure 5-8 The layout of a multi-page print job



`UIPrintFormatter` uses all the properties depicted in the diagram in Figure 5-8 to compute the number of pages needed for the print job; it stores this value in the read-only `pageCount` property.

Using a Print Formatter

UIKit allows you to assign a single print formatter for a print job. This can be a useful capability if you have plain-text or HTML documents, because UIKit has concrete print-formatter classes for these types of textual content. The framework also implements a concrete print-formatter subclass that enables you to print the content of certain UIKit views in a printer-friendly way.

The `UIPrintFormatter` class is the abstract base class for the system-provided print formatters. Currently, iOS provides the following built-in print formatters:

- `UIViewPrintFormatter`—automatically lays out the content of a view over multiple pages.

  To obtain a print formatter for a view, call the view's `viewPrintFormatter` method.

Not all built-in UIKit classes support printing. Currently, only the view classes `UIWebView`, `UITextView`, and `MKMapView` know how to draw their contents for printing.

View formatters should not be used for printing your own custom views. To print the contents of a custom view, use a `UIPrintPageRenderer` instead.

- `UISimpleTextPrintFormatter`—automatically draws and lays out plain-text documents. This formatter allows you to set global properties for the text, such a font, color, alignment, and line-break mode.
- `UIMarkupTextPrintFormatter`—automatically draws and lays out HTML documents.

Note: The `UIPrintFormatter` class is not intended to be subclassed by third-party developers. If you need to do custom layout, you should write a page renderer instead.

Although the following discussion pertains to the use of a single formatter (and no page renderer), much of the information about print formatters applies to print formatters used in conjunction with page renderers, which is described in "Using One or More Formatters with a Page Renderer" (page 72).

Printing Text or HTML Documents

Many apps include textual content that users might want to print. If the content is plain text or HTML text, and you have access to the backing string for the displayed textual content, you can use an instance of `UISimpleTextPrintFormatter` or `UIMarkupTextPrintFormatter` to lay out and draw the text for printing. Simply create the instance, initializing it with the backing string, and specify the layout properties. Then assign it to the `printFormatter` instance variable of the shared `UIPrintInteractionController` instance.

Listing 5-2 illustrates how you might use a `UIMarkupTextPrintFormatter` object to print an HTML document. It adds an additional inch of margin inside the (printer-defined) printable area. You can determine the printable area by examining the `printPaper` property of the `UIPrintInteractionController` object. For a more complete example of how to create margins of a specific width, see the *UIKit Printing with UIPrintInteractionController and UIViewPrintFormatter* sample code project.

Listing 5-2 Printing an HTML document (without header information)

```
- (IBAction)printContent:(id)sender {
    UIPrintInteractionController *pic = [UIPrintInteractionController
sharedPrintController];
    pic.delegate = self;


    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
```

```
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = self.documentName;
    pic.printInfo = printInfo;

   UIMarkupTextPrintFormatter *htmlFormatter = [[UIMarkupTextPrintFormatter alloc]
        initWithMarkupText:self.htmlString];
    htmlFormatter.startPage = 0;
    htmlFormatter.contentInsets = UIEdgeInsetsMake(72.0, 72.0, 72.0, 72.0); // 1
inch margins
    pic.printFormatter = htmlFormatter;
    pic.showsPageRange = YES;

    void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
        ^(UIPrintInteractionController *printController, BOOL completed, NSError
 *error) {
            if (!completed && error) {
              NSLog(@"Printing could not complete because of error: %@", error);
            }
        };
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [pic presentFromBarButtonItem:sender animated:YES
completionHandler:completionHandler];
    } else {
        [pic presentAnimated:YES completionHandler:completionHandler];
    }
}
```

Remember that if you use a single print formatter for a print job (that is, a `UIPrintFormatter` object assigned to the `printFormatter` property of the `UIPrintInteractionController` instance), you cannot draw header and footer content on each printed page. To do this you must use a `UIPrintPageRenderer` object plus any needed print formatters. For more information, see "Using One or More Formatters with a Page Renderer" (page 72).

The procedure for using a `UISimpleTextPrintFormatter` object to lay out and print a plain text document is almost identical. However, the class of this object includes properties that enable you to set the font, color, and alignment of the printed text.

Using a View Print Formatter

You can use an instance of the `UIViewPrintFormatter` class to lay out and print the contents of some system views. The UIKit framework creates these view print formatters for the view. Often the same code used to draw the view for display is used to draw the view for printing. Currently, the system views whose contents you can print using a view print formatter are instances of `UIWebView`, `UITextView`, and `MKMapView` (MapKit framework).

To get the view print formatter for a `UIView` object, call `viewPrintFormatter` on the view. Set the starting page and any layout properties and then assign the object to the `printFormatter` property of the `UIPrintInteractionController` shared instance. Alternatively, you can add the view print formatter to a `UIPrintPageRenderer` object if you are using that object to draw portions of the printed output. Listing 5-3 shows code that uses a view print formatter from a `UIWebView` object to print the contents of that view.

Listing 5-3 Printing the contents of a web view

```
- (void)printWebPage:(id)sender {
    UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
    void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
        ^(UIPrintInteractionController *printController, BOOL completed, NSError
*error) {
        if(!completed && error){
            NSLog(@"FAILED! due to error in domain %@ with error code %u",
            error.domain, error.code);
        }
    };
    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = [urlField text];
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
    controller.printInfo = printInfo;
    controller.showsPageRange = YES;

    UIViewPrintFormatter *viewFormatter = [self.myWebView viewPrintFormatter];
    viewFormatter.startPage = 0;
    controller.printFormatter = viewFormatter;
```

```
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [controller presentFromBarButtonItem:printButton animated:YES
completionHandler:completionHandler];
    }else
        [controller presentAnimated:YES completionHandler:completionHandler];
}
```

For a complete example based on `UIViewPrintFormatter`, see the *UIKit Printing with UIPrintInteractionController and UIViewPrintFormatter* sample code project.

Using a Page Renderer

A page renderer is an instance of a custom subclass of `UIPrintPageRenderer` that draws the full or partial content of a print job. To use one, you must create the subclass, add it to your project, and instantiate it when you prepare a `UIPrintInteractionController` instance for a print job. Then assign the page renderer to the `printPageRenderer` property of the `UIPrintInteractionController` instance. A page renderer can have one or more print formatters associated with it; if it does, it mixes its drawing with the drawing of the print formatters.

A page renderer can draw and lay out printable content on its own or it can use print formatters to handle some or all of the rendering of specific ranges of pages. Thus, for relatively straightforward formatting needs, you can use an instance of `UIPrintPageRenderer` "as-is" to connect multiple print formatters. However, most page renderers are typically instances of custom subclasses of `UIPrintPageRenderer`.

> **Note:** If you want to print header and footer information, such as a repeating document title and an incrementing page count, you must use a custom subclass of `UIPrintPageRenderer`.

The `UIPrintPageRenderer` base class includes properties for the page count and for heights of headers and footers of pages. It also declares several methods that you can override to draw specific portions of a page: the header, the footer, the content itself, or to integrate page renderer and print formatter drawing.

Setting Page Renderer Attributes

If your page renderer is going to draw in the header or footer of each printed page, you should specify a height for the header and footer. To do this, assign float values (representing points) to the `headerHeight` and `footerHeight` properties inherited by your subclass. If these properties have height values of 0 (the default), the `drawHeaderForPageAtIndex:inRect:` and `drawFooterForPageAtIndex:inRect:` methods are not called.

If your page renderer is going to draw in the content area of the page—that is, the area between any header or footer areas—then your subclass should override the `numberOfPages` method to compute and return the number of pages the page renderer will draw. If print formatters associated with the page renderer are going to draw all of the content between the header and footer, then the print formatters will compute the number of pages for you. This situation occurs your page renderer just draws in the header and footer areas and you let the print formatter draw all the other content.

With a page renderer with no associated print formatter, the layout of each page of printable content is entirely up to you. When computing layout metrics, you can take into account the `headerHeight`, `footerHeight`, `paperRect`, and `printableRect` properties of `UIPrintPageRenderer` (the last two properties are read-only). If the page renderer uses print formatters, the layout metrics also include the `contentInsets`, `maximumContentHeight`, and `maximumContentWidth` properties of `UIPrintFormatter`. See "Setting the Layout Properties for the Print Job" (page 63) for an illustration and explanation.

Implementing the Drawing Methods

When an app uses a page renderer to draw printable content, UIKit calls the following methods for each page of requested content. Note that there is no guarantee that UIKit calls these methods in page-sequence order. Moreover, if users request a subset of pages for printing (that is, they specify a page range), UIKit does not invoke the methods for pages not in the subset.

> **Note:** Remember that if you are drawing images or PDF documents as printable content using non-UIKit API (for example, Quartz API), then you must "flip" the UIKit's coordinate system—putting the origin in the lower-left corner with positive values going upwards—to match the coordinate system used by Core Graphics. See "Coordinate Systems and Drawing in iOS" (page 13) for more information.

The `drawPageAtIndex:inRect:` method calls each of the other draw methods, in the order listed below. Your app can override this method if you want to have complete control over what is drawn for printing.

Override...	To...
`drawHeaderForPageAtIndex:inRect:`	Draw content in the header. This method is not called if headerHeight is 0.
`drawContentForPageAtIndex:inRect:`	Draw the content of the print job (that is, the area between the header and the footer).

Override...	To...
`drawPrintFormatter: forPageAtIndex:`	Intermix custom drawing with the drawing performed by print formatters. This method is called for each print formatter associated with a given page. See "Using One or More Formatters with a Page Renderer" (page 72) for more information.
`drawFooterForPageAtIndex: inRect:`	Draw content in the footer. This method is not called if footerHeight is 0.

All of these drawing methods are set up for drawing to the current graphics context (as returned by `UIGraphicsGetCurrentContext`). The rectangle passed into each method—defining header area, footer area, content area, and entire page—has values relative to the origin of the page, which is in the upper-left corner.

Listing 5-4 shows example implementations of the `drawHeaderForPageAtIndex:inRect:` and `drawFooterForPageAtIndex:inRect:` methods. They use `CGRectGetMaxX` and `CGRectGetMaxY` to compute placement of text in the `footerRect` and `headerRect` rectangles in the coordinate system of the printable rectangle.

Listing 5-4 Drawing the header and footer of a page

```
- (void)drawHeaderForPageAtIndex:(NSInteger)pageIndex  inRect:(CGRect)headerRect
{
    UIFont *font = [UIFont fontWithName:@"Helvetica" size:12.0];
    CGSize titleSize = [self.jobTitle sizeWithFont:font];
    //center title in header
    CGFloat drawX = CGRectGetMaxX(headerRect)/2 - titleSize.width/2;
    CGFloat drawY = CGRectGetMaxY(headerRect) - titleSize.height;
    CGPoint drawPoint = CGPointMake(drawX, drawY);
    [self.jobTitle drawAtPoint:drawPoint withFont: font];
}

- (void)drawFooterForPageAtIndex:(NSInteger)pageIndex  inRect:(CGRect)footerRect
{
    UIFont *font = [UIFont fontWithName:@"Helvetica" size:12.0];
    NSString *pageNumber = [NSString stringWithFormat:@"%d.", pageIndex+1];
    // page number at right edge of footer rect
    CGSize pageNumSize = [pageNumber sizeWithFont:font];
```

```
    CGFloat drawX = CGRectGetMaxX(footerRect) - pageNumSize.width - 1.0;
    CGFloat drawY = CGRectGetMaxY(footerRect) - pageNumSize.height;
    CGPoint drawPoint = CGPointMake(drawX, drawY);
    [pageNumber drawAtPoint:drawPoint withFont: font];
}
```

Using One or More Formatters with a Page Renderer

A page renderer can draw printable content in conjunction with one or more print formatters. For example, an app can use a `UISimpleTextPrintFormatter` object to draw pages of textual content for printing, but use a page renderer to draw a document title in each page header. Or an app can use two print formatters, one to draw header (or summary) information at the beginning of the first page and another print formatter to draw the remaining content; then it might use a page renderer to draw a line separating the two parts.

As you may recall, you can use a single print formatter for a print job by assigning it the `printFormatter` property of the `UIPrintInteractionController` shared instance. But if you use a page renderer *and* print formatters, you must associate each print formatter with the page renderer. You do this one of two ways:

- Include each print formatter in the array assigned to the `printFormatters` property.
- Add each print formatter by calling the `addPrintFormatter:startingAtPageAtIndex:` method.

Before you associate a print formatter with a page renderer, be sure to set its layout properties, including the starting page (`startPage`) of the print job. Once you have set these properties, `UIPrintFormatter` computes the number of pages for the print formatter. Note that if you specify a starting page by calling `addPrintFormatter:startingAtPageAtIndex:`; that value overwrites any value assigned to `startPage`. For a discussion of print formatters and layout metrics, see "Setting the Layout Properties for the Print Job" (page 63).

A page renderer can override `drawPrintFormatter:forPageAtIndex:` to integrate its drawing with the drawing performed by the print formatters assigned to a given page. It can draw in an area of the page where the print formatter doesn't draw and then call the `drawInRect:forPageAtIndex:` method on the passed-in print formatter to have it draw its portion of the page. Or the page renderer can achieve an "overlay" effect by having the print formatter draw first and then drawing something over the content drawn by the print formatter.

For complete examples based on `UIPrintPageRenderer`, see the *PrintPhoto*, *Recipes and Printing*, and *UIKit Printing with UIPrintInteractionController and UIViewPrintFormatter* sample code projects.

Testing the Printing of App Content

The iOS 4.2 SDK (and later) provides a Printer Simulator app that you can use to test your app's printing capabilities. The app simulates printers of various general types (inkjet, black-and-white laser, color laser, and so on). It displays printed pages in the OS X Preview app. You can set a preference to show the printable area of each page. Printer Simulator also logs information from the printing system about each print job.

You can run the Printer Simulator in one of thee ways:

- Choose Open Printer Simulator from the File menu in the iOS Simulator.
- Choose the Printer Simulator in Xcode using the Xcode > Open Developer Tool menu.
- You can find Printer Simulator in the following file-system location:

```
<Xcode>/Platforms/iPhoneOS.platform/Developer/Applications/PrinterSimulator.app
```

When testing your app's printing code, you should also implement the completion handler passed into the `present...` methods and log any errors returned from the printing system. These errors are typically programming errors, which you should catch before your app is deployed. See "Responding to Print-Job Completion and Errors" (page 79) for details.

Common Printing Tasks

All the coding tasks described below are things that an app does (or can do) in response to a request for printing. Although most of the tasks can occur in any order, you should first check that the device is capable of printing and you should conclude with presenting the printing options. See "Printing Printer-Ready Content" (page 61), "Using Print Formatters and Page Renderers" (page 63), and "Using a Page Renderer" (page 69) for complete examples.

An important set of tasks not covered here is adding a print button to an appropriate place in the app's user interface, declaring an action method, making a target-action connection, and implementing the action method. (See "The Printing User Interface" (page 52) for a recommendation on which print button to use.) The following tasks (except for "Specifying Paper Size, Orientation, and Duplexing Options" (page 75)) are part of the implementation of the action method.

Testing for Printing Availability

Some iOS devices do not support printing. You should immediately determine this fact once your view loads. If printing is not available for the device, you should either not programmatically add any printing user-interface element (button, bar button item, and so on), or you should remove any printing element loaded from a nib

file. To determine if printing is available, call the `isPrintingAvailable` class method of `UIPrintInteractionController`. Listing 5-5 illustrates how you might do this; it assumes an outlet to a print button (`myPrintButton`) loaded from a nib file.

Listing 5-5 Enabling or disabling a print button based on availability of printing

```
- (void)viewDidLoad {
    if (![UIPrintInteractionController isPrintingAvailable])
        [myPrintButton removeFromSuperView];
    // other tasks...
}
```

> **Note:** Although you could disable a printing element such as an "action" bar button item, removal is recommended. The value returned by `isPrintingAvailable` never changes for a given device because it reflects whether that device supports printing, not whether printing is currently available.

Specifying Print-Job Information

An instance of the `UIPrintInfo` class encapsulates information about a print job, specifically:

- The output type (indicating the type of content)
- The print-job name
- The printing orientation
- The duplex mode
- The identifier of the selected printer

You do not need to assign values to all `UIPrintInfo` properties; users choose some of these values and UIKit assumes default values for others. (Indeed, you don't even have to explicitly create an instance of `UIPrintInfo`.)

However, in most cases you'll want to specify some aspects of a print job, such as output type. Get an instance of `UIPrintInfo` by calling the `printInfo` class method. Assign values to the object's properties you want to configure. Then assign the `UIPrintInfo` object to the `printInfo` property of the shared `UIPrintInteractionController` instance. Listing 5-6 gives an example of this procedure.

Listing 5-6 Setting properties of a `UIPrintInfo` object and assigning it to the `printInfo` property

```
UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
```

```
UIPrintInfo *printInfo = [UIPrintInfo printInfo];
printInfo.outputType = UIPrintInfoOutputGeneral;
printInfo.jobName = [self.path lastPathComponent];
printInfo.duplex = UIPrintInfoDuplexLongEdge;
controller.printInfo = printInfo;
```

One of the `UIPrintInfo` properties is the printing orientation: portrait or landscape. You might want the printing orientation to suit the dimensions of the object being printed. In other words, if the object is large, and its width is greater than its height, landscape would be a suitable printing orientation for it. Listing 5-7 illustrates this with an image.

Listing 5-7 Setting the printing orientation to match image dimension

```
  UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
// other code here...
  UIPrintInfo *printInfo = [UIPrintInfo printInfo];
  UIImage *image = ((UIImageView *)self.view).image;
  printInfo.outputType = UIPrintInfoOutputPhoto;
  printInfo.jobName = @"Image from PrintPhoto";
  printInfo.duplex = UIPrintInfoDuplexNone;
  // only if drawing...
  if (!controller.printingItem && image.size.width > image.size.height)
    printInfo.orientation = UIPrintInfoOrientationLandscape;
```

Specifying Paper Size, Orientation, and Duplexing Options

By default, UIKit presents a set of default paper sizes for printable content based on the destination printer and the output type of the print job, as specified by the `outputType` property of the `UIPrintInfo` object.

For example, if the output type is `UIPrintInfoOutputPhoto`, the default paper size is 4 x 6 inches, A6, or some other standard size, depending on locale; if the output type is `UIPrintInfoOutputGeneral` or `UIPrintInfoOutputGrayscale`, the default paper size is US Letter (8 1/2 x 11 inches), A4, or some other standard size, depending on locale.

For most apps, these default paper sizes are acceptable. However, some apps might need a special paper size. A page-based app might need to show the user how content will actually appear on paper of a given size, an app that produces brochures or greeting cards might have its own preferred size, and so on.

In this case, the delegate of the print interaction controller can implement the `UIPrintInteractionControllerDelegate` protocol method `printInteractionController:choosePaper:` to return a `UIPrintPaper` object representing the optimal combination of available paper size and printable rectangle for a given content size.

The delegate has two approaches it can take. It can examine the passed-in array of `UIPrintPaper` objects and identify the one that is most suitable. Or it can let the system pick the most suitable object by calling the `UIPrintPaper` class method `bestPaperForPageSize:withPapersFromArray:`. Listing 5-8 shows an implementation of the method for an app that supports multiple document types, each with its own page size.

Listing 5-8 Implementing the `printInteractionController:choosePaper:` method

```
- (UIPrintPaper *)printInteractionController:(UIPrintInteractionController *)pic
    choosePaper:(NSArray *)paperList {
    // custom method & properties...
    CGSize pageSize = [self pageSizeForDocumentType:self.document.type];
    return [UIPrintPaper bestPaperForPageSize:pageSize
        withPapersFromArray:paperList];
}
```

Typically, apps that use custom page renderers factor the paper size into their calculations of the number of pages for a print job (`numberOfPages`).

If your app needs to present the user with a choice of page size (for a word processing app, for example), you must implement that UI yourself, and you must then use that paper size in your `printInteractionController:choosePaper:` implementation. For example:

```
// Create a custom CGSize for 8.5" x 11" paper.
CGSize custompapersize = CGSizeMake(8.5 * 72.0, 11.0 * 72.0);
```

The `UIPrintInfo` class also lets you provide additional settings, such as the printing orientation, selected printer, and the duplexing mode (if the printer supports duplex printing). Users can change the selected printer and duplex settings from the values you provide

Integrating Printing Into Your User Interface

There are two ways to integrate printing into your user interface:

- Using a print interaction controller.
- From the activity sheet (in iOS 6.0 and later).

The way you integrate printing depends on which of these techniques you choose.

Presenting Printing Options Using a Print Interaction Controller

`UIPrintInteractionController` declares three methods for presenting the printing options to users, each with its own animation:

- `presentFromBarButtonItem:animated:completionHandler:` animates a popover view from a button in the navigation bar or toolbar (usually the print button).
- `presentFromRect:inView:animated:completionHandler:` animates a popover view from an arbitrary rectangle in the app's view.
- `presentAnimated:completionHandler:` animates a sheet that slides up from the bottom of the screen.

The first two of these methods are intended to be invoked on iPad devices; the third method is intended to be invoked on iPhone and iPod touch devices. You can conditionally code for each device type (or, user-interface idiom) by calling the `UI_USER_INTERFACE_IDIOM` and comparing the result to `UIUserInterfaceIdiomPad` or `UIUserInterfaceIdiomPhone`. Listing 5-9 gives an example of this.

Listing 5-9 Presenting printing options based upon current device type

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    [controller presentFromBarButtonItem:self.printButton animated:YES
        completionHandler:completionHandler];
} else {
    [controller presentAnimated:YES completionHandler:completionHandler];
}
```

If your app calls one of the iPad-specific methods on an iPhone (and requests animation), the default behavior is to display the printing options in a sheet that slides up from the bottom of the screen. If your app calls the iPhone-specific method on iPad, the default behavior is to animate the popover view from the current window frame.

If you call one of the `present...` methods when the printing options are already displayed, `UIPrintInteractionController` hides the printing-options view or sheet. You must call the method again to display the options.

If you assign a printer ID or a duplex mode as print-info values, these appear as defaults in the printing options. (The printer must be capable of double-sided printing for the duplex control to appear.) If you want to let your users select ranges of pages for printing, you should set the `showsPageRange` property of the `UIPrintInteractionController` object to `YES` (`NO` is the default value). Be aware, however, that no page-range control appears in the printing options if you supply printable content via the `printingItems` property or if the total number of pages is 1, even if `showsPageRange` is `YES`.

If you want the printing UI to appear within a particular view, you can do this by implementing the `printInteractionControllerParentViewController:` method in your `UIPrintInteractionControllerDelegate` class. This method should return the view controller that should be used as the parent for the print interaction controller. If the provided parent view controller is a `UINavigationController` instance, the printing UI is pushed into view in that controller. For any other `UIViewController` instance, the print navigation is shown as a modal dialog within the specified view.

Printing From the Activity Sheet

If your app uses an activity sheet (in iOS 6.0 and later), you can allow printing from the activity sheet. Printing from the activity sheet is simpler than using a print interaction controller. There are two caveats, however:

- The app cannot control whether a user can choose a page range.
- The app cannot use delegate methods to override behavior, such as manually overriding the paper size selection process.

To use this technique, your app must create an activity items array containing:

- A `UIPrintInfo` object.
- Either a page renderer formatter object, a page renderer or a printable item.
- Any additional activity items appropriate for your app.

You then call `initWithActivityItems:applicationActivities:`, passing that array as the first parameter and `nil` as the second parameter (or an array of custom activities, if your app provides any).

Finally, you present the activity view using the standard view controller `present*` methods. If the user chooses to print from that activity view, iOS creates a print job for you. For more information, read *UIActivityViewController Class Reference* and *UIActivity Class Reference*.

Responding to Print-Job Completion and Errors

The final parameter of the presentation methods declared by `UIPrintInteractionController` and described in "Presenting Printing Options Using a Print Interaction Controller " (page 77) is a completion handler. The completion handler is a block of type `UIPrintInteractionCompletionHandler` that is invoked when a print job completes successfully or when it is terminated because of an error. You can provide a block implementation that cleans up state that you have set up for the print job. The completion handler can also log error messages.

The example in Listing 5-10 clears an internal property upon completion or error; if there is an error, it logs information about it.

Listing 5-10 Implementing a completion-handler block

```
void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
        ^(UIPrintInteractionController *pic, BOOL completed, NSError *error) {
    self.content = nil;
    if (!completed && error)
        NSLog(@"FAILED! due to error in domain %@ with error code %u",
            error.domain, error.code);
  };
```

UIKit automatically releases all objects assigned to the `UIPrintInteractionController` instance at the end of a print job (except for the delegate), so you do not have to do this yourself in a completion handler.

A printing error is represented by an `NSError` object having a domain of `UIPrintErrorDomain` and an error code declared in `UIPrintError.h`. In almost all cases, these codes indicate programming errors, so usually there is no need to inform the user about them. However, some errors can result from an attempt to print a file (located by a file-scheme `NSURL` object) that results in an error.

Improving Drawing Performance

Drawing is a relatively expensive operation on any platform, and optimizing your drawing code should always be an important step in your development process. Table A-1 lists several tips for ensuring that your drawing code is as optimal as possible. In addition to these tips, you should always use the available performance tools to test your code and remove hotspots and redundancies.

Table A-1 Tips for improving drawing performance

Tip	Action
Draw minimally	During each update cycle, you should update only the portions of your view that actually changed. If you are using the `drawRect:` method of `UIView` to do your drawing, use the update rectangle passed to that method to limit the scope of your drawing. For OpenGL drawing, you must track updates yourself.
Call `setNeedsDisplay:` judiciously	If you are calling `setNeedsDisplay:`, always spend the time to calculate the actual area that you need to redraw. Don't just pass a rectangle containing the entire view. Also, don't call `setNeedsDisplay:` unless you actually need to redraw content. If the content hasn't actually changed, don't redraw it.
Mark opaque views as such	Compositing a view whose contents are opaque requires much less effort than compositing one that is partially transparent. To make a view opaque, the contents of the view must not contain any transparency and the `opaque` property of the view must be set to `YES`.
Reuse table cells and views during scrolling	Creating new views during scrolling should be avoided at all costs. Taking the time to create new views reduces the amount of time available for updating the screen, which leads to uneven scrolling behavior.
Reuse paths by modifying the current transformation matrix	By modifying the current transformation matrix, you can use a single path to draw content on different parts of the screen. For details, see "Using Coordinate Transforms to Improve Drawing Performance" (page 24).
Avoid clearing the previous content during scrolling	By default, UIKit clears a view's current context buffer prior to calling its `drawRect:` method to update that same area. If you are responding to scrolling events in your view, clearing this region repeatedly during scrolling updates can be expensive. To disable the behavior, you can change the value in the `clearsContextBeforeDrawing` property to `NO`.

Tip	Action
Minimize graphics state changes while drawing	Changing the graphics state requires work by the underlying graphics subsystems. If you need to draw content that uses similar state information, try to draw that content together to reduce the number of state changes needed.
Use Instruments to debug your performance	The Core Animation instrument can help you spot drawing performance problems in your app. In particular: • Flash Updated Regions makes it easy to see what parts of your view are actually being updated. • Color Misaligned Images helps you see images that are aligned poorly, which results in both fuzzy images and poor performance. For more information, see "Measuring Graphics Performance in Your iOS Device" in *Instruments User Guide*.

Supporting High-Resolution Screens In Views

Apps built against iOS SDK 4.0 and later need to be prepared to run on devices with different screen resolutions. Fortunately, iOS makes supporting multiple screen resolutions easy. Most of the work of handling the different types of screens is done for you by the system frameworks. However, your app still needs to do some work to update raster-based images, and depending on your app you may want to do additional work to take advantage of the extra pixels available to you.

See "Points Versus Pixels" (page 16) for important background information related to this topic.

Checklist for Supporting High-Resolution Screens

To update your apps for devices with high-resolution screens, you need to do the following:

- Provide a high-resolution image for each image resource in your app bundle, as described in "Updating Your Image Resource Files" (page 83).
- Provide high-resolution app and document icons, as described in "Updating Your App's Icons and Launch Images" (page 85).
- For vector-based shapes and content, continue using your custom Core Graphics and UIKit drawing code as before. If you want to add extra detail to your drawn content, see "Points Versus Pixels" (page 16) for information on how to do so.
- If you use OpenGL ES for drawing, decide whether you want to opt in to high-resolution drawing and set the scale factor of your layer accordingly, as described in "Drawing High-Resolution Content Using OpenGL ES or GLKit" (page 85).
- For custom images that you create, modify your image-creation code to take the current scale factor into account, as described in "Drawing to Bitmap Contexts and PDF Contexts" (page 19).
- If your app uses Core Animation, adjust your code as needed to compensate for scale factors, as described in "Accounting for Scale Factors in Core Animation Layers" (page 28).

Drawing Improvements That You Get for Free

The drawing technologies in iOS provide a lot of support to help you make your rendered content look good regardless of the resolution of the underlying screen:

- Standard UIKit views (text views, buttons, table views, and so on) automatically render correctly at any resolution.
- Vector-based content (`UIBezierPath`, `CGPathRef`, PDF) automatically takes advantage of any additional pixels to render sharper lines for shapes.
- Text is automatically rendered at higher resolutions.
- UIKit supports the automatic loading of high-resolution variants (`@2x`) of your images.

If your app uses only native drawing technologies for its rendering, the only thing you need to do to support higher-resolution screens is provide high-resolution versions of your images.

Updating Your Image Resource Files

Apps running in iOS 4 should now include two separate files for each image resource. One file provides a standard-resolution version of a given image, and the second provides a high-resolution version of the same image. The naming conventions for each pair of image files is as follows:

- Standard: <*ImageName*><*device_modifier*>`.`<*filename_extension*>
- High resolution: <*ImageName*>`@2x`<*device_modifier*>`.`<*filename_extension*>

The <*ImageName*> and <*filename_extension*> portions of each name specify the usual name and extension for the file. The <*device_modifier*> portion is optional and contains either the string `~ipad` or `~iphone`. You include one of these modifiers when you want to specify different versions of an image for iPad and iPhone. The inclusion of the `@2x` modifier for the high-resolution image is new and lets the system know that the image is the high-resolution variant of the standard image.

> **Important:** The order of the modifiers is critical. If you incorrectly put the `@2x` after the device modifier, iOS will not find the image.

When creating high-resolution versions of your images, place the new versions in the same location in your app bundle as the original.

Loading Images into Your App

The `UIImage` class handles all of the work needed to load high-resolution images into your app. When creating new image objects, you use the same name to request both the standard and the high-resolution versions of your image. For example, if you have two image files, named `Button.png` and `Button@2x.png`, you would use the following code to request your button image:

```
UIImage *anImage = [UIImage imageNamed:@"Button"];
```

Note: In iOS 4 and later, you may omit the filename extension when specifying image names.

On devices with high-resolution screens, the `imageNamed:`, `imageWithContentsOfFile:`, and `initWithContentsOfFile:` methods automatically looks for a version of the requested image with the `@2x` modifier in its name. If it finds one, it loads that image instead. If you do not provide a high-resolution version of a given image, the image object still loads a standard-resolution image (if one exists) and scales it during drawing.

When it loads an image, a `UIImage` object automatically sets the `size` and `scale` properties to appropriate values based on the suffix of the image file. For standard resolution images, it sets the `scale` property to 1.0 and sets the size of the image to the image's pixel dimensions. For images with the `@2x` suffix in the filename, it sets the `scale` property to 2.0 and halves the width and height values to compensate for the scale factor. These halved values correlate correctly to the point-based dimensions you need to use in the logical coordinate space to render the image.

Note: If you use Core Graphics to create an image, remember that Quartz images do not have an explicit scale factor, so their scale factor is assumed to be 1.0. If you want to create a `UIImage` object from a `CGImageRef` data type, use the `initWithCGImage:scale:orientation:` to do so. That method allows you to associate a specific scale factor with your Quartz image data.

A `UIImage` object automatically takes its scale factor into account during drawing. Thus, any code you have for rendering images should work the same as long as you provide the correct image resources in your app bundle.

Using an Image View to Display Multiple Images

If your app uses the `UIImageView` class to present multiple images for a highlight or animation, all of the images you assign to that view must use the same scale factor. You can use an image view to display a single image or to animate several images, and you can also provide a highlight image. Therefore, if you provide high-resolution versions for one of these images, then all must have high-resolution versions as well.

Updating Your App's Icons and Launch Images

In addition to updating your app's custom image resources, you should also provide new high-resolution icons for your app's icon and launch images. The process for updating these image resources is the same as for all other image resources. Create a new version of the image, add the `@2x` modifier string to the corresponding image filename, and treat the image as you do the original. For example, for app icons, add the high-resolution image filename to the `CFBundleIconFiles` key of your app's `Info.plist` file.

For information about specifying the icons and launch images for your app, see "App-Related Resources" in *iOS App Programming Guide*.

Drawing High-Resolution Content Using OpenGL ES or GLKit

If your app uses OpenGL ES or GLKit for rendering, your existing drawing code should continue to work without any changes. When drawn on a high-resolution screen, though, your content is scaled accordingly and will appear more blocky. The reason for the blocky appearance is that the default behavior of the `CAEAGLLayer` class, which you use to back your OpenGL ES renderbuffers (directly or indirectly), is the same as other Core Animation layer objects. In other words, its scale factor is set to 1.0 initially, which causes the Core Animation compositor to scale the contents of the layer on high-resolution screens. To avoid this blocky appearance, you need to increase the size of your OpenGL ES renderbuffers to match the size of the screen. (With more pixels, you can then increase the amount of detail you provide for your content.) Because adding more pixels to your renderbuffers has performance implications, though, you must explicitly opt in to support high-resolution screens.

To enable high-resolution drawing, you must change the scale factor of the view you use to present your OpenGL ES or GLKit content. Changing the `contentScaleFactor` property of your view from 1.0 to 2.0 triggers a matching change to the scale factor of the underlying `CAEAGLLayer` object. The `renderbufferStorage:fromDrawable:` method, which you use to bind the layer object to your renderbuffers, calculates the size of the render buffer by multiplying the layer's bounds by its scale factor. Thus, doubling the scale factor doubles the width and height of the resulting render buffer, giving you more pixels for your content. After that, it is up to you to provide the content for those additional pixels.

Listing B-1 shows the proper way to bind your layer object to your renderbuffers and retrieve the resulting size information. If you used the OpenGL ES app template to create your code, then this step is already done for you, and the only thing you need to do is set the scale factor of your view appropriately. If you did not use the OpenGL ES app template, you should use code similar to this to retrieve the render buffer size. You should never assume that the render buffer size is fixed for a given type of device.

Listing B-1 Initializing a render buffer's storage and retrieving its actual dimensions

```
GLuint colorRenderbuffer;
```

```
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];

// Get the renderbuffer size.
GLint width;
GLint height;
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_WIDTH_OES,
&width);
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_HEIGHT_OES,
&height);
```

Important: A view that is backed by a `CAEAGLLayer` object should not implement a custom `drawRect:` method. Implementing a `drawRect:` method causes the system to change the default scale factor of the view so that it matches the scale factor of the screen. If your drawing code is not expecting this behavior, your app content will not be rendered correctly.

If you do opt in to high-resolution drawing, you also need to adjust the model and texture assets of your app accordingly. For example, when running on iPad or on a high-resolution device, you might want to choose larger models and more detailed textures to take advantage of the increased number of pixels. Conversely, on a standard-resolution iPhone, you can continue to use smaller models and textures.

An important factor when determining whether to support high-resolution content is performance. The quadrupling of pixels that occurs when you change the scale factor of your layer from 1.0 to 2.0 puts additional pressure on the fragment processor. If your app performs many per-fragment calculations, the increase in pixels may reduce your app's frame rate. If you find your app runs significantly slower at the higher scale factor, consider one of the following options:

- Optimize your fragment shader's performance using the performance-tuning guidelines found in *OpenGL ES Programming Guide for iOS*.
- Choose a simpler algorithm to implement in your fragment shader. By doing so, you are reducing the quality of each individual pixel to render the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and 2.0. A scale factor of 1.5 provides better quality than a scale factor of 1.0 but needs to fill fewer pixels than an image scaled to 2.0.
- OpenGL ES in iOS 4 and later offers multisampling as an option. Even though your app can use a smaller scale factor (even 1.0), implement multisampling anyway. An added advantage is that this technique also provides higher quality on devices that do not support high-resolution displays.

The best solution depends on the needs of your OpenGL ES app; you should test more than one of these options and choose the approach that provides the best balance between performance and image quality.

Loading Images

For both functional and aesthetic reasons, images are a pervasive element of app user interfaces. They can be a key differentiating factor for apps.

Many images used by apps, including launch images and app icons, are stored as files in the app's main bundle. You can have launch images and icons that are specific to device type (iPad versus iPhone and iPod touch) and that are optimized for high-resolution displays. You can find full descriptions of these bundled image files in "Advanced App Tricks" and "App-Related Resources" in *iOS App Programming Guide*. "Updating Your Image Resource Files" (page 83) discusses adjustments that make your image files compatible with high-resolution screens.

In addition, iOS provides support for loading and displaying images using both the UIKit and Core Graphics frameworks. How you determine which classes and functions to use to draw images depends on how you intend to use them. Whenever possible, though, it is recommended that you use the classes of UIKit for representing images in your code. Table C-1 lists some of the usage scenarios and the recommended options for handling them.

Table C-1 Usage scenarios for images

Scenario	Recommended usage
Display an image as the content of a view	Use the `UIImageView` class to display the image. This option assumes that your view's only content is an image. You can still layer other views on top of the image view to draw additional controls or content.
Display an image as an adornment for part of a view	Load and draw the image using the `UIImage` class.
Save some bitmap data into an image object	You can do this using the UIKit functions or Core Graphics functions described in "Creating New Images Using Bitmap Graphics Contexts" (page 41).
Save an image as a JPEG or PNG file	Create a `UIImage` object from the original image data. Call the `UIImageJPEGRepresentation` or `UIImagePNGRepresentation` function to get an `NSData` object, and use that object's methods to save the data to a file.

System Support for Images

The UIKit framework as well as the lower-level system frameworks of iOS give you a wide range of possibilities for creating, accessing, drawing, writing, and manipulating images.

UIKit Image Classes and Functions

The UIKit framework has three classes and one protocol that are related to images in some way:

`UIImage`
: Objects of this class represent images in the UIKit framework. You can create them from several different sources, including files and Quartz image objects. Methods of the class enable you to draw images to the current graphics context using different blend modes and opacity values.

  The `UIImage` class automatically handles any required transformations for you, such as applying the proper scale factor (taking into consideration high-resolution displays) and, when given Quartz images, modifying the coordinate system of the image so that it matches the default coordinate system of UIKit (where the y origin is at the upper left).

`UIImageView`
: Objects of this class are views that display either a single image or animate a series of images. If an image is to be the sole content of a view, use the `UIImageView` class instead of drawing the image.

`UIImagePickerController` and `UIImagePickerControllerDelegate`
: This class and protocol give your app a way to obtain images (photos) and movies supplied by the user. The class presents and manages user interfaces for choosing and taking photos and movies. When users pick a photo, it delivers the selected `UIImage` object to the delegate, which must implement the protocol methods.

In addition to these classes, UIKit declares functions that you can call to perform a variety of tasks with images:

- **Drawing into an image-backed graphics context.** The `UIGraphicsBeginImageContext` function creates an offscreen bitmap graphics context. You can draw in this graphics context and then extract a `UIImage` object from it. (See "Drawing Images" (page 40) for additional information.)
- **Getting or caching image data.** Each `UIImage` object has a backing Core Graphics image object (`CGImageRef`) that you can access directly. You can then pass the Core Graphics object to the Image I/O framework to save the data. You can also convert the image data in a `UIImage` object to either a PNG or JPEG format by calling the `UIImagePNGRepresentation` or `UIImageJPEGRepresentation` functions. You can then access the bytes in the data object and you can write the image data to a file.
- **Writing an image to the Photo Album on a device.** Call the `UIImageWriteToSavedPhotosAlbum` function, passing in a `UIImage` object, to put that image in the Photo Album on a device.

"Drawing Images" (page 40) identifies scenarios when you would use these UIKit classes and functions.

Other Image-Related Frameworks

You can use several system frameworks other than UIKit to create, access, modify, and write images. If you find that you cannot accomplish a certain image-related task using a UIKit method or function, a function of one of these lower-level frameworks might be able do what you want. Some of these functions might require a Core Graphics image object (`CGImageRef`). You can access the `CGImageRef` object backing a `UIImage` object through the `CGImage` property.

> **Note:** If a UIKit method or function exists to accomplish a given image-related task, you should use it instead of any corresponding lower-level function.

The Core Graphics framework of Quartz is the most important of the lower-level system frameworks. Several of its functions correspond to UIKit functions and methods; for example, some Core Graphics functions allow you to create and draw to bitmap graphics contexts, while others let you create images from various sources. However, Core Graphics offers more options for handling images. With Core Graphics you can create and apply image masks, create images from portions of existing images, apply color spaces, and access a number of additional image attributes, including bytes per row, bits per pixel, and rendering intent.

The Image I/O framework is closely related to Core Graphics. It allows an app to read and write most image file formats, including the standard web formats, high dynamic range images, and raw camera data. It features fast image encoding and decoding, image metadata, and image caching.

Assets Library is a framework that allows an app to access assets managed by the Photos app. You can get an asset either by representation (for example, PNG or JPEG) or URL. From the representation or URL you can obtain a Core Graphics image object or the raw image data. The framework also lets you write images to the Saved Photos Album.

Supported Image Formats

Table C-2 lists the image formats supported directly by iOS. Of these formats, the PNG format is the one most recommended for use in your apps. Generally, the image formats that UIKit supports are the same formats supported by the Image I/O framework.

Table C-2 Supported image formats

Format	Filename extensions
Portable Network Graphic (PNG)	`.png`
Tagged Image File Format (TIFF)	`.tiff` or `.tif`
Joint Photographic Experts Group (JPEG)	`.jpeg` or `.jpg`

Format	Filename extensions
Graphic Interchange Format (GIF)	`.gif`
Windows Bitmap Format (DIB)	`.bmp` or `.BMPf`
Windows Icon Format	`.ico`
Windows Cursor	`.cur`
XWindow bitmap	`.xbm`

Maintaining Image Quality

Providing high-quality images for your user interface should be a priority in your design. Images provide a reasonably efficient way to display complicated graphics and should be used wherever they are appropriate. When creating images for your app, keep the following guidelines in mind:

- **Use the PNG format for images.** The PNG format provides lossless image content, meaning that saving image data to a PNG format and then reading it back results in the exact same pixel values. PNG also has an optimized storage format designed for faster reading of the image data. It is the preferred image format for iOS.
- **Create images so that they do not need resizing.** If you plan to use an image at a particular size, be sure to create the corresponding image resource at that size. Do not create a larger image and scale it down to fit, because scaling requires additional CPU cycles and requires interpolation. If you need to present an image at variable sizes, include multiple versions of the image at different sizes and scale down from an image that is relatively close to the target size.
- **Remove alpha channels from opaque PNG files.** If every pixel of a PNG image is opaque, removing the alpha channel avoids the need to blend the layers containing that image. This simplifies compositing of the image considerably and improves drawing performance.

Document Revision History

This table describes the changes to *Drawing and Printing Guide for iOS*.

Date	Notes
2012-09-19	Revised wording throughout, fixed a number of technical errors, and updated the printing chapter for iOS 6.
2011-09-26	Made several minor changes.
2011-03-09	Made some minor changes.
2010-12-07	Added "Drawing to a Bitmap Graphics Context" to the Images chapter.
2010-11-15	First version of the document describing the concepts, APIs, and techniques for drawing and printing in iOS.

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