iPhone OS Programming Guide

2008-07-08

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

iPhone OS comprises the operating system and technologies that you use to run applications natively on iPhone and iPod touch devices. iPhone OS is a new platform but builds upon the knowledge and technology that went into the creation of Mac OS X. Because the needs of a mobile device are different than those of a Macintosh computer, iPhone OS was built with mobile users in mind. iPhone OS introduces new design concepts and technologies that provide a more intuitive user experience while also providing the performance and battery life users expect. iPhone OS also incorporates technologies, such as the Multi-Touch interface, that are simply not present on desktop computers.

The iPhone SDK contains the code, information, and tools you need to develop, test, run, debug, and tune applications for the iPhone OS platform. The Xcode tools have been updated to support development for the iPhone OS platform. In addition to providing the basic editing, compilation, and debugging environment for your code, Xcode also provides the launching point for testing your applications on an iPhone or iPod touch device. It also lets you run applications in the iPhone simulator, which mimics the basic iPhone OS environment on your local Macintosh computer.

Who Should Read This Document?

This document is targeted at developers who are new to the iPhone OS platform and who want to understand the available technologies and how to use those technologies to build applications. Although many of the technologies described in this document are also present in Mac OS X, this document does not assume any familiarity with Mac OS X or its technologies.

iPhone Programming Guide is an essential guide for anyone looking to develop software for iPhone and iPod touch devices. It provides an overview of the technologies and tools that have an impact on the development process and provides you with important technical information about how to build applications for the platform. You should use this document to do the following:

- Orient yourself to the iPhone OS platform
- Learn about iPhone software technologies, why you might want to use them, and when.
- Learn how to create applications and run them in iPhone OS.
- Get tips and guidance on the best ways to design and implement your applications.

Organization of This Document

This document has the following chapters and appendix.

- "iPhone OS Overview" (page 17) provides a conceptual overview of iPhone OS and how you create applications.
- "iPhone OS Technologies" (page 33) provides an overview of the technologies in iPhone OS and where you can go to get more information about them.
- "Development Environment" (page 41) provides an overview of Xcode tools and how you use them to develop applications for iPhone OS.
- "Application Design Guidelines" (page 69) provides guidance to help you design applications that will run efficiently and be easy for users to understand.
- "The Application Environment" (page 87) contains information about how to use and configure the `UIApplication` object, which is at the heart of every iPhone application.
- "Windows and Views" (page 109) describes the iPhone windowing model and shows you how you use views to organize your user interface.
- "Event Handling" (page 139) describes the iPhone event model and shows you how to handle Multi-Touch events.
- "Graphics and Drawing" (page 151) describes the graphics architecture of iPhone OS and shows you how to draw shapes and images and incorporate animations into your content.
- "Audio and Video Technologies" (page 173) shows you how to use the audio and video technologies available in iPhone OS.
- "Device Features" (page 185) shows you how to integrate features such as location tracking, the accelerometers, and the built-in camera into your application.

- "Application Preferences" (page 193) shows you how to implement the interface for your application preferences.
- "Apple Applications URL Schemes" (page 207) provides information about the system-supported URL schemes that are used to launch other applications.

Providing Feedback

If you have feedback about the documentation, you can provide it using the built-in feedback form at the bottom of every page.

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the ADC website, which is at the following URL:

http://developer.apple.com/bugreporter/

You must have a valid ADC login name and password to file bugs. You can obtain a login name for free by following the instructions found on the Bug Reporting page.

See Also

The following documents provide additional information related to iPhone development:

- *UIKit Framework Reference* provides reference information for the classes discussed in this document.
- *Cocoa Fundamentals Guide* provides information on the design patterns and practices used by iPhone applications.
- *View Controller Programming Guide for iPhone OS* provides information on the use of view controllers in creating interfaces for iPhone applications.
- *iPhone Human Interface Guidelines* provides information about how to design the user interface of an iPhone application.
- *The Objective-C 2.0 Programming Language* introduces Objective-C and the Objective-C runtime system, which is the basis of much of the dynamic behavior and extensibility of iPhone OS.

iPhone OS Overview

iPhone OS is the platform used to develop applications for iPhone and iPod touch devices. iPhone OS complements the existing support for web applications by providing an environment for running applications natively on a device. The applications you create live side by side with the system applications, such as the Phone, iPod, Stocks, and Weather applications and are built using most of the same frameworks. Native applications provide a natural performance advantage over web applications that can be used to create more advanced applications such as games. They also let you interact with features that might not be available in a web-based application, such as the accelerometers.

If you are an existing Mac OS X developer, some aspects of iPhone OS should seem very familiar. Many of the same underlying technologies used to build Mac OS X applications are also used to build iPhone applications. iPhone OS leverages the maturity of the Mac OS X architecture and framework stack to provide a reliable and already tested platform for developing applications. Even the newer frameworks that do not have direct analogs in Mac OS X still borrow from the basic structure and design methodologies that have already been proven on that platform.

In addition to the platform technologies, the development environment in Mac OS X is based on another proven technology: the Xcode development tools. Xcode makes it easy to develop and test your iPhone applications locally, on your Mac OS X development system, and on a device. When you build an application, you tell Xcode whether you want to build and run that application and run it on a device or in the iPhone simulator. The simulator mimics most of the iPhone OS environment, providing a way for you to get your application up and running quickly. Of course, some features require you to do your testing on a device, but going back and forth between the simulator and device is easy.

All of the tools and technologies you need to develop applications for iPhone OS are included with the iPhone SDK.

iPhone OS Feature Summary

The following list is a high-level summary of some of the features available in iPhone OS. Because the technology stack in iPhone OS is rich, this list does not call out every feature, but instead tries to call out some of the more specialized features that you might be interested in as an application developer. For a specific list of frameworks and technologies available in iPhone OS, see "iPhone OS Technologies" (page 33).

- Infrastructure
  - High-level infrastructure for running your application's main processing loop

- Support for interface abstractions such as windows and views
- Support for handling Multi-Touch events
- Support for security features such as encryption, certificate management, and trust policies
- Support for internationalizing your software
- Support for communicating between applications using URL schemes
- Access to low-level features such as threads, ports, and standard I/O
- Support for basic data types such as collections

- Media Handling
  - Support for rendering 2D and 3D graphics
  - Support for managing, displaying, and creating images
  - Support for playing back and recording audio
  - Support for playing back full-screen video content
  - Support for game development
  - Support for animating your user interface
  - Support for displaying web-based content from your application's interface

- Hardware Access
  - Access to the accelerometer data
  - Support for taking pictures with the camera, where available

- Networking
  - Support for BSD sockets and higher-level socket abstractions
  - Support for encrypted network connections
  - Support for resolving DNS hosts
  - Support for Bonjour services
  - Support for registering custom URL schemes

- Data Management
  - User interface elements for organizing and displaying data
  - Access to the user's contact information
  - Access to the user's photo library
  - Access to the user's current location information
  - Support for SQLite databases
  - Support for XML parsing
  - Support for application preferences

Development Tools

Xcode, Interface Builder, and Instruments are the main tools you use to manage your iPhone source files, build them, and run them. **Xcode** is an integrated development environment (IDE) that provides the main work environment for your project. You use this program daily for writing, compiling, running, and debugging your code. It has all of the features you would expect in an advanced code development application and many more. Among the main features are the following:

- A project management system for defining software products
- A code editing environment that includes features such as syntax coloring, code completion, and symbol indexing
- An advanced documentation window for viewing and searching iPhone OS documentation
- An advanced build system with dependency checking and build rule evaluation
- Integrated source-level debugging using GDB
- Support for running iPhone applications in a simulator or on a device

Interface Builder is a tool you use in conjunction with Xcode to reduce the amount of time it takes to build and test your application's user interface. Interface Builder is a graphical application that you use to assemble your application's interface visually. From the Interface Builder editing environment, you drag and drop standard system components (such as views and controls) into a window, arrange them, and configure their attributes, all in a matter of seconds. The resulting interface reflects both the visual appearance and the object relationships that will exist in your application at runtime. All of this information is then saved in a special resource file that you load in your application when you need that particular interface.

Instruments is an advanced debugging and performance analysis application that you use to gather information about the runtime behavior of your application once it is built. Instruments lets you run your application in the simulator or on a device and track the amount of memory you are using, look for leaks, and find out other information about your application's activity. You can also compare the data you gather from several different runs to track your application's improvement.

For more information about Xcode and the other tools you use to build and run your projects, see "Development Environment" (page 41).

About iPhone Development

If you typically develop applications for desktop operating systems, such as Mac OS X or Windows, you may find developing for iPhone OS requires you to rethink your overall design. Many of the features you might expect to find in a desktop operating system may simply be irrelevant or impractical in iPhone OS. For example, the text-system facilities in iPhone OS are geared toward the needs of mobile users, who typically write email or take notes, not generate long reports. The lack of some features typically present in desktop applications is intentional and has been done to optimize the experience of working on a mobile device.

More important than the features that are not present are the ones that are. Part of the iPhone user experience is the ability to interact with iPhone and iPod touch devices in ways that you cannot interact with desktop applications. The Multi-Touch interface is a revolutionary new way to receive

events, reporting on each separate finger that touches the screen and making it possible to handle multifinger gestures and other complex input easily. Built-in hardware features such as the accelerometers, although present in some desktop systems, are used more extensively in iPhone OS to track the screen's current orientation and adjust your content accordingly.

The following sections provide you with a high-level introduction to some of the fundamental design practices used in iPhone OS. This list is not exhaustive but touches on the practices that may alter the way you approach your application's design. And although some of these practices will be familiar to existing Mac OS X programmers, many will also be new.

Application Styles

Much of the success of iPhone and iPod touch lies in the user experience. The integration of the hardware with the system software and built-in system applications provides a single, coherent experience for the user that focuses on the user's needs in the moment. While on the go, a user does not want to spend time digging through screen after screen of data looking for information. The user needs to find the required information quickly and move on to the next task.

The first step in designing your own applications is to decide what style of application you plan to provide. The application style is the key to determining which types of views and controls you should use in your interface and how information should be organized. iPhone OS defines three basic styles for applications:

- Productivity style
- Utility style
- Immersive style

The following sections provide an overview of these styles and the types of content for which they are suited. For more information about picking the actual controls to use in your interface, see *iPhone Human Interface Guidelines*.

Note: Many application styles can also be implemented using HTML and JavaScript and delivered via the web. **Web-based applications** can often perform many of the same tasks typically performed by native applications, even though they may not have access to the exact same set of features. Because they are often simpler and faster to create, web-based applications should be considered as an option for development over native applications.

You can create such applications using Dashcode or using your favorite HTML development tools. For information and guidance about how to create web-based applications, see *Safari Web Content Guide for iPhone OS*. For information about using Dashcode to create your web-based applications, see *Dashcode User Guide*.

Productivity Applications

In a **productivity application**, the focus is on the organization and manipulation of detailed information. Information in productivity applications is usually text-based, but may also be image based if the content is more visually oriented. To facilitate the navigation of the information, productivity applications tend to use multiple screens and make use of system controls to handle the navigation from screen to screen. The Settings application (Figure 1-1) is an example of a productivity application that uses a hierarchical navigation model to navigate from general to specific preferences.

Figure 1-1 A productivity application

Because the focus in productivity applications is on data and organization, productivity applications typically rely on system views and controls for their presentation and do little or no custom drawing. Table views are commonly used in productivity applications, as are text fields, labels, and other data-oriented views. Productivity applications are also the main clients of navigation bars and toolbars, which provide the behavior for the primary navigational models.

Utility Applications

Utility applications perform a targeted task that requires relatively little user input. Users open a utility application to see a quick summary of information or to perform a simple task on a small number of objects. The interface for a utility application should be a visually appealing and uncluttered to make it easier to spot the needed information quickly. The Weather application (Figure 1-2) is an example of a utility application, as is the Stocks application.

Figure 1-2 A utility application

The key to designing a utility application is the focused use of appropriate graphics. And although utility applications should have a pleasing visual appearance, their actual drawing needs should be minimal. As a result, utility applications can use any of the iPhone drawing technologies (Quartz, Core Animation, or OpenGL ES) to draw their content.

Immersive Applications

Immersive applications offer a full-screen, visually rich environment that's focused on the content and the user's experience with that content. This style of application is most commonly used for implementing games and multimedia-centric applications. Because they are graphics oriented, immersive applications often present custom interfaces, relying less on standard system views and controls. They also hide the status bar, as shown in Figure 1-3.

Figure 1-3 A full-screen media player application

Games and media rich applications that require frequent screen updates typically use OpenGL ES to draw content, because it provides good performance for full-screen content at high frame rates. Many game developers are also familiar with OpenGL for desktop operating systems, so OpenGL ES is often a good match for those types of developers looking to create games in iPhone OS.

Application Basics

If you decide that building a native iPhone application is the right approach for your project, you should understand the basic concepts that underlie iPhone development. Writing a native iPhone application is not like writing a Mac OS X application. Although there are similarities between the two processes, developing for iPhone OS requires a much tighter integration with the overall system. This integration permeates everything from your basic programming practices to the technologies you use. The following sections describe some of the ways in which your application interacts with iPhone OS and how you can organize your application to take advantage of that environment.

The Runtime Environment

The applications you create reside side-by-side with the system applications on the user's Home screen. When the user taps your application's icon, it launches and becomes the visible application, filling the entire screen with its content. Besides the kernel, the only other programs that run while your application is running are the background daemons needed to manage critical system behaviors. When the user presses the Home button or performs an action that would require the launching of a different application, the system quits your application.

Because most applications are run for very short periods of time, good performance is essential. Your application should launch and quit quickly, doing as little during those transition times as possible. Even while your application is running, you should always pay attention to its performance. Performance is critical if you want users to accept your application and use it regularly. If it takes too long to launch your application or perform key tasks, the user is going to be less inclined to use it. Moreover, your application's workflow and user interface need to be well laid out to make it easier for the user to find key information quickly.

Although your application typically runs only for short periods of time, you should still make it appear as though your application never quit. Whenever your application quits, you should save out any state information you would need to put your application in the same configuration the next time it launches. This sort of behavior provides consistency for the user and also reduces the amount of repetitive navigation needed each time your application launches.

For more information about performance and managing state information, see "Application Design Guidelines" (page 69).

Application Security

Security is an important concern for users, especially on a mobile device. Users store personal files and contact information on these devices and do not want that information to be used for malicious purposes. Despite the best efforts of application developers, however, hackers still manage to find ways to gain control of applications via the network and other means. For this reason, iPhone OS automatically runs applications in a protected, "sandboxed" environment.

The **sandbox environment** used by iPhone OS prevents applications from accessing resources outside of their own domain. For example, the sandbox prevents your application from modifying system files and the files of other applications. It also prevents your application from connecting to low-number network ports or any other resources that typically require root-level privileges. Thus, if an attacker were to compromise your code and attempt to modify any protected resources, the attempts would simply fail.

Even with the sandbox environment in place, you should not be lax in your own security planning. The presence of a sandbox does not protect your own code from direct attacks by malicious entities. If you accept input from the user, you should always validate it. If there is an exploitable buffer overflow in your input-handling code, an attacker might be able to crash your program or use it to execute the attacker's code. The sandbox limits the damage an attacker can cause, but does not prevent such attacks from happening.

Memory Management

iPhone OS is primarily an object-oriented system, so most of the memory you create is in the form of Objective-C objects. Objects in iPhone OS use a reference-counting scheme to know when it is safe to free up the memory occupied by the object. When you first create an object, it starts off with a reference count of 1. Clients receiving that object can opt to retain it, thereby incrementing its reference count by 1. When a client no longer needs an object, it releases it, thereby decrementing its reference count by 1. When an object's reference count equals 0, the system automatically reclaims the memory for the object.

Note: iPhone OS does not support memory management using the garbage collection feature that is in Mac OS X v10.5 and later.

If you want to allocate plain blocks of memory—that is, memory not associated with an object—you can do so using the standard Malloc library of calls. As is the case with any memory you allocate using `malloc`, you are responsible for releasing that memory when you are done with it by calling the `free` function. The system does not release Malloc-based blocks for you.

Regardless of how you allocate memory, managing your overall memory usage is more important in iPhone OS than it is on Mac OS X. Although iPhone OS has a virtual memory system, it does not use a swap file. This means that code pages can be flushed as needed but your application's data must all fit into memory at the same time. The system monitors the overall amount of free memory and does what it can to give your application the memory it needs. If memory usage becomes too critical though, the system may terminate your application. However, this option is only used as a last resort to ensure that the system has enough memory to perform critical operations such as receiving phone calls.

For more information about how to allocate objects in iPhone OS, see *Cocoa Fundamentals Guide*. For information and tips on how to improve your application's memory usage, see "Managing Your Memory Usage" (page 70).

Application Structure

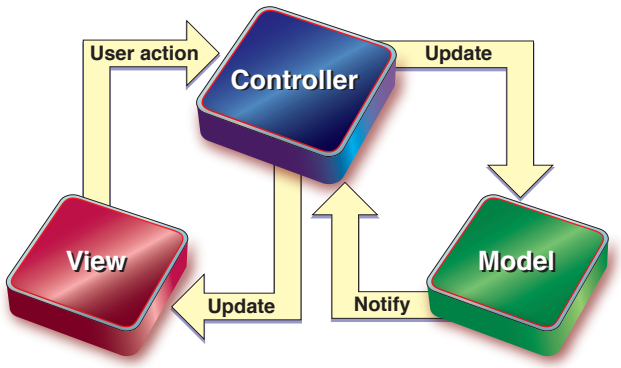
The structure of iPhone applications is based on the **Model-View-Controller (MVC)** design pattern because it benefits object-oriented programs in several ways. MVC–based programs tend to be more adaptable to changing requirements—in other words, they are more easily extensible than programs that do not use MVC. Furthermore, the objects in these programs tend to be more reusable and their interfaces tend to be better defined.

In the MVC design pattern, the **model layer** consists of objects that represent the data your application manages. The objects in this layer should be organized in the way that makes the most sense for the data. External interactions with model objects occur through a well-defined set of interfaces, whose job is to ensure the integrity of the underlying data at all times.

The **view layer** defines the presentation format and appearance of the application. This layer consists of your application's windows, views, and controls. The views can be standard system views or custom views you create. You configure these views to display the data from your model objects in an appropriate way. In addition, your view objects need to generate notifications in response to events and user interactions with that data.

The **controller layer** acts as the bridge between the model and view layers. It receives the notifications generated by the view layer and uses them to make the corresponding changes in the data model. Similarly, if the data in the data layer changes for other reasons (perhaps because of some internal computation loop), it notifies an appropriate controller object, which then updates the views. Figure 1-4 shows the basic model-view-controller relationships.

Figure 1-4 The model-view-controller design pattern



The Multi-Touch Interface

The **Multi-Touch interface** in iPhone OS makes it possible for your application to recognize and respond to distinct events generated by multiple fingers touching the device. The ability to respond to multiple fingers offers considerable power but represents a significant departure from the way traditional mouse-based event-handling systems operate. Handling touch events may require you to rethink the way users should interact with your application's data.

As each finger touches the surface of the device, the touch sensor generates a new touch event. While the finger remains in contact with the device, additional touch events are generated to indicate the finger's new position. When the finger loses contact with the device surface, the system delivers a touch ended event.

While one finger is down and moving around the surface of the device, it is perfectly reasonable for the user to place another finger on the device and begin moving it. When this happens, the event system generates another new touch event and delivers it with the original touch event in a single event object. Each touch event is distinct and continues tracking one of the fingers, relaying information about that finger's location and tracking state. Because the events arrive together, you can correlate them to one another and use them to identify trends. For example, if the events indicate the user is performing a pinch-close or pinch-open gesture (as shown in Figure 1-5) and the underlying view supports magnification, you could use those events to change the current zoom level.

Figure 1-5 Using touch events to detect gestures

For information about handling touch events in your code, see "Event Handling" (page 139).

Windows and Drawing

iPhone applications use high-quality graphics, instead of large amounts of text, to create a more pleasant user experience. The presentation of your application's content is an important part of distinguishing your application and making it fun to use. It can also be a way to convey useful information to the user. The following sections describe several key aspects of the visual portion of your application's user interface that you need to take into account as you proceed with your design.

The Full-Screen Window

Because only one application is visible at a time, every iPhone application is a full-screen application. Windows in iPhone OS are not user-resizable, cannot be closed by the user, and do not have a title bar. Instead, the window simply provides the background on which you present your application's content. As a result, your application needs to create only one window that spans the entire screen. You use that window to present all of your application's contents, including information that appears to span multiple screens. When you want to present a new screen of information, you use views, not windows, to slide that new information into place.

Although your window occupies the entire screen, there are still some options you need to consider before creating it:

- Does your application display the device's status bar?
- Which window orientations does your application support?

With the exception of immersive applications, most applications display a status bar. The presence of the status bar affects how you lay out the contents of your window. You can configure the status bar to be either totally opaque or partially transparent. For an opaque status bar, you would typically position your window's content view so that it is not obscured by the status bar. For a transparent status bar, however, your content view would fill the entire window.

When it comes to window orientations, applications can support portrait mode, landscape mode, or both. Supporting both orientations is not required and for most applications is not recommended. Orientation modes should reflect a way of presenting data that is unique to that orientation. For

example, the iPod application uses portrait mode to display songs and videos hierarchically but uses landscape mode to display them using Cover Flow. In your own applications, you should support both orientations only when doing so lets you enhance the user experience.

The Importance of Animation

Animation may seem like eye candy in many situations, but in iPhone OS it plays a very important role. Animation is used extensively to provide the user with contextual information and immediate feedback. For example, when the user navigates hierarchical data in a productivity style application, rather than just replace one screen with another, iPhone applications animate the movement of each new screen into place. The direction of movement indicates whether the user is moving up or down in the hierarchy and also provides a visual cue that there is new information to look at.

Because of its importance, support for animation is built-in to the low-level drawing infrastructure of iPhone OS. All windows and views are backed by Core Animation layers, which provide the basic drawing surface needed to create animations. In addition, many view-based properties are inherently animatable—meaning that changing the value of the property generates Core Animation commands to animate the change. All an application has to do to run these animations is specify the start and end of the animation block and the rest is handled for you. The infrastructure also means that it is very easy to augment the built-in animations with explicit animations that you create.

For information about using the built-in view-based animations, see "Animating Views" (page 131). For more information about Core Animation, see "Applying Core Animation Effects" (page 170).

Update-Based Drawing

Views provide the infrastructure for rendering your application's custom content to the screen. Views use an on-demand update model for flushing changes to the screen. Your code can request updates to all or part of your view in response to user interactions or changes to your application's data. As part of its regular housekeeping chores, the application gathers any pending update requests and delivers them to the appropriate view, coalescing duplicate requests into a single request.

Using the standard system views and controls simplifies the work you have to do by eliminating the need for you to draw anything at all. If you have custom content you want to draw, however, you can also define custom view objects and use UIKit and Quartz. UIKit provides high-level interfaces for performing typical drawing operations such as displaying and animating text, images, views, and simple rectangles. Quartz provides a vector-based drawing environment that you can use to draw path-based shapes, gradients, patterns, text, images, and PDFs.

Developers that prefer OpenGL for their drawing must also use views but they use them only to provide the underlying drawing surface. The EAGL framework provides a bridge between the native views of UIKit and the drawing surface model used by OpenGL ES. OpenGL ES provides a low-level but powerful drawing interface that you can use to create custom 2D and 3D content such as the type found in games and other media-centric applications.

For information about the drawing technologies in iPhone OS and for tips on how to draw your custom content, see "Graphics and Drawing" (page 151).

Key Integration Features

There are many features associated with iPhone and iPod touch that users take for granted. Some of these features are hardware related, such as the automatic adjustment of views in response to a change in a device's orientation. Others are software related, like the fact that all iPhone applications share a single list of contacts. Because so many of these features are integral to the basic user experience, you should consider them during your initial design to see how they might fit into your application.

Accelerometers

The accelerometers in iPhone and iPod touch provide valuable input for the system and for your own custom applications. An accelerometer measures changes in velocity along a single linear axis. Both iPhone and iPod touch have 3 accelerometers to measure changes along each of the primary axes in three-dimensional space, which allows you to detect motion in any direction. In addition, because gravity applies a constant acceleration straight down, the accelerometers can be used to detect the current orientation of a device, as shown in Figure 1-6.

Figure 1-6 Measuring the force of gravity using the accelerometers

The system uses the accelerometers to monitor a device's current orientation and to notify applications when that orientation changes. Each application can decide whether or not it wants to change the orientation of its interface when these notifications occur. Some applications may look better in either portrait or landscape mode but not both. Applications like Safari and the iPod use orientation changes to adjust the way they present their content. For applications that support orientation changes, UIKit makes handling those changes easier by providing support for automatically adjusting the size and orientation of your window's content.

In addition to handling orientation changes, applications that want to access the accelerometer data directly can also do so using UIKit. Games and other types of applications can use the raw accelerometer data to detect motion and use it as input. The system delivers accelerometer updates at regular (and configurable) intervals, making it suitable for use by many types of applications.

For information about using view controllers to handle orientation changes, see *View Controller Programming Guide for iPhone OS*. For information about receiving accelerometer events directly, see "Accessing Accelerometer Events" (page 185).

Core Location

iPhone OS–based devices are meant for users on the go. Therefore the software you write should also take this fact into account. And because the Internet and web make it possible to do business anywhere, being able to tailor information for the user's current location can make for a compelling user experience. After all, why list coffee shops in New York for someone who is thirsty and currently in Los Angeles? That's where the Core Location framework can help.

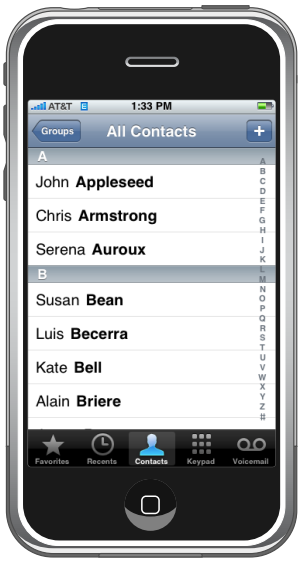
The Core Location framework monitors signals coming from cell phone towers and Wi-Fi hotspots and uses them to triangulate the user's current position. Using this framework, you can specify the desired accuracy of the location information along with the threshold for reporting changes in the current location. The framework generates an initial position fix for you but thereafter reports updates only when the desired threshold is exceeded or when a more accurate position fix becomes available. Of course, receiving continuous updates does require the use of the device's onboard radios, which if left on for too long can drain the user's battery. Therefore, you should always use this framework judiciously to get the information you need but then turn off location updates when you do not need them.

For an example showing you how to get location data in your application, see "Getting the User's Current Location" (page 188).

Contacts

The user's list of contacts is an important resource that all system applications share. The Phone, Mail, and SMS Text applications use it to identify people the user needs to contact and to facilitate basic interactions such as starting a phone call, email, or text message. Your own applications can access this list of contacts for similar purposes or to get other information relevant to your application's needs.

Figure 1-7 Accessing the user's contacts



iPhone OS provides both direct access to the user's contacts and indirect access through a set of standard picker interfaces. Using the direct access, you can obtain the contact information directly from the contacts database. You might use this information in cases where you want to present contact

information in a different way or filter it based on application-specific criteria. In cases where you do not need custom interface, however, iPhone OS also provides the set of standard system interfaces for picking and creating contacts. Incorporating these interfaces into your applications requires little effort but makes your application look and feel like it's part of the system.

You access the user's contact information using the Address Book and Address Book UI frameworks. For more information about these frameworks, see *Address Book Framework Reference* and *Address Book UI Framework Reference*.

The Camera and Photo Library

The Camera application on iPhone lets users take new pictures and store them in a centralized photo library along with the other pictures they upload from their computer. And although the iPod touch has no camera, it does have a photo library to hold the user's uploaded pictures. iPhone OS provides access to both of these features through classes in the UIKit framework.

Figure 1-8 Accessing the built-in camera

Through the camera and photo library support in UIKit, you can incorporate system-provided picker interfaces into your application. These interfaces provide standard system views for selecting a photo from the user's photo library or taking a picture using the camera. When the user dismisses the picker interface, it returns the selected image back to your application. Because it is the standard picker interface, it has all the same user controls available in the Camera or Photos applications, including the ability to edit and crop the selected image. Thus, the behavior of your application stays consistent with other system-supplied applications.

For information on how to use the picker interface to take pictures and access the user's photos, see "Taking Pictures with the Camera" (page 190) and "Picking a Photo from the Photo Library" (page 192).

Before You Go Any Further

The iPhone OS platform use the Objective-C language as the primary language for application development. Objective-C is an object-oriented language that is a superset of the C language. Although most of its basic syntax is the same as C, the syntax for working with objects may be new to many C and C++ developers. Over time, this has also lead to the use of many design patterns that simplify application development but which may be new to some developers. Understanding these patterns will make it easier for you to create applications that work with the system frameworks rather than fight them.

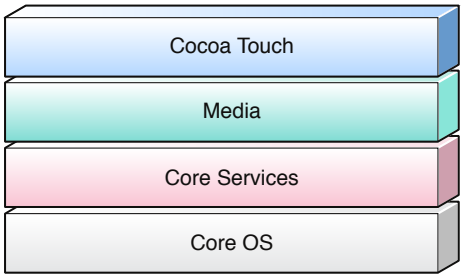
Although you use Objective-C primarily for your interface-related code, you may continue to use other programming languages for developing other parts of your application. Because it is based on C, the Objective-C language supports integration with both C and C++ code. You may also use any libraries that are supported by the Xcode tools and can be linked to your Objective-C code modules.

Information about both the Objective-C language and design patterns can be found in *Cocoa Fundamentals Guide*. This one book is a great primer that you can use to get up to speed quickly. After reading this document, you should have enough information to start exploring the iPhone OS frameworks and the rest of this document. Of course, if you want additional information about the Objective-C language, and how you can mix C, C++, and Objective-C source files, read *The Objective-C 2.0 Programming Language*.

iPhone OS Technologies

The implementation of iPhone OS can be viewed as a set of layers, which are shown in Figure 2-1. At the lower layers of the system are the fundamental services on which all applications rely, while higher-level layers contain more sophisticated services and technologies.

Figure 2-1 Layers of iPhone OS



As you write your code, you should prefer the use of higher-level frameworks over lower-level frameworks whenever possible. The higher-level frameworks are there to provide object-oriented abstractions for lower-level constructs. These abstractions generally make it much easier to write code because they reduce the number of lines of code you have to write and encapsulate potentially complex features, such as sockets and threads. Although they abstract out lower-level technologies, they do not mask those technologies from you. The lower-level frameworks are still available for developers who prefer using them or who want to use aspects of those frameworks that are not exposed at the higher level.

The following sections provide more detail about what is in each of the exposed layers of iPhone OS, starting with the topmost layers and working downward.

Cocoa Touch

The **Cocoa Touch** layer is one of the most important layers in iPhone OS. It comprises the UIKit and Foundation frameworks (`UIKit.framework` and `Foundation.framework`), which provide the basic tools and infrastructure you need to implement graphical, event-driven applications in iPhone OS. It also includes several other frameworks that provide key services for accessing device features, such as the user's contacts.

The **UIKit** framework (`UIKit.framework`) is an Objective-C framework that provides the key infrastructure for implementing graphical, event-driven applications in iPhone OS. Every application in iPhone OS uses this framework to implement this core set of features:

- Application management
- Graphics and windowing support
- Event-handling support
- User interface management
- Objects representing the standard system views and controls
- Support for text and web content

In addition to providing the fundamental code for building your application, UIKit also incorporates support for some device-specific features, such as the following:

- Accelerometer data
- The built-in camera (where present)
- The user's photo library
- Device-specific information

For information about the classes of the Foundation and UIKit frameworks, see *Foundation Framework Reference* and *UIKit Framework Reference*.

Media

The graphics and media technologies in iPhone OS are geared toward creating the best multimedia experience available on a mobile device. More importantly, these technologies were designed to make it easy for you to build good-looking and -sounding applications quickly. The high-level frameworks in iPhone OS make it easy to create advanced graphics and animations quickly, while the low-level frameworks provide you with access to the tools you need to do things exactly the way you want.

Graphics Technologies

High-quality graphics are an important part of all iPhone applications, and iPhone OS provides several key technologies to do your 2D and 3D drawing. Table 2-1 lists the technologies available for you to use in iPhone OS.

Table 2-1 2D and 3D graphics technologies

Framework	Services
`OpenGLES.framework`	The **OpenGL ES** framework (`OpenGLES.framework`) is based on the OpenGL ES v1.1 specification and provides tools for drawing 2D and 3D content. It is a C-based framework that works closely with the device hardware to provide high frame rates for full screen game-style applications. You always use this framework in conjunction with the EAGL framework. For details about the available OpenGL ES support, see "Drawing with OpenGL ES" (page 162).
`EAGL.framework`	The **EAGL** framework provides the interface between your OpenGL ES drawing code and the native window objects of your application.
`QuartzCore.framework`	**Core Animation** (`QuartzCore.framework`) is an advanced animation and compositing technology that uses an optimized rendering path to implement complex animations and visual effects. It provides a high-level, Objective-C interface for configuring animations and effects that are then rendered in hardware for performance. Core Animation is integrated into many parts of iPhone OS, including UIKit classes such as `UIView`, providing animations for many standard system behaviors. You can also use the Objective-C interface in this framework to create custom animations.
`CoreGraphics.framework`	**Quartz** (`CoreGraphics.framework`) is the same advanced, vector-based drawing engine that is used in Mac OS X for drawing. It provides support for path-based drawing, anti-aliased rendering, gradients, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. Although the API is C-based, it uses object-based abstractions to represent fundamental drawing objects, making it easy to store and reuse your graphics content.

For more information about the available graphics technologies, including examples of how to use them, see "Graphics and Drawing" (page 151).

Core Audio

Native support for audio is provided by the Core Audio family of frameworks, listed in Table 2-2. **Core Audio** is a low-latency C-based interface that supports the manipulation of multichannel audio. You can use Core Audio in iPhone OS to generate, record, mix, and play audio in your applications. You can also use Core Audio to access the vibrate capability on devices that support it.

Table 2-2 Core Audio frameworks

Framework	Services
`CoreAudio.framework`	Provides audio type and file format information.

Framework	Services
AudioToolbox.framework	Provides playback and recording services for audio files and streams. This framework also provides support for managing audio files and playing system alert sounds.
AudioUnit.framework	Provides services for using audio units, which are audio processing modules.

For information about using Core Audio to play and record audio, see "Using Sound in iPhone OS" (page 173).

OpenAL

In addition to Core Audio, iPhone OS includes support for the **Open Audio Library (OpenAL)**. The OpenAL interface is a cross-platform standard for delivering 3D audio in applications. You can use it to implement high-performance positional audio in games and other programs that require high-quality audio output. Because OpenAL is a cross-platform standard, the code modules you write using OpenAL in iPhone OS can be ported to run on many other platforms.

For information about OpenAL, including how to use it, see http://www.openal.org.

Video Technologies

iPhone OS provides support for full-screen video playback through the **Media Player** framework (MediaPlayer.framework). This framework supports the playback of movie files with the .mov, .mp4, .m4v, and .3gp filename extensions and using the following compression standards:

- H.264 Baseline Profile Level 3.0 video, up to 640 x 480 at 30 fps. Note that B frames are not supported in the Baseline profile.
- MPEG-4 Part 2 video (Simple Profile)
- Numerous audio formats, including:
  - AAC
  - Apple Lossless (ALAC)
  - A-law
  - IMA/ADPCM (IMA4)
  - linear PCM
  - µ-law

For information on how to use this framework, see "Playing Video Files" (page 183).

Core Services

The **Core Services** layer provides the fundamental system services that all applications use. Even if you do not use these technologies directly, every other technology in the system is built on top of them.

Address Book

The Address Book framework (`AddressBook.framework`) provides programmatic access to the contacts stored on a user's device. Applications that need access to this information, such as email and chat programs, can use this framework to access the data stored in contact records directly. Those programs can use the information internally or provide a custom user interface for displaying that data. For information about the functions in this framework, see *Address Book Framework Reference*.

The Address Book UI framework (`AddressBookUI.framework`) complements the Address Book framework by providing a graphical interface for accessing the user's contacts. You use the Objective-C classes in this framework to present the system standard interfaces for picking existing contacts and creating new contacts. For information about the classes in this framework, see *Address Book UI Framework Reference*.

Core Foundation

The **Core Foundation** framework (`CoreFoundation.framework`) is a set of C-based interfaces that provide basic data management and service features for iPhone applications. This framework includes support for the following:

- Collection data types (arrays, sets, and so on)
- Bundle support
- String management
- Date and time management
- Raw data block management
- Preferences management
- URL and Stream manipulation
- Thread and run loop support
- Port and socket communication

The Core Foundation framework is closely related to the Foundation framework, which provides Objective-C interfaces for the same basic features. In situations where you need to mix Foundation objects and Core Foundation types, you can take advantage of the "toll-free bridging" that exists between the two frameworks. **Toll-free bridging** means that you can use some Core Foundation and Foundation types interchangeably in the methods and functions of either framework. This support is available for many of the data types, including the collection and string data types. The class and type descriptions for each framework list whether an object is toll-free bridged and, if so, what object it is bridged with.

For more information about this framework, see *Core Foundation Framework Reference*.

Core Location

The Core Location framework (`CoreLocation.framework`) lets you determine the current latitude and longitude of a device. The framework uses the available hardware to triangulate the user's position based on nearby signal information. The Maps application uses this feature to show the user's current position on a map, and you can incorporate this technology into your own applications to provide position-based information to the user. For example, a service that searched for nearby restaurants, shops, or facilities could base that search on the user's current location. For information about how to use this framework to get the user's location, see "Getting the User's Current Location" (page 188).

CFNetwork

The **CFNetwork** framework (`CFNetwork.framework`) is a high-performance, C-based framework that provides a set of object-oriented abstractions for working with network protocols. These abstractions give you detailed control over the protocol stack and make it easy to use lower-level constructs such as BSD sockets. You can use this framework to simplify tasks such as communicating with FTP and HTTP servers or resolving DNS hosts. Here are some of the tasks you can perform with the CFNetwork framework. You can:

- Use BSD sockets
- Create encrypted connections using SSL or TLS
- Resolve DNS hosts
- Work with HTTP, authenticating HTTP and HTTPS servers
- Work with FTP servers
- Publish, resolve, and browse Bonjour services

CFNetwork is based, both physically and theoretically, on BSD sockets. For information on how to use CFNetwork, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*.

Security

In addition to its built-in security features, iPhone OS also provides an explicit **Security** framework (`Security.framework`) that you can use to guarantee the security of the data your application manages. This framework provides interfaces for managing certificates, public and private keys, and trust policies. It supports the generation of cryptographically secure, pseudo-random numbers. It also supports the storage of certificates and cryptographic keys in the keychain, which is a secure repository for sensitive user data.

Note: OpenSSL is not supported for third party development, nor are the `libssl` or `libcrypto` libraries.

For information about the functions and features associated with the Security framework, see *Security Framework Reference*.

SQLite

The **SQLite library** lets you embed a lightweight SQL database into your application without running a separate remote database server process. From your application, you can create local database files and manage the tables and records in those files. The library is designed for general purpose use but is still optimized to provide fast access to database records.

The header file for accessing the SQLite library is located in `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS2.0.sdk/usr/include/sqlite3.h`, where <*Xcode*> is the path to your Xcode installation directory. For more information about using SQLite, go to http://www.sqlite.org.

XML Support

Support for manipulating XML content is provided by the `libXML2` and `libxslt` libraries. These are open source libraries that you can use to parse or write arbitrary XML data quickly and transform XML content to HTML.

The header files for accessing the `libXML2` library are located in `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS2.0.sdk/usr/include/libxml2/`, where <*Xcode*> is the path to your Xcode installation directory. The headers for the `libxslt` library are in `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS2.0.sdk/usr/include/libxslt/`. For more information about using `libXML2` and `libxslt`, go to http://xmlsoft.org/index.html.

Core OS

The **Core OS** layer encompasses the kernel environment, drivers, and basic interfaces of the operating system. The kernel itself is based on Mach and is responsible for every aspect of the operating system. It manages the virtual memory system, threads, file system, network, and inter-process communication. The drivers at this layer also provide the interface between the available hardware and the system frameworks that vend hardware features. Access to kernel and drivers is restricted to a limited set of system frameworks and applications.

iPhone OS provides a set of interfaces for accessing many low-level features of the operating system. Your application accesses these features through the `LibSystem` library. The interfaces are C-based and provide support for the following:

- Threading (POSIX threads)
- Networking (BSD sockets)
- File-system access
- Standard I/O
- Bonjour and DNS services
- Locale information
- Memory allocation

- Math computations

For information about the functions available in the LibSystem library, see *iPhone OS Manual Pages* in the iPhone Reference Library.

Development Environment

To develop compelling iPhone applications you use Xcode, Apple's premier integrated development environment (IDE).

In this chapter, you first see an overview of the development process. The rest of the chapter expands on this process, from using Xcode to develop your code, to testing your iPhone application using the iPhone Simulator or a device. You also learn how to use the Xcode tools to debug and to fine-tune your application's performance.

The Development Process

Xcode is a suite of software development tools used to create iPhone OS and Mac OS X applications. This suite includes applications, command-line tools, frameworks, and libraries you use to develop software products. The centerpiece of the suite is the Xcode application, which provides an elegant, powerful user interface for creating and managing software development projects. You use Xcode to organize and edit your source files, view documentation, build your application, debug your code, and optimize your application's performance.

The iPhone application development process is divided into these major steps:

1. Create your project.

   Xcode provides several predefined project templates that get you started. You choose the template that implements the type of application you want to develop.

2. Design the user interface.

   Interface Builder lets you design your application's user interface graphically and save those designs as resource files that you load into your program at runtime.

   If you do not wish to use Interface Builder, you may layout your user interface programmatically.

3. Write code.

   Xcode provides several features that help you to write code fast, including class and data modeling, code completion, direct access to documentation, refactoring, and more.

4. Build and run your application.

You build your application on your computer and run it on the iPhone Simulator or your device.

The iPhone Simulator implements the iPhone OS interfaces, providing an environment that closely resembles the environment devices provide. It allows you to run your applications in Mac OS X, providing a way to quickly test your application's functionality without the need for an iPhone OS–based device. However, running applications in the Simulator is not the same as running them in actual devices.

First, the Simulator uses Mac OS X versions of the low-level system frameworks instead of the versions that run on the devices. Secondly, there may be hardware-based functionality that's unavailable on the Simulator. But, in general, the Simulator is a great tool to perform initial testing of your applications.

To compile and debug your code, Xcode relies on open-source tools, such as GCC and GDB. Xcode also supports team-based development with source control systems, such as Subversion, CVS, and Perforce.

5. Create your application's default image.

   This is the image displayed while your application loads, after the user taps your application's icon. See "Capturing Screenshots" (page 61) for details.

6. Create your application's Preferences schema files.

   Preferences schema files define the interface the Settings application displays when a user of your application navigates to its root preference page. For more information about defining application preferences pages, see "Application Preferences" (page 193).

7. Measure and tune application performance.

   Instruments is a dynamic performance analysis tool that lets you peer into your code as it's running and gather important metrics about what it is doing. You can view and analyze the data Instruments collects in real time, or you can save that data and analyze it later. You can collect data about your application's use of the CPU, memory, the file system, and the network, among other resources.

   Shark is another tool that helps you find performance bottlenecks in your code. It produces profiles of hardware and software performance events and shows how your code works as a whole and its interaction with iPhone OS.

   With Instruments and Shark, you can find and eliminate performance bottlenecks in your code.

The rest of this chapter takes you through the development process.

Creating Your Project and Writing Code

These sections describe the process of creating an iPhone application project and writing code for it.

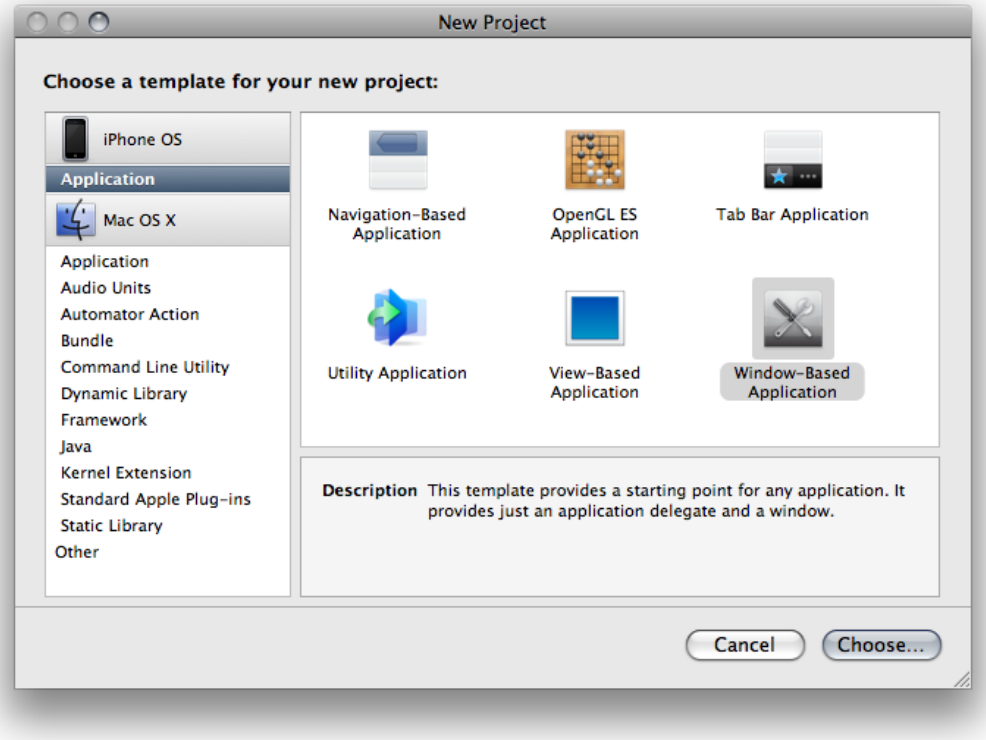
Beginnings

Xcode provides several iPhone application project templates to get you up and running developing your application. You can choose from these types of application:

- **Navigation-Based Application.** An application that uses a navigation controller.
- **OpenGL ES Application.** An application that uses an OpenGL ES–based view.
- **Tab Bar Application.** An application that uses a tab bar.
- **Utility Application.** An application that has a main view and a flipside view.
- **View-Based Application.** An application that uses a single view.
- **Window-Based Application.** A starting point for any application, containing an application delegate and a window.

To create a project, follow these steps:

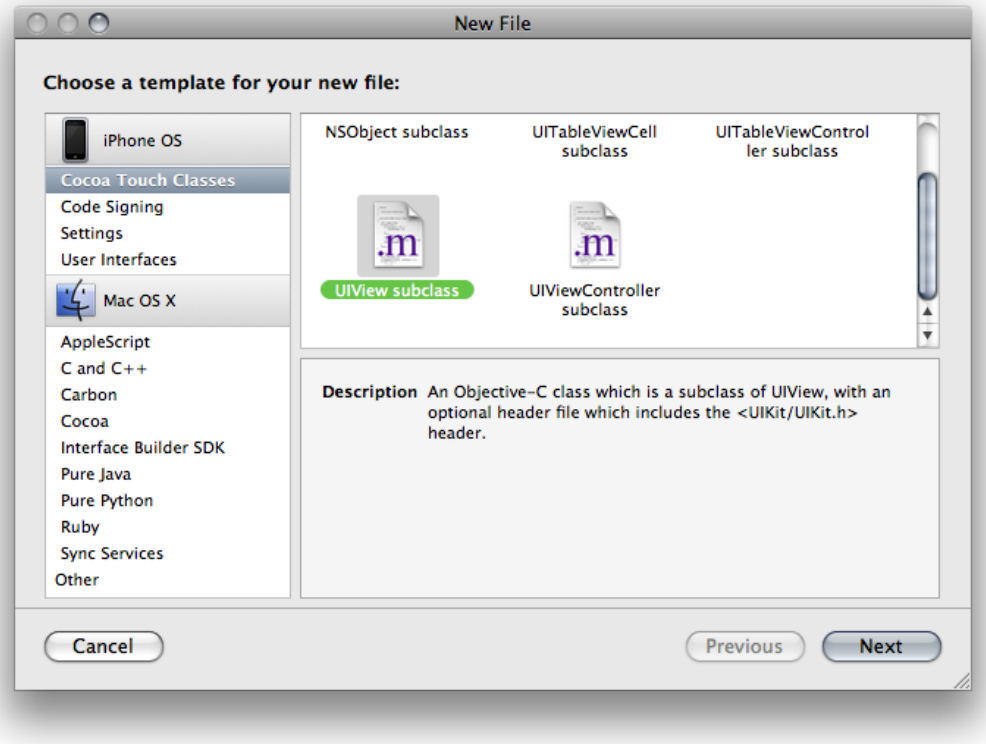
1. Launch Xcode.
2. Choose File > New Project.
3. Select the Window-Based Application template and click Choose.



4. Name your project and choose a location for it on your file system.
5. Add the `MyView` class to the project.

a. Choose File > New File.

b. Select the Cocoa Touch UIView subclass template and click Next.



c. In the File Name text field, enter `MyView.m`.

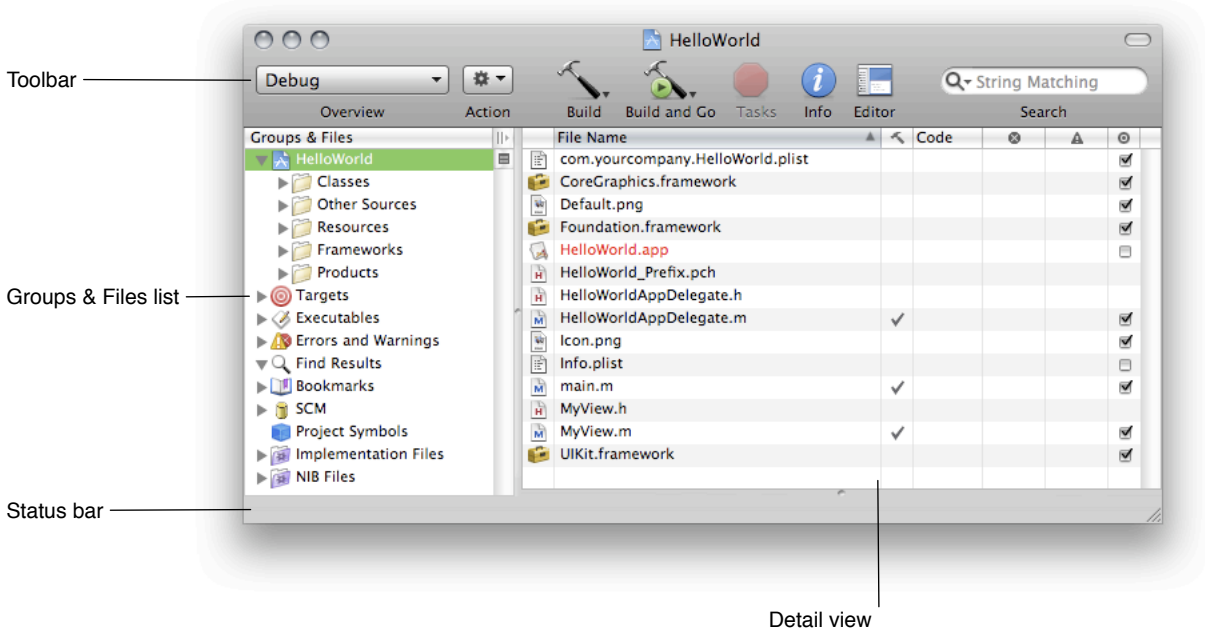
d. Select the “Also create "MyView.h"” option and click Finish.

6. Select the SDK to use to develop your application.

   The Active SDK setting specifies whether Xcode builds your application for a device or the iPhone Simulator. If you have a development device plugged in at the time you create the project, Xcode sets the Active SDK setting to build for your device. Otherwise, it sets it to build for the simulator.

To develop an iPhone application, you work on an Xcode project. And you do most of your work on projects through the **project window**, which displays and organizes your source files and other resources needed to build your application. It allows you to access and edit all the pieces of your project. Figure 3-1 shows the project window.

Figure 3-1 Project window

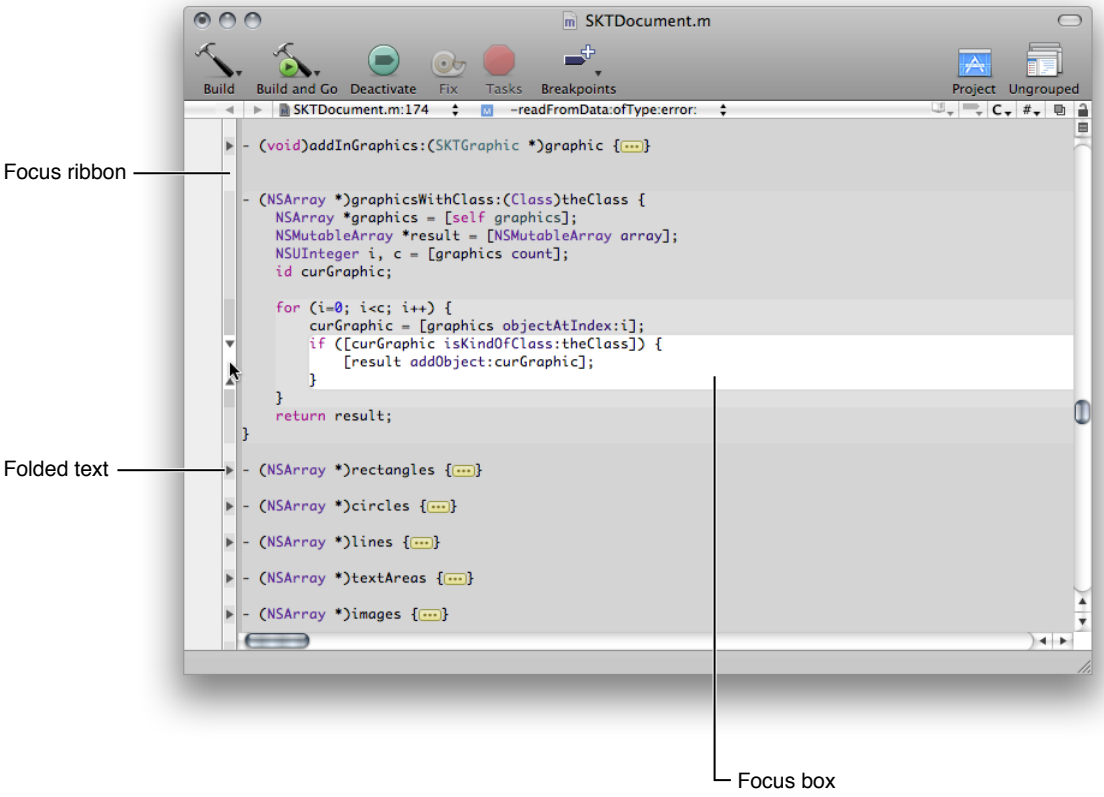


The project window contains the following key controls for navigating your project:

- **Groups & Files list.** Provides an outline view of your project contents. You can move files and folders around and organize your project contents in this list. The current selection in the Groups & Files list controls the contents displayed in the detail view.
- **Detail view.** Shows the item or items selected in the Groups & Files list. You can browse your project's contents in the detail view, search them using the Search field, or sort them according to column. The detail view helps you rapidly find and access your project's contents.
- **Toolbar.** Provides quick access to the most common Xcode commands.
- **Status bar.** Displays status messages for the project. During an operation—such as building or indexing—Xcode displays a progress indicator in the status bar to show the progress of the current task.
- **Favorites bar.** Lets you store and quickly return to commonly accessed locations in your project. The favorites bar is not displayed by default. To display the favorites bar use the View > Layout menu.

The main tool you use to write your code in Xcode is the Xcode text editor, shown in Figure 3-2. You can also edit files directly in the project window.

Figure 3-2 Text editor in a window



This is an advanced text editor that provides several convenient features:

- **Header-file lookup.** By Command–double-clicking a symbol, you can view the header file that declares the symbol.
- **API reference lookup.** By Option–double-clicking a symbol, you get access to API reference that provides information about the symbol's usage.
- **Code completion.** As you type code, you can have the editor help out by inserting text for you that completes the name of the symbol Xcode thinks you're going to enter. Xcode does this in an unobtrusive and overridable manner.
- **Code folding.** With code folding, you can collapse code that you're not working on and display only the code that requires your attention. Figure 3-2 illustrates how code folding can be used to focus on a particular section of code.

To learn more about creating projects, see *Xcode Project Management Guide*.

In Xcode, the text editor is where you spend most of your time. You can write code, build your application, and debug your code. Let's see how Xcode assists you in the first task.

First, modify the `HelloWorldAppDelegate` class to use the `MyView` class:

1. In the Groups & Files list, select the HelloWorld group.
2. In the detail view, double-click `HelloWorldAppDelegate.m`.

3. In the `HelloWorldAppDelegate` editor window:

    a. Add the following code line below the existing `#import` line.

    ```
    #import "MyView.h"
    ```

    b. Add the following code lines to the applicationDidFinishLaunching: method, bellow the override-point comment.

    ```
    MyView *view = [[MyView alloc] initWithFrame:[window frame]];
    [window addSubview:view];
    [view release];
    ```

After making these changes, the code in the `HelloWorldAppDelegate.m` file should look like this:

```
#import "HelloWorldAppDelegate.h"
#import "MyView.h"

@implementation HelloWorldAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

   // Override point for customization after app launch
   MyView *view = [[MyView alloc] initWithFrame:[window frame]];
   [window addSubview:view];
   [view release];

   [window makeKeyAndVisible];
}

- (void)dealloc {
    [window release];
    [super dealloc];
}

@end
```

Listing 3-1 shows code that draws text on your application's window. The following sections show how to add this code to your project using code completion and API reference lookup.

Listing 3-1 Method to draw "Hello, World!" on a view

```
- (void) drawRect:(CGRect) rect {
    /* Draw "Hello, World!" */
    NSString *hello   = @"Hello, World!";
    CGPoint  location = CGPointMake(10, 20);
    UIFont   *font    = [UIFont systemFontOfSize:24];
    [[UIColor whiteColor] set];
    [hello drawAtPoint:location withFont:font];
}
```

During development, you may need fast access to reference for a particular symbol or high-level documentation about API usage or an iPhone OS technology. Xcode gives you easy access to such resources through the Research Assistant and the Documentation window.

For more information about editing source code, see *Xcode Workspace Guide*.

Using Code Completion

This section shows how code completion can reduce the amount of typing needed to write your code. When **code completion** is active, Xcode uses both text you have typed and the context into which you have typed it to provide suggestions for completing the token it thinks you intend to type. For new Xcode users, code completion is not active by default.

To activate code completion:

1. Open the Xcode Preferences window.

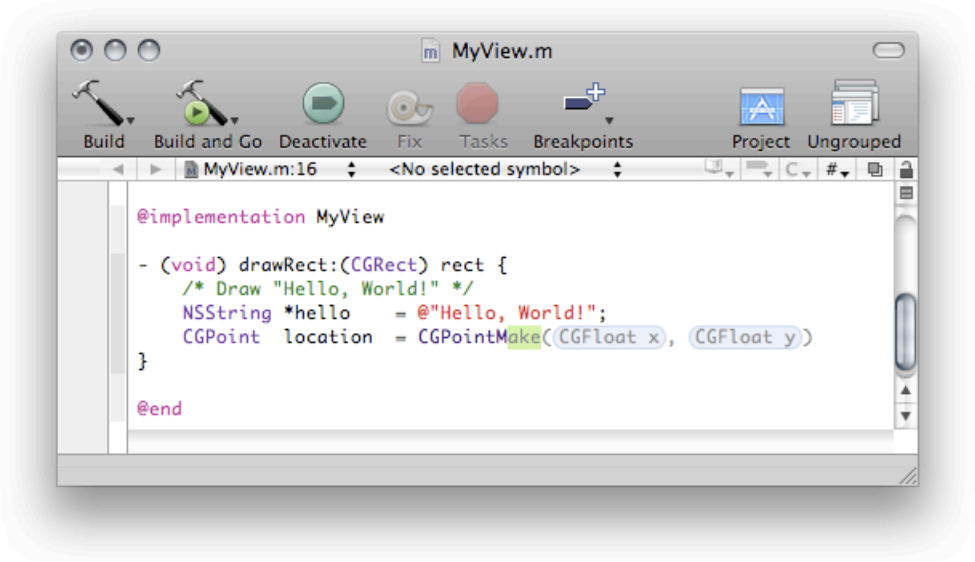
   Choose Xcode > Preferences (or press Command–,).

2. In the Code Sense pane and under the Code Completion section, choose Immediate from the Automatically Suggest pop-up menu.

3. Click OK.

In a Custom iPhone application project, you would add this code to the `MyView.m` file in your project. Double-clicking that file in the detail view opens a text editor window with the contents of that file. Figure 3-3 shows how that window may look as you add that code to the file.

Figure 3-3 Using code completion



Note how code completion works in the code line that starts with `CGPoint`. As you type the name of the `CGPointMake` function, Xcode recognizes that symbol and offers a suggestion. You can accept the highlighted part of the suggestion by pressing Tab. However, because Xcode has suggested the function needed in this case, you can enter the value for the parameters instead.

To jump to the first parameter, choose **Edit > Select Next Placeholder** (or press Control–/), and type `10`. The Select Next Placeholder command moves you among the arguments in function or method calls that Xcode suggests as completions to the text you're typing.

Repeat the command to jump to the second argument, and type `20`. As you can see, code completion can be an invaluable tool as you write code.

Make sure to enter the semicolon (`;`) at the end of the line and press Return.

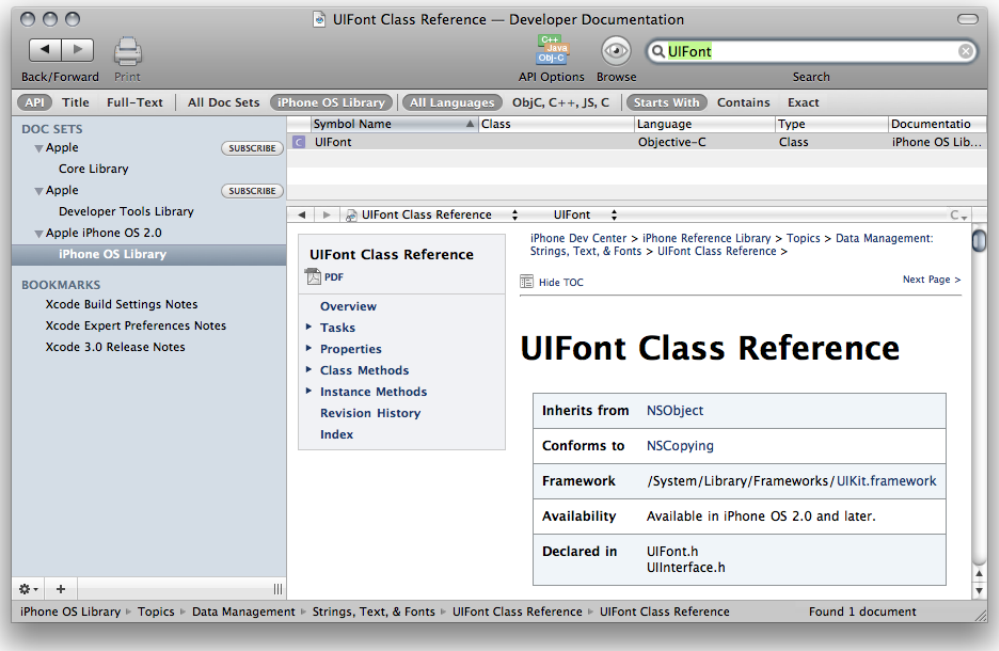
Using API Reference Lookup

As you write code and learn the iPhone OS API, you may need to consult API reference to find out how to use a class, function, or constant. The text editor provides direct access to API reference.

In `MyView.m`, enter `UIFont` below the line that starts with `CGPoint`. At this point you may not know much about the `UIFont` class.

To display the `UIFont` reference, select the class name and choose Help > Find Selected Text in API Reference (you can also Option–double-click the class name). This command searches for the selected symbol in the API reference for your project's SDK and displays it in the Documentation window, shown in Figure 3-4.

Figure 3-4 Viewing API reference in the Documentation window



In the Documentation window you can learn all aspects of the `UIFont` class, such as the tasks you can perform with it and how it relates and all the methods you can use on it or its instances.

While the Documentation window is a great tool, sometimes you may not want to take your focus away from the text editor while you write code, but need basic information about a symbol in a condensed way. The Research Assistant provides such information in a small and unobtrusive window as you write code.

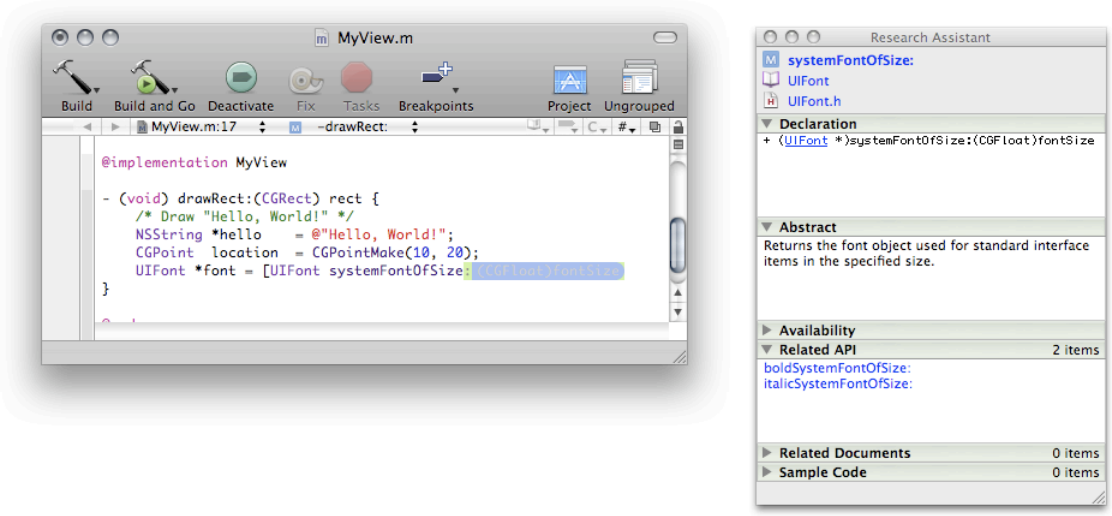
To display the Research Assistant, choose Help > Show Research Assistant.

Continue entering the line that starts with `UIFont`, by entering:

```
UIFont *font = [UIFont systemFontOfSize:24];
```

When the Research Assistant recognizes that you're typing the `systemFontOfSize:` method of the UIFont class, it displays its reference, as shown in Figure 3-5. All you have to do is glance at the Research Assistant to get essential details about the method.

Figure 3-5 Viewing API reference in the Research Assistant



From the Research Assistant you can quickly jump to more comprehensive reference for the method, the `UIFont` class, or even view the `UIFont.h` header file to view the method declaration.

Enter the last lines of the `drawRect:` instance method:

```
[[UIColor whiteColor] set];
[hello drawAtPoint:location withFont:font];
```

To learn more abut code completion, API reference look-up, or other features the text editor provides, see *Xcode Workspace Guide*.

Accessing Documentation

Documentation is an important resource of the software development process. As you develop iPhone applications in Xcode, you're likely to use documentation to learn about iPhone OS and its technologies, read about system frameworks, and look-up API reference. Xcode provides two ways to access documentation:

- The Research Assistant
- The Documentation window

The **Research Assistant** is a lightweight window, shown in Figure 3-5 (page 50), that provides a condensed view of the API reference for the selected item, without taking your focus away from the editor in which the item is located.

This window provides an unobtrusive way to consult API reference without using the Documentation window. However, when you need to delve deeper into the reference, the Documentation window is just a click away.

The **Documentation window** (Figure 3-6) lets you browse and search the developer documentation (which includes API reference, guides, and article collections about particular tools or technologies) installed on your computer. It provides access to a wider and more detailed view of the documentation than the Research Assistant, for the times when you need additional help.

Figure 3-6 The Documentation window

For more information about accessing documentation in Xcode, see *Xcode Workspace Guide*.

Setting Your Application's Icon

One of your application's resources is its icon, which appears in a user's home screen to identify your application.

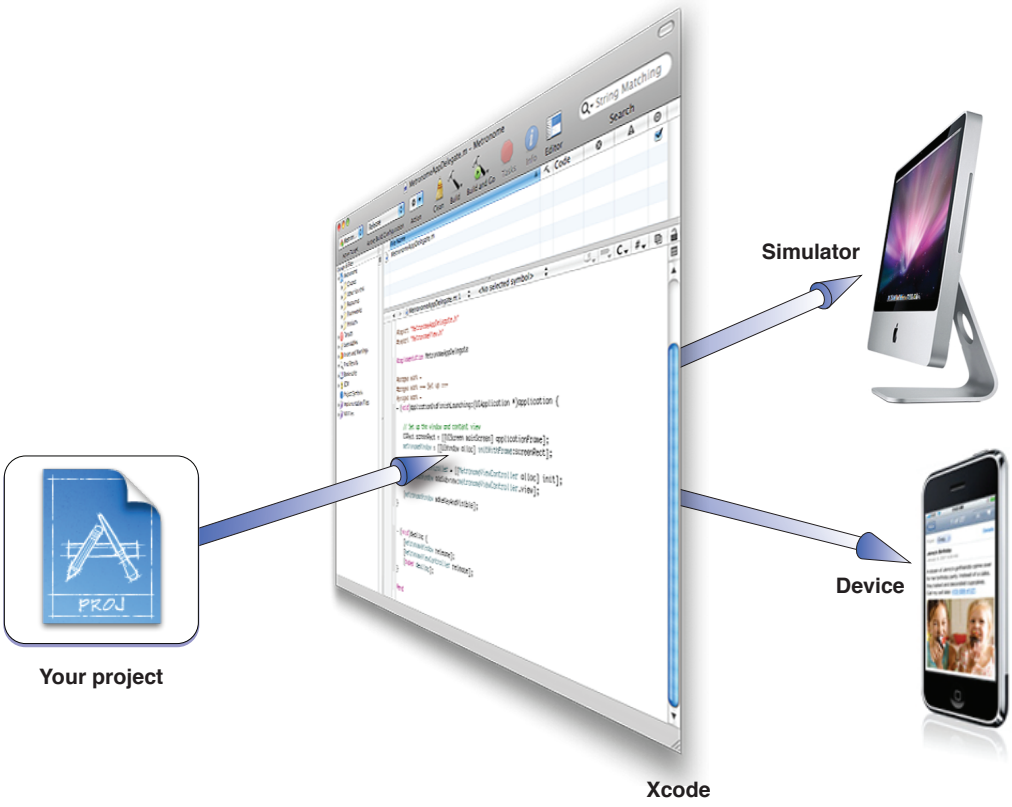
After adding your icon file to your project, you must set the `CFBundleIconFile` property in the `Info.plist` file to your icon's filename.

Building and Running Your Application

Building your application involves the following steps:

- Compiling your source files and generating your application binary.
- Placing the binary on the iPhone Simulator or your device.

Xcode performs these tasks for you when you execute the Build command.



To learn more about building applications, see *Xcode Project Management Guide*.

Working with the iPhone Simulator

The iPhone Simulator facilitates desktop-based iPhone application development. The iPhone Simulator provides a runtime environment that is very similar to the environment a device provides. You can use the iPhone Simulator to test your application just as it would run on a device, but with the convenience of your desktop; you have both your project and your application running on one screen. However, the environments are not identical; the simulator doesn't emulate devices.

Many of your application's performance aspects, such as CPU and network usage, can be measured effectively only on devices. But you can measure some aspects, such as your application's memory usage and object allocation, reliably on the Simulator because they are not tied to the runtime environment.

The following sections show how to use the iPhone Simulator SDK to build your application and run it on the iPhone Simulator.

Building Your Application for the iPhone Simulator

To build your application for the iPhone Simulator:

1. Select the iPhone Simulator SDK from the Project > Set Active SDK menu or the Overview toolbar item.
2. Choose Build > Build (or Build > Build and Run).

   The status bar in the project window indicates whether the build was successful or whether there are build errors or warnings. You can view build errors and warnings in the text editor or the project window.

Running Your Application on the Simulator

To run the application you built in "Building Your Application for the iPhone Simulator" (page 53), choose Run > Run.

Xcode launches the iPhone Simulator and starts your application.

Just like a device, you can go to the home screen on the iPhone Simulator and use the applications installed in it. Using its Hardware menu, you can rotate and lock the simulator. You can also use the Option key to pinch the simulator screen.

There may be times when you need to run code on the Simulator but not on a device, and the other way around. In those occasions, you can use the `TARGET_IPHONE_SIMULATOR` preprocessor macro, which evaluates to `true` when your application runs on the Simulator.

For example, change the `drawRect:` method in `MyView.m` so that it prints "Hello, Simulator!" when run on the iPhone Simulator and "Hello, Device!" when it runs on a device. Modify the highlighted lines in Listing 3-2 to implement this change.

Listing 3-2 Conditionalizing code for the iPhone Simulator or a device

```
- (void) drawRect:(CGRect) rect {
    /* Draw "Hello, World!" */
    #if TARGET_IPHONE_SIMULATOR
        NSString *hello   = @"Hello, iPhone Simulator!";
    #else
        NSString *hello   = @"Hello, Device!";
    #endif
```

```
    CGPoint  location = CGPointMake(10, 20);
    UIFont   *font    = [UIFont systemFontOfSize:24];
    [[UIColor whiteColor] set];
    [hello drawAtPoint:location withFont:font];
}
```

Now when you build and run the application in the iPhone Simulator, the message targeted at the Simulator appears on the screen. In order for your application to build for the simulator and for your device, you may also need to conditionalize framework linking. "Conditional Linking to System Frameworks" (page 66)describes this process.

To view your application's Console output use the Organizer, as described in "Preparing Devices for Development" (page 55).

Capabilities of the iPhone Simulator

The iPhone Simulator makes it easy to test your applications using the power and convenience of your desktop or laptop computer. Although, your development computer may not simulate complicated touch events, such as multifinger touches, the Simulator lets you perform pinches. To perform a pinch, hold Option while tapping on the Simulator screen.

iPhone OS supports the Objective-C runtime introduced in Mac OS X v10.5 except for access to Objective-C class metadata. This means that, if your application accesses Objective-C class metadata, it may not run on the simulator. See *Objective-C 2.0 Runtime Reference* for more information.

Working with a Device

Building your application for an iPhone OS–based device provides the most realistic environment for you to test its operation and efficiency. Xcode makes building the application, transferring it to the device, and launching it a simple affair.

The following sections describe how to build and run your application for iPhone OS–based devices and how to manage them in the Organizer.

Preparing Devices for Development

In order to test your application on a device, you must configure your computer and your device for iPhone OS development. In this process, you create or obtain the following digital assets.

- **Certificate signing request.** A certificate signing request (CSR) contains personal information used to generate your development certificate. You submit this request to the iPhone Developer Program Portal.
- **Development certificate.** A development certificate identifies an iPhone application developer. After the CSR is approved, you download your developer certificate from the portal and add it to your Keychain.

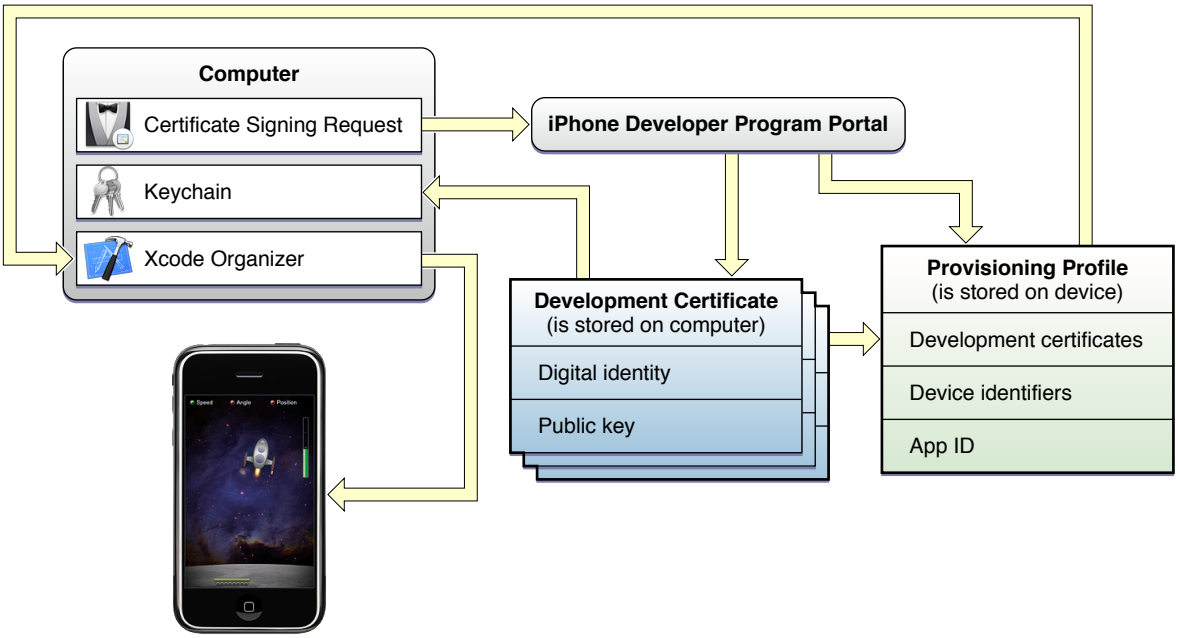
  When you build your iPhone application with Xcode, it looks for your development certificate in your Keychain; if it finds the certificate, Xcode signs your application, otherwise, it reports a build error.

- **Provisioning profile.** A provisioning profile associates one or more development certificates, devices, and an iPhone application ID (a unique identifier for the iPhone applications you or your organization develop under an iPhone Developer Program contract).

  To be able to install iPhone applications signed with your development certificate on a device, you must install at least one provisioning profile on the device. This provisioning profile must identify you (through your development certificate) and your device (by listing its unique device identifier). If you're part of an iPhone Developer team, other members of your team, with appropriately defined provisioning profiles, may run applications you build on their devices.

Figure 3-7 illustrates the relationship between these digital assets.

Figure 3-7 Preparing computers and devices for iPhone development



These are the requirements your computer and your development device must meet so that you can build iPhone applications that run on your device:

- Your computer must have your development certificate in your Keychain.
- Your device must contain at least one provisioning profile that contains your developer certificate and identifies your device.
- Your development device must have iPhone OS 2.0 or later installed.

These are the steps you must follow to configure your computer and development device for iPhone development:

1. Become an iPhone Developer Program member
2. Register your device with the Program Portal
3. Install iPhone OS on your device

4. Obtain your development certificate
5. Add your development certificate to your Keychain
6. Obtain your provisioning profile
7. Add your provisioning profile to Xcode
8. Install your provisioning profile on your device

The following sections describe these tasks in detail.

Becoming an iPhone Developer Program Member

In order to become an iPhone Developer Program member you, or an agent of your organization, enroll in the program through the iPhone Dev Center.

Important: To have access to the iPhone Developer Program Portal from the iPhone Dev Center, you must be a member of the iPhone Developer Program.

If you're part of an organization that enrolled in the iPhone Developer program, a team admin must add you as a member of your organization's team.

After becoming a member of the iPhone Developer Program, you must get an iPhone **application ID**. This ID is used by iPhone OS to identify the applications you create. Therefore, if you plan on creating more than one application, you should create an ID prefix (an application ID with an asterisk at the end), which can be shared by your applications; each application specifies its full ID in its information property list (`Info.plist`) file (see "The Information Property List" (page 95) for details).

In the following sections iPhone Developer Program members that are sole developers are identified as **sole members**. Members that are part of a team are known as **team members**.

Registering Your Device with the Program Portal

To register your development device with the portal:

1. Launch Xcode.
2. Choose Window > Organizer to open the Organizer window.
3. Plug-in your device and select it in the devices list.
4. Copy your device UDID from the Identifier text field in the Summary pane.
5. If you're a sole member, go to the portal to register your device. Otherwise, send your team admin your device UDID so that they can register it on the Portal.

Installing iPhone OS on Your Device

To run applications you develop using the iPhone SDK, your device must be running iPhone OS 2.0 or later.

To learn how to install iPhone OS on your device, see "Restoring System Software" (page 61).

Obtaining Your Development Certificate

Xcode uses your development certificate to code-sign your application before it uploads it to your device for testing.

Start by generating a certificate signing request (CSR) on your computer:

1. Launch Keychain Access, located in `/Applications/Utilities`.
2. Choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority.
3. In the Certificate Information window:
    a. In the User Email Address field, enter your email address.
    b. In the Common Name field, enter your name.
    c. In the "Request is" group, select the "Saved to disk" option.
    d. Select "Let me specify key pair information."
    e. Click Continue.
    f. Choose a location for the CSR file.
    g. In the Key Pair Information pane, choose 2048 as the key size and RSA as the algorithm.

    The Certificate Assistant saves a CSR file to your Desktop.

    This process creates a public/private key pair. The public key is stored in your development certificate. Your private key is stored in your Keychain. You must ensure that you don't loose your private key and that only you have access to it. Therefore, it's a good idea to backup your private key. Backing up your private key may also help if you need to use more than one computer to develop iPhone applications. See "Backing Up Your Digital Identifications" (page 62) for more information.
4. Open the CSR file in a text editor and copy the entire text, including the enclosing tags.
5. Submit the CSR to the Program Portal.
6. If you're a sole member, approve your CSR. Otherwise, wait for approval.

After the CSR is approved, you can download your development certificate from the Program Portal. As with your private key, you should backup your development certificate in case you need to develop iPhone applications on another computer. See "Backing Up Your Digital Identifications" (page 62) for details.

Adding Your Development Certificate to Your Keychain

Your development certificate must be in your Keychain so that Xcode can digitally sign your iPhone applications.

To add your development certificate to your Keychain, in your computer:

1. Open your development certificate with the Keychain Access application by double-clicking it or dragging onto the Keychain Access application icon.
2. In the Add Certificates dialog, ensure Keychain is set to "login" and click OK.

Obtaining Your Provisioning Profile

To obtain your provisioning profile:

1. If you're a sole member, create your provisioning profile in the Program Portal. Otherwise, your team admin creates one or more provisioning profiles for your team.
2. Download your provisioning profile from the Program Portal.

Adding Your Provisioning Profile to the Xcode Organizer

You use the Organizer to add provisioning profiles to your development device.

To an a provisioning profile to Xcode:

1. Drag the provisioning profile file onto the Xcode icon in the Dock.
2. Restart Xcode.

After this operation, the `~/Library/MobileDevice/Provisions` directory should contain your provisioning profile and it should also appear in the Organizer > Summary > Provisioning pane.

Installing Your Provisioning Profile on Your Device

After adding your provisioning profile to the Organizer, you can add it to your device:

1. Open the Organizer window.

   Your provisioning profile should appear in the Summary > Provisioning pane. If you don't see it there, follow the instructions in "Adding Your Provisioning Profile to the Xcode Organizer" (page 59).
2. Plug-in your device and select it in the devices list.
3. Click the checkbox next to the provisioning profile to install it on your device.

   Once installed, a checkmark should appear in the checkbox next to the provisioning profile. If the checkmark doesn't appear, ensure that the provisioning profile includes your device UDID, your development certificate, and a valid application ID. Go to the Program Portal or contact your team admin to verify that the provisioning profile contains this information. You need to go back to the "Obtaining Your Provisioning Profile" (page 59) step if changes are made to your profile.

Building Your Application for a Device

To build your application for a device:

1. Select the device SDK from the Project > Set Active SDK menu or the Overview toolbar item.
2. Choose Build > Build (or Build > Build and Run).

   The status bar in the project window indicates whether the build was successful or whether there are build errors or warnings. You can view build errors and warnings in the text editor or the project window.

iPhones and iPod touches support two instruction sets, ARM and Thumb. Xcode uses Thumb instructions by default because using Thumb typically reduces code size by about 35 percent relative to ARM. Applications that have extensive floating point code might perform better if they use ARM instructions rather than Thumb. You can turn off Thumb for your application by turning off the Compile for Thumb build setting.

Running Your Application on a Device

To run the application you built in "Building Your Application for a Device" (page 60), choose Run > Run. If you have more than one devices attached to your computer, you can choose the device onto which Xcode puts the built application using the Project > Set Active Executable menu.

Once running, you can test that your application performs as you intend using all the capabilities of your device. You should especially ensure that your application uses the device's resources—CPU, memory, battery, and so on—as efficiently as possible. See "Tuning Application Performance" (page 65) for more information.

Using the Organizer

To view your application's Console logs or crash information, or to take screenshots of your application as it runs, you use the Xcode Organizer window.

The following sections show how to use the Organizer to perform these tasks.

Viewing Console and Crash Logs

To view a device's console output:

1. Open the Organizer window.
2. Select the device whose console log you want to view.
3. Click Console.

You can use the search field to filter log entries. You can also save the log to a file.

The Crash Log pane in the Organizer contains information about application crashes. You may have to unplug and replug your device to refresh the crash list.

Capturing Screenshots

Screenshots help to document your application. This is also how you create your application's default image, which iPhone OS displays when the user taps your application's icon.

To capture a screenshot:

1. Configure your application's screen for the screenshot.

   Depending on your application's workflow, you may need to place breakpoint in your code and run your application until it reaches that point.

2. Open the Organizer window, select your device, and click Screenshots.

3. Click Capture.

   To make that screenshot your application's default image, click Save As Default Image.

Restoring System Software

When you develop applications use a particular version of the iPhone SDK, such as iPhone SDK 2.0, you should test those applications on devices running the iPhone OS version the SDK targets, such as iPhone OS 2.0.

The iPhone SDK contains the iPhone OS software for which you can develop applications.

To restore a device:

1. Launch Xcode and open the Organizer window.

2. Plug the device into your computer.

3. Select the device in the Devices list.

4. From the Software Version pop-up menu, choose the version of iPhone OS you want to place on the device.

   If the version you want to install is not listed in the Software Version pop-up menu:

   a. Download the iPhone OS release you want to install on the device from http://developer.apple.com.

   b. From the Software Version pop-up menu, choose "Other Version."

   c. Navigate to the disk image containing the iPhone OS developer software and click Open.

      Xcode extracts the iPhone OS software from the disk image. You can dispose of the disk image you downloaded.

   d. From the Software Version pop-up menu, choose the newly downloaded iPhone OS version.

5. Click Reset iPhone or Reset iPod, depending on your device's type.
6. Use iTunes to name your device.

Backing Up Your Digital Identifications

When you create a certificate signing request (CSR) to obtain your development certificate, you generated a public/private key pair. The public key is included in your development certificate. The private key is stored in your Keychain. With these two items in your computer, Xcode can code-sign the iPhone applications you build with it. If you need to use another computer to develop iPhone applications, you must transfer these items to the other computer and add them to your Keychain.

This section shows how to export your private key from your Keychain in your development computer, store your private key and development certificate in a protected disk image, and add both items to a second computer for iPhone development.

To export your private key from your Keychain:

1. Launch Keychain Access.
2. In the category list, select Keys.
3. Select the private key you use for iPhone development.
4. Choose Export from the private key shortcut menu.

   (To display the private key shortcut menu, Control-click the selected row.)
5. Enter a password to protect the private key.
6. Select a location for the private key and use the Personal Information Exchange (`.p12`) format for the file.

To generate a protected disk image containing your private key and development certificate:

1. Place your private key and development certificate file in a newly created folder, named `iPhone Developer Identifications`.
2. Launch the Disk Utility application, located in `/Applications/Utilities`.
3. Choose File > New > Disk Image from Folder.
4. Choose the `iPhone Developer Identifications` directory you created earlier.
5. Select a location for the new protected disk image.
6. From the Encryption pop-up menu, choose "256-AES encryption".
7. In the dialog that appears, enter a password for the disk image.

   You should deselect the "Remember password in my keychain" option. Otherwise, anybody with access to your computer may open the disk image.

8. Place the protected disk image in a secure location.

Now, when you need to develop iPhone applications on another computer:

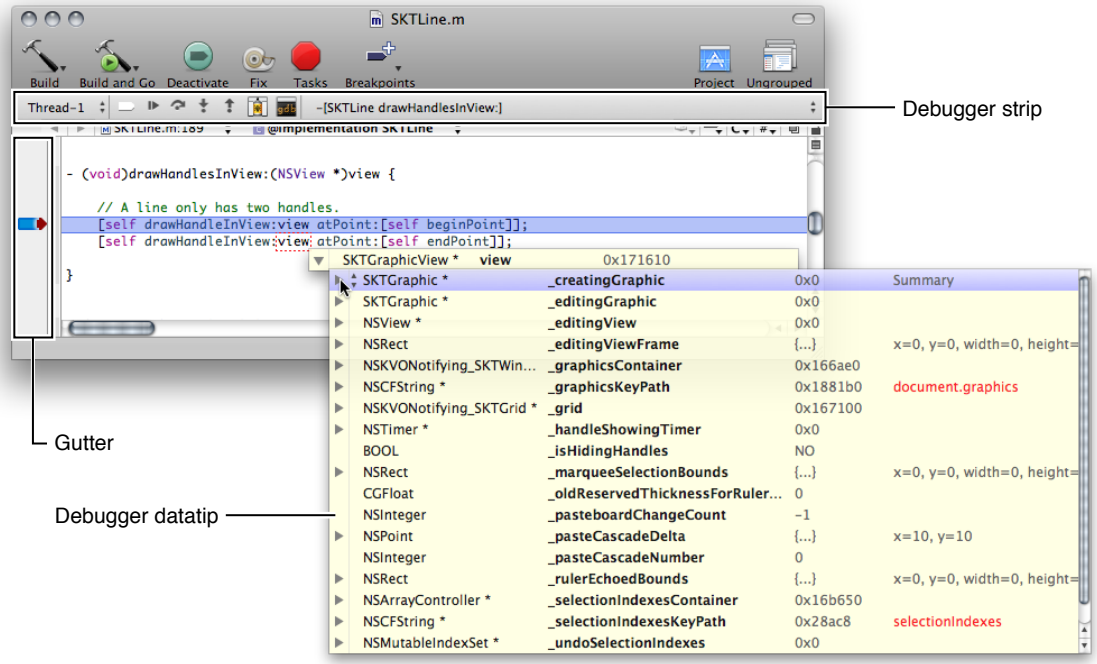
1. Copy the `iPhone Developer Identifications.dmg` disk-image file to the second computer.
2. On the second computer:
    a. Open the disk image.
    b. Import the private key into your Keychain:
        a. Launch Keychain Access.
        b. Choose File > Import Items.
        c. Choose the private key file to import.
3. Add the development certificate to your Keychain. See "Adding Your Development Certificate to Your Keychain" (page 58) for details.

Debugging Your Code and Measuring Performance

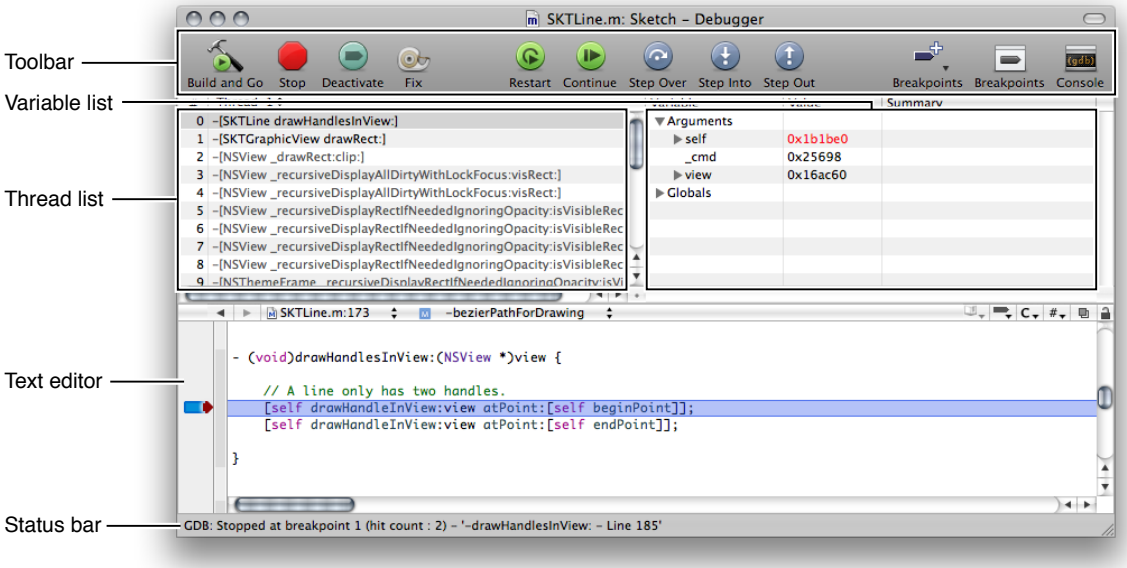
Debugging with Xcode

Xcode provides several debugging environments you can use to find and squash bugs in your code:

- **The text editor.** The text editor allows you to debug your code right *in* your code. It provides most of the debugging features you need. You can
    - Add and set breakpoints
    - View your call stack per thread
    - View the value of variables by hovering the mouse pointer over them
    - Execute a single line of code
    - Step in to, out of, or over function or method calls



- **The Debugger window.** When you need to perform more focused debugging, the Debugger window provides all the debugging features the text editor provides using a traditional interface. This window provides lists that allow you to see your call stack and the variables in scope at a glance.



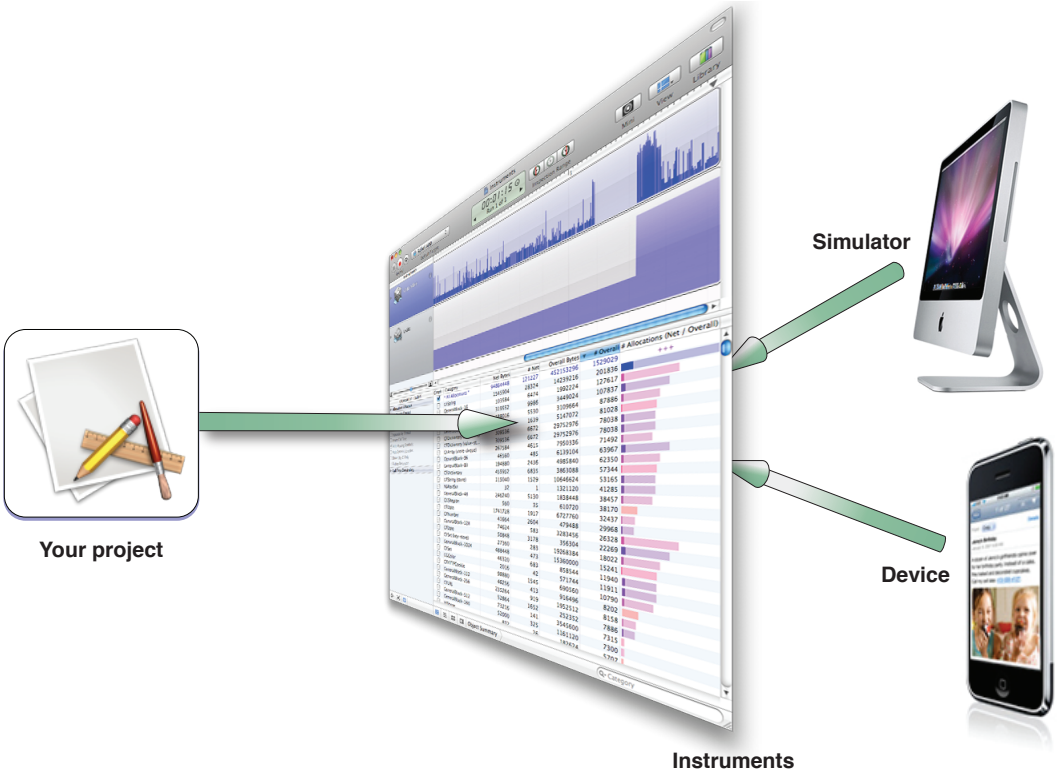
- **The GDB Console.** A GDB console window is available for text-based debugging.

For more information about the Xcode debugging facilities, see *Xcode Debugging Guide*.

Tuning Application Performance

Optimizing your application's performance is an important phase of the development, more so in iPhone OS–based devices, which, although powerful computing devices, do not have the memory or CPU power that desktop or laptop computers possess. You also have to pay attention to your application's battery use, as it directly impacts in your customer's battery-life experience.

The Instruments application lets you gather a variety of application performance metrics, such as memory and network use. It lets you gather data from iPhone applications running on the Simulator or on devices.



To complement the performance data Instruments collects, the Shark application lets you view system-level events, such as system calls, thread scheduling decisions, interrupts, and virtual memory faults. You can see how your code's threads interact with each other and how your code interacts with iPhone OS. See *Shark User Guide* to learn more about general Shark usage.

To learn more about measuring and analyzing application performance, see *Instruments User Guide*. This document provides general information about Instruments.

It is important that your iPhone applications use the resources of iPhone OS–based devices as efficiently as possible to provide their users a compelling experience. For example, your application should not use resources in a way that makes the application feel sluggish to the user or drains their batteries too quickly. Applications that use too much memory run slower. Applications that rely on the network for their operation, must use it as sparingly as possible because powering up the radios for network communications is a significant drag on the battery.

The Instruments application provides an advanced data gathering interface that lets you know exactly how your application uses resources, such as the CPU, memory, file system, and so on.

Instruments uses probes, known as instruments, to collect performance data. An **instrument** collects a specific type of data, such as network activity or memory usage. You find which instruments are available for iPhone OS in the Instruments Library.

To measure your application's performance:

1. Build and run your application on the device as described in "Working with a Device" (page 55).
2. Launch Instruments.

   The Instruments application is located at `<Xcode>/Applications`. (`<Xcode>` refers to the installation location of the development tools.)
3. Choose a template, such as Activity Monitor, to create the trace document.

   A **trace document** contains one or more instruments that collect data about a process.
4. From the Default Target toolbar item, select the iPhone OS–based device containing the application from which you want to collect performance data.
5. Add or remove instruments from the trace document to collect the desired data.
6. Use the Default Target toolbar item, to launch or attach to the target application.
7. Click Record to start collecting data and use your application, exercising the areas whose you want to examine.

For more information about using Instruments to measure performance, see *Instruments User Guide*.

When performance problems in your code are more related to the interaction between your code, iPhone OS, and the hardware architecture of the device, you may use Shark to get information about those interactions and find performance bottlenecks. For information about using Shark on your iPhone applications, see *Shark User Guide*.

Conditional Linking to System Frameworks

There may be occasions when you need to configure your application target so that it links against one framework to run on the iPhone Simulator and another framework to run on a device. For example, the `CFNetwork` API is a stand-alone framework (`CFNetwork.framework`) in the iPhone SDK for a device but a subframework of the Core Services framework (`CoreServices.framework`) on the iPhone SDK for the iPhone Simulator.

In this case, you need to specify the framework linking details for the `CFNetwork` API separately for the simulator and for a device. Follow these steps:

1. In Xcode, choose Project > Edit Active Target "<application_target>" to open the target editor.
2. Click the Build tab to display the build settings editor.

3. From the Show pop-up menu, choose All Settings.
4. Select Linking > Other Linker Flags.
5. Add the device linking details:
    a. Select the Linking > Other Linker Flags build setting.
    b. Choose Add Build Setting Condition from the Action (gear) pop-up menu.
    c. In the Value column, enter `-framework CFNetwork`.
    d. From the Any SDK pop-up menu in the Title column, choose Device - iPhone.
6. Add the simulator linking details:
    a. Select the Linking > Other Linker Flags build setting.
    b. Choose Add Build Setting Condition from the Action (gear) pop-up menu.
    c. In the Value column, enter `-framework CoreServices`.
    d. From the Any SDK pop-up menu in the Title column, choose Simulator - iPhone.

Managing Application Data

As you develop your application, you might need to rely on user settings and application data to remain on the iPhone Simulator or your development device between builds. Xcode doesn't remove any user settings or application data as you build your project and install the application on its host. But you may need to erase that information as part of testing your application the way users will use it. You may also need to remove application data after performing a clean build, to ensure that the newly installed application reflects structural changes the application may have, such as reorganized localized resources.

To remove all application data from the iPhone Simulator or a device, remove the application from the Home screen. See "Uninstalling Applications" in *iPhone Simulator Programming Guide* for details.

Application Design Guidelines

Designing applications for iPhone OS is not like designing applications for Mac OS X or other desktop operating systems. The iPhone and iPod touch are designed to be carried in a pocket but still provide much of the same behavior as a much larger desktop computer. They must also be able to communicate over Wi-Fi or cell networks and still be able to run for extended periods of time on battery power. They must also be fast and responsive to the user, rather than spend a lot of time booting the system or launching applications. All of these factors require you to take a slightly different approach when creating your own applications.

Although the iPhone and iPod touch have great power for their size, they are not as powerful as a Macintosh computer. iPhone OS goes to great lengths to ensure that the underlying system code runs at peak efficiency and conserves the available resources. That said, your own applications must also be designed with similar efficiency in mind. In addition to managing power, your application must moderate its use of system resources, such as memory, disk space, graphics, and CPU.

Beyond just managing resources, your applications also need to take a different approach when it comes to design. Efficiency comes in many forms, and an efficient workflow is one way to improve both resource usage and overall acceptance by users. You must create an application that can be used quickly by the user, providing relevant information right away rather than forcing the user to navigate endless screens of data. Your application's interface must be tailored not only to run on a smaller screen but also to use an entirely different set of interaction techniques. The Multi-Touch event model gives you options for interacting with the user that are simply not possible in a desktop application, and are a great strength of iPhone OS.

This chapter describes the design considerations that must go into creating an iPhone application. All of the information in this chapter is crucial for ensuring that your application runs well in iPhone OS and will be successful with users.

The Runtime Environment

The runtime environment of iPhone OS is designed for the fast and secure execution of programs. The following sections describe the key aspects of this runtime environment and provide guidance on how best to operate within it.

Fast Launch, Short Use

The strength of iPhone OS–based devices is their immediacy. A typical user pulls a device out of a pocket or bag and uses it for a few seconds, or maybe a few minutes, before putting it away again. The user might be taking a phone call, looking up a contact, changing the current song, or getting some piece of information during that time. If it takes a long time to do any of these things, the user is less likely to use the device. Because of this, your applications should be designed to launch and perform tasks quickly.

In addition to launching quickly, your application must be prepared to exit quickly too. Whenever the user leaves the context of your application, whether by pressing the Home button or using a feature that opens content in another application, iPhone OS tells your application to quit. At that time, you need to save any unsaved changes to disk and exit as quickly as possible, because if your application takes longer than 5 seconds to quit, it may be terminated outright. Because write speeds for the disk are typically slower than read speeds, you should be frugal in choosing what information you want to save.

In addition to saving out changes to user data, you should save out any application state information that is needed to return the user to the same place in your application when it is relaunched. Returning your application to the same state gives users the impression that your application never quit and makes it easier for them to pick up where they left off. Forcing users to navigate through the same set of screens each time your application launches is a frustrating experience.

The Virtual Memory System

To manage program memory, iPhone OS uses essentially the same virtual memory system found in traditional desktop systems. In iPhone OS, each program still has its own virtual address space, but the amount of usable virtual memory in iPhone OS is constrained by the amount of physical memory available. This is because iPhone OS does not write volatile pages to disk when memory gets full. Instead, the virtual memory system frees up nonvolatile memory, as needed, to make sure the running application has the space it needs. It does this by removing memory pages that are not being used and that contain read-only contents, such as code pages. Such pages can always be loaded back into memory later if they are needed again.

If memory continues to be constrained, the system may also send notifications to the running applications, asking them to free up additional memory. All applications should respond to this notification and do their part to help relieve the memory pressure. For information on how to handle such notifications in your application, see "Observing Low-Memory Notifications" (page 72).

Managing Your Memory Usage

Because the iPhone OS virtual memory model does not include disk swap space, you must be careful to allocate no more memory than is available on the device. iPhone OS warns the running application when low-memory conditions occur and may terminate the application if the problem persists. Being responsive about your application's memory usage and cleaning up memory in a timely manner are therefore crucial.

The following sections provide guidance on how to use memory efficiently and how to respond when there is only a small amount of available memory.

Reducing Your Application's Memory Footprint

Table 4-1 lists some tips on how to reduce your application's overall memory footprint. Starting off with a low footprint gives you more room for the data you need to manipulate.

Table 4-1 Tips for reducing your application's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iPhone OS, your application should not have any memory leaks. Allowing leaks to exist means your application may not have the memory it needs later. You can use the Instruments application to track down leaks in your code, both in the simulator and on actual devices. For more information on using Instruments, see *Instruments User Guide*.
Make resource files as small as possible.	Files reside on the disk but must be loaded into memory before they can be used. Property list files and images are two resource types where you can save space with some very simple actions. To reduce the space used by property list files, write those files out in a binary format using the `NSPropertyListSerialization` class. For images, compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iPhone applications—use the `pngcrush` tool.)
Use SQLite for large data sets.	If your application manipulates large amounts of structured data, store it in a SQLite database instead of in a flat file. SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but actually slows down your application right away. In addition, if you end up not using the resource, loading it simply wastes memory.
Build your program using Thumb.	Adding the `-mthumb` compiler flag can reduce the size of your code by up to 35%. Be sure to turn this option off for floating-point intensive code modules, however, because the use of Thumb on these modules can cause performance to degrade.

Allocating Memory Wisely

iPhone applications use a managed memory model, whereby you must explicitly retain and release objects. Table 4-2 lists tips for allocating memory inside your program.

Table 4-2 Tips for allocating memory

Tip	Actions to take
Reduce your use of autoreleased objects.	Objects released using the `autorelease` method stay in memory until you explicitly drain the autorelease pool or until the next time around your event loop. Whenever possible, avoid using the `autorelease` method when you can instead use the `release` method to reclaim the memory occupied by the object immediately. If you must create moderate numbers of autoreleased objects, create a local autorelease pool and drain it periodically to reclaim the memory for those objects before the next event loop.
Impose size limits on resources.	Avoid loading large resource files when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iPhone OS–based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the `mmap` and `munmap` functions to map portions of the file into and out of memory. For more information about mapping files into memory, see *File-System Performance Guidelines*.
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your application may be unable to complete the calculations. Your applications should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information on how to allocate memory in iPhone applications, and for more information on autorelease pools, see Cocoa Objects in *Cocoa Fundamentals Guide*.

Observing Low-Memory Notifications

When the system dispatches a low-memory notification to your application, heed the warning. iPhone OS notifies the frontmost application whenever the amount of free memory dips below a safe threshold. If your application receives this notification, it must free up as much memory as it can by releasing objects it does not need or clearing out memory caches that it can recreate easily later.

UIKit provides several ways to receive low-memory notifications, including the following:

- Implement the `applicationDidReceiveMemoryWarning:` method of your application delegate.
- Override the `didReceiveMemoryWarning` method in your custom `UIViewController` subclass.
- Register to receive the `UIApplicationDidReceiveMemoryWarningNotification` notification.

Upon receiving any of these notifications, your handler method should respond by immediately freeing up any unneeded memory. View controllers should purge any views that are currently offscreen, and your application delegate should release any data structures it can or notify other application objects to release memory they own.

If your custom objects have known purgeable resources, you can have those objects register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and release their purgeable resources directly. Registering for the `UIApplicationDidReceiveMemoryWarningNotification`

notification is appropriate if you have a few objects that manage most of your purgeable resources and it is appropriate to purge all of those resources. If you have many purgeable objects or want to coordinate the release of only a subset of those objects, however, you might want to use your application delegate to release the desired objects.

Important: Like the system applications, your applications should always handle low-memory warnings, even if they do not receive those warnings during your testing. System applications consume small amounts of memory while processing requests. When a low-memory condition is detected, the system delivers low-memory warnings to all running programs (including your application) and may terminate some background applications (if necessary) to ease memory pressure. If not enough memory is released—perhaps because your application is leaking or still consuming too much memory—the system may still terminate your application.

Performance and Responsiveness

Performance and responsiveness are two areas that must be factored into your designs for iPhone applications. When an application uses the network frequently or is not optimized for the tasks its performing, it wastes precious battery power. In addition, an application that is inefficient risks looking sluggish. On mobile devices, users expect the software to respond quickly and may stop using applications that do not.

The following sections offer tips and guidance on how to make sure your iPhone OS-based applications are efficient, responsive, and good at conserving power.

Using Memory Efficiently

An effective way to improve the performance of your application is to reduce the amount of resident private memory you use as much as possible. Because the system does not use a swap file, your own code should monitor its memory usage, eliminate leaks, and respond to low-memory warnings when they arrive. For more information, see "Managing Your Memory Usage" (page 70).

Improving Drawing Performance

Creating high-quality graphics is a processor-intensive task. You should spend plenty of time optimizing your application's drawing code to ensure that it is as fast and efficient as possible. Check your drawing code for hot spots using the Instruments application. If your drawing code is written using OpenGL ES, you can also use the OpenGL ES instrument to gather OpenGL–specific statistics.

Applications have two main paths for drawing: OpenGL and the native platform technologies. The OpenGL path is well suited for game developers and applications that need to produce quality graphics with high frame rates. The native technologies such as Quartz and UIKit provide higher-level drawing interfaces and are well suited for applications that perform on-demand updates and detail-oriented drawing. Core Animation is another native technology that is used to animate rendered content using a highly-optimized rendering path. For best performance, you should never mix OpenGL drawing surfaces with views that draw using native platform technologies.

For additional tips on how to optimize your overall drawing code, see "Drawing Tips" (page 156).

Reducing Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iPhone OS conserves power by shutting down any hardware features that are not currently being used. Your application can help improve battery life by minimizing the amount of time you spend using these features. Here are some actions that you should take to minimize your hardware usage:

- Connect to the external servers via the network only when needed. Avoid polling.
- When you must connect to the network, transmit the smallest amount of data possible.
- If you obtain the user's current location using the Core Location framework, disable location updates as soon as you have a position fix.

Tuning Your Code

iPhone OS comes with several applications for tuning the performance of your application. Most of these tools run on Mac OS X and are suitable for tuning some aspects of your code while it runs in the simulator. For example, you can use the simulator to eliminate memory leaks and make sure your overall memory usage is as low as possible. You can also remove any hotspots in your code that might be caused by an inefficient algorithm or a previously unknown bottleneck.

After you have tuned your code in the simulator, you should then use the Instruments application to further tune your code on a device. Running your code on an actual device is the only way to tune your code fully. Because the simulator runs in Mac OS X, it has the advantage of a faster CPU and more usable memory, so its performance is generally much better than the performance on an actual device. And using Instruments to trace your code on an actual device may point out additional performance bottlenecks that need tuning.

Security

Protecting the integrity of the user's data on any computer system is everyone's responsibility. iPhone OS provides several levels of security to help prevent malicious users from taking advantage of flaws in your application. At the same time, you should do everything possible to eliminate vulnerabilities in your code by engaging in secure coding practices.

The following sections detail the security provided by iPhone OS and what you can do in your own code to foil hacker attacks.

The Application Sandbox

For security reasons, iPhone OS restricts an application and its preferences and data to a unique location in the file system. This location is part of the security feature known as the application's "sandbox." The **sandbox** is a set of fine-grained controls limiting access to files, preferences, network resources, hardware, and so on. In iPhone OS, an application and its data reside in a secure location that no other application can access. When an application is installed, the system computes a unique

opaque identifier for the application. Using a root application directory and this identifier, the system constructs a path to the application's home directory. Thus an application's home directory could be depicted as having the following structure:

/ApplicationRoot/ApplicationID/

As part of the installation process, the system configures the runtime environment of the application, copies the application bundle to its home directory, and creates the subdirectories of the home directory. By providing a unique location for an application and its data, the application sandbox simplifies operations such as backup-and-restore and uninstallation.

It's important to note that the sandbox does not protect your application from direct attacks by malicious entities. For example, if you accept input from the user, don't validate it, and there is an exploitable buffer overflow in your input-handling code, an attacker might be able to cause your program to crash or even take control of the program so that it executes the attacker's code. The sandbox limits the damage an attacker can cause, but it cannot prevent attacks.

For more information about accessing the application-specific directories created for each application, see "Application Directory Structure" (page 76).

Using the Available Security Technologies

The following sections describe the security-related interfaces and show how you would use them in your applications.

Certificate, Key, and Trust Services

You use the Certificate, Key, and Trust Services API to do the following:

- Evaluate trust for a certificate
- Retrieve information from a certificate
- Convert between a DER (Distinguished Encoding Rules) representation of a certificate and a certificate keychain item
- Retrieve the certificate associated with a specific cryptographic identity
- Retrieve the private key that's associated with a specific cryptographic identity
- Create a pair of asymmetric keys
- Use a private key to generate a digital signature for a block of data
- Use a public key to verify a signature
- Use a public key to encrypt a block of data
- Use a private key to decrypt a block of data
- Retrieve the value of a trust policy
- Set anchor certificates

For more information on how to use these services, see *Certificate, Key, and Trust Services Programming Guide* and *Certificate, Key, and Trust Services Reference*.

Keychain Services

You can use the keychain to store certificates and cryptographic keys, or store passwords and other data. You can create, modify, and delete items in the keychain. Each keychain item has a number of attributes that specify the purpose of the item, whether it is encrypted, when it was created and by whom, and so forth. You can search for items with any combination of attributes.

iPhone OS gives you access only to your application's own keychain items (with the exception of items for which another application has passed you a persistent reference). For this reason, the user is never asked to authorize access to a keychain item.

For more information about accessing the keychain, see *Keychain Services Programming Guide* and *Keychain Services Reference*.

Randomization Services

Randomization services provides a cryptographically secure pseudo random number generator. You can use this feature to generate unique IDs or passwords or use it in any situation where the standard `rand` function call (in the `LibSystem` library) is not sufficiently random.

For more information about randomization services, see *Randomization Services Reference*.

File and Data Management

Files in iPhone OS share space on the flash-based memory with the user's media and personal files. For security purposes, your application is placed in its own directory and is limited to reading and writing files in that directory only. The following sections describe the structure of your application's local file system and several techniques for reading and writing files.

Application Directory Structure

For security purposes, an application has only a few locations in which it can write its data and preferences. When an application is installed on a device, a home directory is created for the application. (For more information about the application home directory itself, see "The Application Sandbox" (page 74).) Inside that directory are several custom subdirectories, which are listed in Table 4-3. Some directories, such as the application bundle directory, cannot be modified by your code, but others can.

Table 4-3 Directories of an iPhone application

Directory	Description
<*Application_Home*>/*AppName*`.app`	This is the bundle directory containing the application itself. Because an application must be signed, you must not make changes to the contents of this directory at runtime. Doing so may prevent your application from launching later.

Directory	Description
<*Application_Home*>/Documents/	This is the directory you should use to write any application-specific data files. For information about how to get the path of this directory, see "Getting Paths to Application Directories" (page 78).
<*Application_Home*>/Library/Preferences/	This directory contains the preferences file for the application. You should not create files in this directory yourself. Instead, use the `NSUserDefaults` class or *CFPreferences* API to access the application preferences. For more information, see "Accessing Your Preferences" (page 204).
<*Application_Home*>/tmp/	This is the directory you should use to write any temporary files. Your application is responsible for cleaning up the contents of this directory; there is no automatic cleanup mechanism. For information about how to get the path of this directory, see "Getting Paths to Application Directories" (page 78). Remember that the contents of this directory are not backed up automatically.

Except for the `tmp` directory, the contents of the directories in Table 4-3 (page 76) are backed up automatically when the user syncs the device to iTunes. For more information about the backup and restore process, see "Backup and Restore" (page 77).

When the system installs an application, it also sets environment variables for some of the application's directories. (An **environment variable** is a process-global symbol that represents some constant value.) Table 4-4 lists the environment variables that are set. Although you can use these environment variables for constructing paths to use for reading and writing data and preferences, there are programmatic alternatives that provide a better solution. For information about those programmatic solutions, see "Getting Paths to Application Directories" (page 78).

Table 4-4 Environment variables for the application sandbox

Directory	Environment variable
Application Home/	`HOME`
Application Home/tmp/	`TMPDIR`

Backup and Restore

You do not have to prepare your application in any way for backup and restore operations. When a device is plugged into a computer and synced, iTunes automatically backs up the applications and data on that device. The host computer copies everything in each application directory (minus the temporary directory) to a location on the host. If you wipe the device at a later date, iTunes will then ask you if you want to restore your applications and data from the backup data located on the host.

Getting Paths to Application Directories

At various levels of the system, there are programmatic ways to obtain file-system paths to the directory locations of an application's sandbox. However, the preferred way to retrieve these paths is with the Cocoa programming interfaces. The `NSHomeDirectory` function (in the Foundation framework) returns the path to the top-level home directory—that is, the directory containing the application, `Documents`, `Library`, and `tmp` directories. In addition to that function, you can also use the `NSSearchPathForDirectoriesInDomains` and `NSTemporaryDirectory` functions to get the paths to your `Documents` and `tmp` directories.

Both the `NSHomeDirectory` and `NSTemporaryDirectory` functions return the properly formatted path information in an `NSString` object. You can use the path-related methods of `NSString` to modify the path information or create new path strings. For example, upon retrieving the temporary directory path, you could append a file name and use the resulting string to create a file with the given name in the temporary directory. If you are using frameworks with ANSI C programmatic interfaces—including those that take paths—recall that `NSString` objects are "toll-free bridged" with their Core Foundation counterparts. This means that you can cast a `NSString` object (such as the return by one of the above functions) to a `CFStringRef` type, as shown in the following example:

```
CFStringRef homeDir = (CFStringRef)NSHomeDirectory();
```

Note: Many (but not all) Foundation types are toll-free bridged with their Core Foundation counterparts. For information on whether a Foundation class is toll-free bridged, see the overview for that class. For more information on toll-free bridging in general, see *Carbon-Cocoa Integration Guide*.

The `NSSearchPathForDirectoriesInDomains` function of the Foundation framework lets you obtain the full path to the application's `Documents/` directory. Because this function was designed originally for Cocoa applications, it returns an array of paths representing the potential Documents directories on the system. To use it in iPhone OS, specify `NSDocumentDirectory` for the first parameter and `NSUserDomainMask` for the second parameter. The resulting array should contain the single path to your application's `Documents` directory. Listing 4-1 shows a typical use of this function.

Listing 4-1 Getting a file-system path to the application's `Documents/` directory

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

Note: You can call `NSSearchPathForDirectoriesInDomains` using directory and domain-mask parameters other than `NSDocumentDirectory` and `NSUserDomainMask`, but the application will be unable to write to any of the returned directory locations. For example, if you specify `NSApplicationDirectory` as the directory parameter and `NSSystemDomainMask` as the domain-mask parameter, you get the path `/Applications` returned (on the device), but your application cannot write any files to this location.

Another consideration to keep in mind is the difference in directory locations between platforms. The paths returned by `NSSearchPathForDirectoriesInDomains`, `NSHomeDirectory`, `NSTemporaryDirectory`, and similar functions differ depending on whether you're running your application on the device or on the Simulator. For example, take the function call shown in Listing 4-1. On the device, the returned path (`documentsDirectory`) is similar to the following:

```
/var/mobile/Applications/30B51836-D2DD-43AA-BCB4-9D4DADFED6A2/Documents
```

However, on the Simulator, the returned path takes the following form:

```
/Volumes/Stuff/Users/johnDoe/Library/Application Support/iPhone
Simulator/User/Applications/118086A0-FAAF-4CD4-9A0F-CD5E8D287270/Documents
```

To read and write user preferences, use the `NSUserDefaults` class or the `CFPreferences` API. These interfaces eliminate the need for you to construct a path to the `Library/Preferences/` directory and read and write preference files directly. For more information on using these interfaces, see "Accessing Your Preferences" (page 204).

If your application contains sound, image, or other resources in the application bundle, you should use the `NSBundle` class or `CFBundle` opaque type to load those resources. Bundles have an inherent knowledge of where resources live inside the application. In addition, bundles are aware of the user's language preferences and are able to choose localized resources over default resources automatically. For more information on bundles, see "The Application Bundle" (page 93).

Reading and Writing File Data

The iPhone OS provides several ways to read, write, and manage files.

- Foundation framework:
  - If you can represent your application's data as a property list, convert the property list to an `NSData` object using the `NSPropertyListSerialization` API. You can then write the data object to disk using the methods of the `NSData` class.
  - If your application's model objects adopt the `NSCoding` protocol, you can archive a graph of these model objects using the `NSKeyedArchiver` class, and especially its `archivedDataWithRootObject:` method.
  - The `NSFileHandle` class in Foundation framework provides random access to the contents of a file.
  - The `NSFileManager` class in Foundation framework provides methods to create and manipulate files in the file system.
- Core OS calls:
  - Calls such as `fopen`, `fread`, and `fwrite` also let you read and write file data either sequentially or via random access.
  - The `mmap` and `munmap` calls provide an efficient way to load large files into memory and access their contents.

Note: The preceding list of Core OS calls is just a sample of the more commonly-used calls. For a more complete list of the available functions, see the list of functions in section 3 of *iPhone OS Manual Pages*.

The following sections show examples of how to use some of the higher-level techniques for reading and writing files. For additional information about the file-related classes of the Foundation framework, see *Foundation Framework Reference*.

Reading and Writing Property List Data

A property list is a form of data representation that encapsulates several Foundation (and Core Foundation) data types, including dictionaries, arrays, strings, dates, binary data, and numerical and Boolean values. Property lists are commonly used to store structured configuration data. For example, the Info.plist file found in every Cocoa and iPhone applications is a property list that stores configuration information about the application itself. You can use property lists yourself to store additional information, such as the state of your application when it quits.

In code, you typically construct a property list starting with either a dictionary or array as a container object. You then add other property-list objects, including (possibly) other dictionaries and arrays. The keys of dictionaries must be string objects. The values for those keys are instances of `NSDictionary`, `NSArray`, `NSString`, `NSDate`, `NSData`, and `NSNumber`.

For an application whose data can be represented by a property-list object (such as an `NSDictionary` object), you could write that property list to disk using a method such as the one shown in Listing 4-2. This method serializes the property list object into an `NSData` object, then calls the `writeApplicationData:toFile:` method (the implementation for which is shown in Listing 4-4 (page 82)) to write that data to disk.

Listing 4-2 Converting a property-list object to an `NSData` object and writing it to storage

```
- (BOOL)writeApplicationPlist:(id)plist toFile:(NSString *)fileName {
    NSString *error;
    NSData *pData = [NSPropertyListSerialization dataFromPropertyList:plist
format:NSPropertyListBinaryFormat_v1_0 errorDescription:&error];
    if (!pData) {
        NSLog(@"%@", error);
        return NO;
    }
    return ([self writeApplicationData:pData toFile:(NSString *)fileName]);
}
```

When writing property list files in iPhone OS, it is important to store your files in binary format. You do this by specifying the `NSPropertyListBinaryFormat_v1_0` key in the *`format`* parameter of the `dataFromPropertyList:format:errorDescription:` method. The binary property-list format is much more compact than the other format options, which are text based. This compactness not only minimizes the amount of space taken up on the user's device, it also improves the time it takes to read and write the property list.

Listing 4-5 shows the corresponding code for loading a property-list file from disk and reconstituting the objects in that property list.

Listing 4-3 Reading a property-list object from the application's `Documents` directory

```
- (id)applicationPlistFromFile:(NSString *)fileName {
    NSData *retData;
    NSString *error;
    id retPlist;
    NSPropertyListFormat format;

    retData = [self applicationDataFromFile:fileName];
    if (!retData) {
        NSLog(@"Data file not returned.");
        return nil;
    }
```

```
    retPlist = [NSPropertyListSerialization propertyListFromData:retData
mutabilityOption:NSPropertyListImmutable format:&format errorDescription:&error];
    if (!retPlist){
        NSLog(@"Plist not returned, error: %@", error);
    }
    return retPlist;
}
```

For more on property lists and the `NSPropertyListSerialization` class, see *Property List Programming Guide for Cocoa*.

Using Archivers to Read and Write Data

An archiver converts an arbitrary collection of objects into a stream of bytes. Although this may sound similar to the process employed by the `NSPropertyListSerialization` class, there is an important difference. A property-list serializer can convert only a limited set of (mostly scalar) data types. Archivers can convert arbitrary Objective-C objects, scalar types, arrays, structures, strings, and more.

The key to the archiving process is in the target objects themselves. The objects manipulated by an archiver must conform to the `NSCoding` protocol, which defines the interface for reading and writing the object's state. When an archiver encodes a set of objects, it sends an `encodeWithCoder:` message to each one, which the object then uses to write out its critical state information to the corresponding archive. The unarchiving process reverses the flow of information. During unarchiving, each object receives an `initWithCoder:` message, which it uses to initialize itself with the state information currently in the archive. Upon completion of the unarchiving process, the stream of bytes is reconstituted into a new set of objects that have the same state as the ones written to the archive previously.

The Foundation framework supports two kinds of archivers—sequential and keyed. Keyed archivers are more flexible and are recommended for use in your application. The following example shows how to archive a graph of objects using a keyed archiver. The `representation` method of the `_myDataSource` object returns a single object (possibly an array or dictionary) that points to all of the objects to be included in the archive. The data object is then written to a file whose path is specified by the `myFilePath` variable.

```
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:[_myDataSource
representation]];
[data writeToFile:myFilePath atomically:YES];
```

Note: You could also send send a `archiveRootObject:toFile:` message to the `NSKeyedArchiver` object to create the archive and write it to storage in one step.

To load the contents of an archive from disk, you simply reverse the process. After loading the data from disk, you use the `NSKeyedUnarchiver` class and its `unarchiveObjectWithData:` class method to get back the model-object graph. For example, to unarchive the data from the previous example, you could use the following code:

```
NSData* data = [NSData dataWithContentsOfFile:myFilePath];
id rootObject = [NSKeyedUnarchiver unarchiveObjectWithData:data];
```

For more information on how to use archivers and how to make your objects support the `NSCoding` protocol, see *Archives and Serializations Programming Guide for Cocoa*.

Writing Data to Your Documents Directory

Once you have an `NSData` object encapsulating the application data (either as an archive or a serialized property list), you can call the method shown in Listing 4-4 to write that data to the application `Documents` directory.

Listing 4-4 Writing data to the application's `Documents` directory

```
- (BOOL)writeApplicationData:(NSData *)data toFile:(NSString *)fileName {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    if (!documentsDirectory) {
        NSLog(@"Documents directory not found!");
        return NO;
    }
    NSString *appFile = [documentsDirectory
stringByAppendingPathComponent:fileName];
    return ([data writeToFile:appFile atomically:YES]);
}
```

Reading Data from the Documents Directory

To read a file from your application's Documents directory, construct the path for the file name and use the desired method to read the file contents into memory. For relatively small files—that is, files less than a few memory pages in size—you could use the code in Listing 4-5 to obtain a data object for the file contents. This example constructs a full path to the file in the `Documents` directory, creates a data object from it, and then returns that object.

Listing 4-5 Reading data from the application's `Documents` directory

```
- (NSData *)applicationDataFromFile:(NSString *)fileName {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    NSString *appFile = [documentsDirectory
stringByAppendingPathComponent:fileName];
    NSData *myData = [[[NSData alloc] initWithContentsOfFile:appFile]
autorelease];
    return myData;
}
```

For files that would require multiple memory pages to hold in memory, you should avoid loading the entire file all at once. This is especially important if you plan to use only part of the file. For larger files, you should consider mapping the file into memory using either the `mmap` function or the `initWithContentsOfMappedFile:` method of `NSData`.

Choosing when to map files versus load them directly is up to you. It is relatively safe to load a file entirely into memory if it requires only a few (3-4) memory pages. If your file requires several dozen or a hundred pages, however, you would probably find it more efficient to map the file into memory. As with any such determination, though, you should measure your application's performance and determine how long it takes to load the file and allocate the necessary memory.

File Access Guidelines

When creating files or writing out file data, keep the following guidelines in mind:

- Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the Flash disk, which has a limited lifespan. Some specific tips to help you minimize file-related operations include:
  - Write only the portions of the file that changed, but aggregate changes when you can. Avoid writing out the entire file just to change a few bytes.
  - When defining your file format, group frequently modified content together so as to minimize the overall number of blocks that need to be written to disk each time.
  - If your data consists of structured content that is randomly accessed, store it in a SQLite database. This is especially important if the amount of data you are manipulating could grow to be more than a few megabytes in size.
- Avoid writing cache files to disk. The only exception to this rule is when your application quits and you need to write state information that can be used to put your application back into the same state when it is next launched.

Saving State Information

When the user presses the Home button, iPhone OS quits your application and returns to the Home screen. Similarly, if your application opens a URI whose scheme is handled by a different application, iPhone OS quits your application and opens the URI in the other application. In other words, any action that would cause your application to suspend or go to the background in Mac OS X causes your application to quit in iPhone OS. Because these actions happen regularly on mobile devices, your application must change the way it manages volatile data and application state.

Unlike most desktop applications, where the user manually chooses when to save files to disk, your application should save changes automatically at key points in your workflow. Exactly when you save data is up to you, but there are two potential options. Either you can save each change immediately as the user makes it, or you can batch changes on the same page together and save them when the page is dismissed, a new page is displayed, or the application quits. Under no circumstances should you let the user navigate to a new page of content without saving the data on the previous page.

When your application is asked to quit, you should save the current state of your application to a temporary cache file or to the preferences database. The next time the user launches your application, use that information to restore your application to its previous state. The state information you save should be as minimal as possible but still let you accurately restore your application to an appropriate point. You do not have to display the exact same screen the user was manipulating previously if doing so would not make sense. For example, if a user edits a contact and then leaves the Phone application, upon returning, the Phone application displays the top-level list of contacts, rather than the editing screen for the contact.

Case Sensitivity

The file system for iPhone OS–based devices is case sensitive. Whenever you work with filenames, you should be sure that the case matches exactly or your code may be unable to open or access the file.

Networking

Applications that want to communicate over the network have several different options for doing so, including the `NSStream` classes in the Foundation framework, the CFNetwork framework, and the low-level functions in the Core OS layer of the system. When implementing code to receive or transmit across the network, remember that doing so is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving helps improve battery life. To that end, you should consider the following tips when writing your network-related code:

- For protocols you control, define your data formats to be as compact as possible.
- Avoid communicating using chatty protocols.
- Transmit data packets in bursts whenever you can.

The cellular and WiFi radios are designed to power down when there is no activity. Doing so can take several seconds though, depending on the radio. If your application transmits small bursts of data every few seconds, the radios may stay powered up and continue to consume power, even when they are not actually doing anything. Rather than transmit small amounts of data more often, it is better to transmit a larger amount of data once or at relatively large intervals.

When communicating over the network, it is also important to remember that packets can be lost at any time. When writing your networking code, you should be sure to make it as robust as possible when it comes to failure handling. It is perfectly reasonable to implement handlers that respond to changes in network conditions, but do not be surprised if those handlers are not called consistently. For example, the Bonjour networking callbacks may not always be called immediately in response to the disappearance of a network service. The Bonjour system service does immediately invoke browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This might occur if the device providing the network service unexpectedly loses network connectivity or the notification is lost in transit.

For more information about the objects of the CFNetwork framework, see *CFNetwork Framework Reference*.

User Interface Design Considerations

A good user interface design requires making appropriate choices about the information you choose to present to the user and the workflow you use to present it. Remember to think about the needs of the user on the go. Think about the information that is most relevant and how you can deliver that information quickly and with a minimal of screens. Use the presentation patterns that are most appropriate for the type of data you are presenting. And remember to make your interface visually appealing.

Creating an application that works well in iPhone OS involves following these high-level principles:

- Focus on what the user needs at any given moment
- Provide a well organized and easy-to-follow workflow
- Provide a clean and uncluttered layout for each screen in your workflow
- Use graphics effectively in your content
- Optimize your code for maximum performance
- Support standard iPhone paradigms, graphics, and gestures
- Provide appropriate feedback to the user
- Integrate appropriately with the system and other applications
- Minimize text entry

In an iPhone application, you must use space efficiently but not be so intent on fitting every last feature onto the screen that your interface is hard to use. Your designs should be clean and well organized, providing exactly the right information with just a few taps. If you are creating an iPhone OS version of an existing desktop application, this might mean leaving out less-important features. Porting applications feature for feature from the desktop is not recommended. Not only can it be hard to fit all of the necessary content on the smaller screen, it loses sight of the principle that you should focus on what the user needs at the moment. In a mobile context, providing information that is appropriate is more important than having lots of features.

For detailed information about designing your user interface, see *iPhone Human Interface Guidelines*.

The Application Environment

An iPhone application executes in a runtime and physical environment that determines its life cycle, where it reads and writes its data, and how it accesses its resources. The following sections describe this environment.

Core Application Architecture

An iPhone application executes in an runtime environment where it is continuously receiving events from the system and responding to those events, often by changing how it presents itself to the user. This continuous pattern of getting events and responding to them appropriately is called the event and drawing cycle. In these cycles, certain objects of an iPhone application play key roles. An iPhone application also has a runtime life cycle (described "The Application Life Cycle" (page 90)) that marks the key junctures from the moment it launches to the moment it quits.

The Event and Drawing Cycle

When an application starts running in iPhone OS, the operating system calls the standard entry point: the `main` function. The `main` function must always do three things:

1. Create a top-level autorelease pool.
2. Call the `UIApplicationMain` function.
3. Release the autorelease pool after `UIApplicationMain` returns.

When you create a project for an iPhone application in Xcode, the project template includes a `main` function with the above structure.

Note: An autorelease pool is used in memory management. It is a Cocoa mechanism for the deferred release of objects created during a functional block of code. For more on autorelease pools, see *Memory Management Programming Guide for Cocoa*. Also see "Application Design Guidelines" (page 69) for specific memory-management guidelines related to autorelease pools.

Listing 5-1 shows you what the main function of a typical iPhone application looks like.

Listing 5-1 The `main` function of an iPhone application

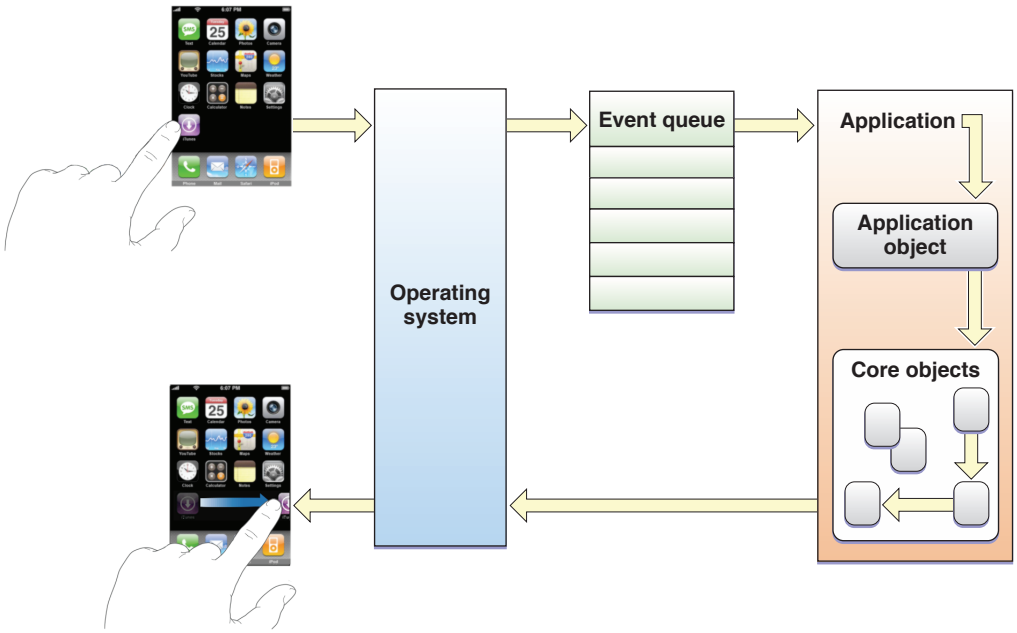
```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

In addition to the `argc` and `argv` parameters passed into `main`, the `UIApplicationMain` function includes string parameters that identify the principal class (that is, the class of the application object) and the class for the delegate of the application object. If you specify `nil` for the principal class, UIKit assumes it to be `UIApplication`. If you specify `nil` for the delegate, be sure to provide an instance of your delegate object in the application's main nib file.

For reasons described later in this chapter, the application object *must* have a delegate that implements methods of the `UIApplicationDelegate` protocol. The principal class can be `UIApplication` or a subclass of it; if you specify a subclass, you can specify the name of that subclass as the delegate class. When you do, iPhone OS automatically sends the instance of your subclass the application-delegate messages.

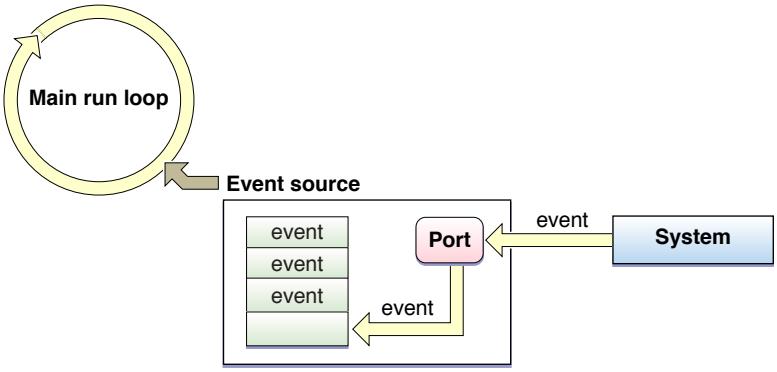
The principal job of the `UIApplicationMain` function is to set up the application's event and drawing cycle. Figure 5-1 depicts this cycle. The operating system detects touch events occurring on the device and places them in the application's event queue. The application object takes an event off the top of the queue and ensures that it is delivered to the object in the application that is best suited to handle it. In response to the handling of the event, one or more objects of the application might update the application's user interface appropriately.

Figure 5-1 The event and drawing cycle



In order for it to establish the event and drawing cycle, an application must establish a connection with the underlying system and to set up the main event loop. The **main event loop** is the run loop of the application's main thread with one input source set up for events coming from the underlying operating system. As events arrive, they are placed in a first-in first-out queue. Figure 5-2 illustrates the conceptual outlines of the main event loop.

Figure 5-2 The main event loop



Note: A run loop monitors sources of input to a process, such as an application. When these sources are ready for processing, the run loop dispatches control; when processing concludes, control passes back to the run loop, which then waits for the next event. Run loops can have multiple input sources, including ports and timers. Each thread of a process has its own run loop. You can use the `NSRunLoop` class of the Foundation framework to manage run loops. For more on `NSRunLoop` and run loops in general, see *Threading Programming Guide*.

The application gets an event from the event queue and dispatches it to the object that should next handle it. In most cases, that object is the `UIWindow` object representing the application's main window. The window object, in turn, forwards the event to the first responder. The **first responder** is typically the view object (`UIView`) of the application on which the touches associated with the event are taking place.

In iPhone OS's Multi-Touch event model, one or more fingers touching the screen—while possibly moving in different directions—constitute a discrete event that is encapsulated as a `UIEvent` object. An event object contains a set of `UITouch` objects that represent the finger touches associated with the event. The system tracks touches through the various phases of the event—touch down, touch moved, touch up—and modifies the state of each touch object as the touch progresses through the phases. The `UIResponder` class is the base class for all objects that respond to events, which include `UIApplication`, `UIWindow`, `UIView`, and all `UIView` subclasses. To respond to an event, a responder object implements one or more `UIResponder` methods that correspond to the event phases for touches.

The `UIResponder` class also defines the programmatic structure of the responder chain, a Cocoa mechanism for cooperative event handling. The **responder chain** is a linked series of responder objects in an application. For a typical event, the chain starts with the first responder; if this responder object cannot handle the event, it passes it to the next responder in the chain. The message travels up the chain—toward higher-level responder objects such as the window, the application, and the application's delegate—until the event is handled. If the event isn't handled, it is discarded.

The responder object that handles the event tends to set in motion a series of programmatic actions that result in the application redrawing all or a portion of its user interface (as well as other possible outcomes, such as the playing of a sound). For example, the first responder could be a control object (that is, an object inheriting from `UIControl`), and a control object handles an event by sending an action message to an object controlling a view. This controller object then asks the view to redraw or otherwise visually adjust a region of itself.

A central paradigm of UIKit is the view hierarchy, which underpins both the responder chain and the way in which an application draws its views. A **view hierarchy** is a hierarchical arrangement of views in a window, with a `UIWindow` object (itself a subclass of `UIView`) at the root of the hierarchy. Every view has a to-one relationship to its superview—the view above it in the hierarchy—and it also has a potentially to-many relationship to its subviews. Visually, the relationship between a view and its subviews is enclosure: a view encloses or contains its subviews, and they are positioned and sized within the coordinate system of their superview. This hierarchical structure for views allows you to construct complex views from an assortment of smaller views, and it is also the basis for the drawing order of a window's views (generally, from superview to subview).

After the application handles an event, the application object fetches the next event from the event queue and the cycle begins again.

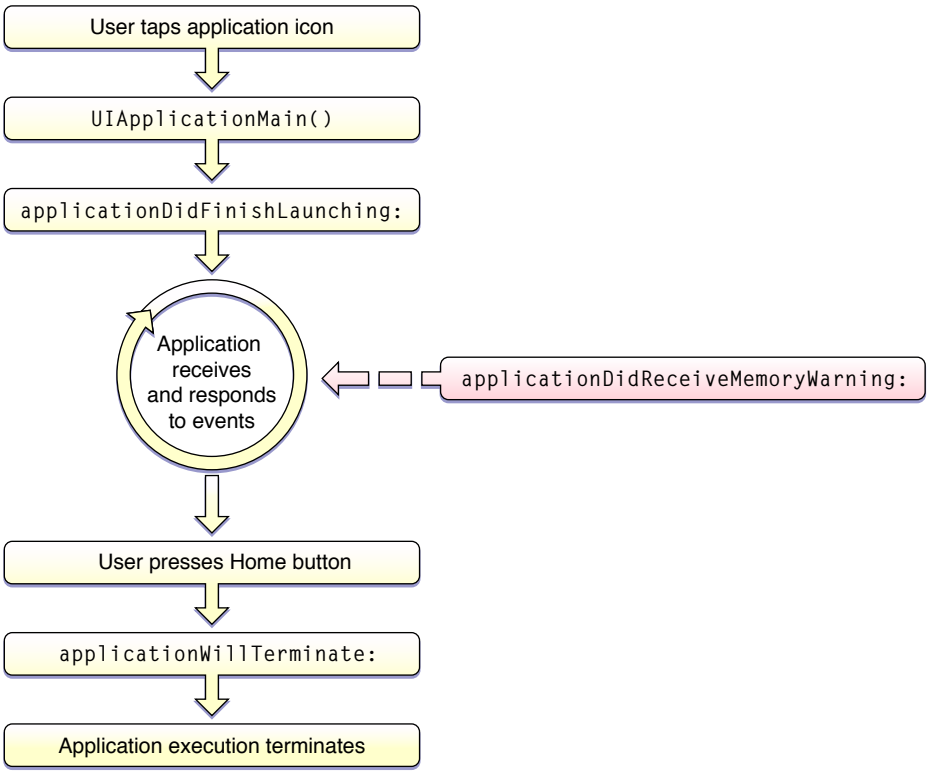
Note: "Event Handling" (page 139) discusses event objects, touch objects, responder objects, and the responder chain in more detail. "Windows and Views" (page 109) provides complete descriptions of the the view hierarchy, view management, and the drawing cycle in an iPhone application.

The Application Life Cycle

You can think of an application during a runtime session as having a life cycle. Most of the time it runs in its event and drawing cycle, getting events and responding to them in some way. But external events can disrupt this processing at any time. For example, the user presses the Home button or, on the iPhone, the device receives a phone call. The system responds to such an event by launching the application that can handle the event. Because only one iPhone application can be running at any given moment, the application that is currently running must quit.

Figure 5-3 depicts the life cycle of an iPhone application. Note that it does not show all the possible notifications of external events an application might receive, only the more important ones.

Figure 5-3 Application life cycle



An iPhone application goes through several stages from the moment it is launched to the moment it ceases execution.

1. The application launches.

   Figure 5-3 (page 91) shows the user tapping an application icon in the Home screen. Although this is the most common way of launching an application, other external events might be the cause. For example, the Safari application is launched when another application passes it a web site's URL for handling.

2. The system calls the application's `main` entry point, which calls the `UIApplicationMain` function. The function instantiates the singleton application object (and the delegate object if a class name is specified for it) and loads the application's main nib file, the name of which is obtained from the `Info.plist` file.

   The `UIApplicationMain` function also starts the application's main event loop.

3. The application sends `applicationDidFinishLaunching:` to its delegate, which does the following:

   - It loads and restores application state and any data that it stored during the previous session of execution. See "Application Design Guidelines" (page 69) for a discussion of requirements and approaches for saving restoring application state and data.
   - It creates and lays out the views for its initial presentation, updating it with the appropriate bits of state and model data.

While the application delegate is doing this, the system loads the launch image file from the application bundle and temporarily displays it. See "Application Icon and Launch Images" (page 102) for information about the launch image.

4. The application receives events and responds to them, often by redrawing or otherwise altering its user interface.

   While the application is running, the application delegate may receive an `applicationDidReceiveMemoryWarning:` message. It should respond to this message by releasing objects or freeing resources the application doesn't need immediately and can recreate or reload, if necessary. If the delegate does not respond to this message, or does not free sufficient memory, the application may be terminated.

5. User presses the Home button.

   Pressing the Home button is the most common case, but (as noted earlier) other external factors may result in an application being asked to terminate execution.

6. The application sends `applicationWillTerminate:` to its delegate. In this method, the application should save current application state and data to storage. It should also delete any temporary files it has created.

7. The application terminates its execution.

Application Interruptions

In addition to the user pressing the Home button, there are other ways for the system to interrupt your application. An application can be interrupted by an incoming phone call, an SMS message, a calendar alert, or by the user pressing the Sleep button on a device. Unlike pressing the Home button, however, these interruptions may be only temporary. If the user decides to take a call or reply to an SMS message, however, doing so causes the termination of your application.

The following steps describe what happens when a phone call arrives. The same set of steps are also followed for the arrival of SMS messages and calendar alerts, with the only difference being the type of information presented to the user.

1. The system detects an incoming phone call.

2. The system calls your application delegate's `applicationWillResignActive:` method. The system also disables the delivery of touch events to your application.

   You can use this method to disable timers, throttle back your OpenGL frame rates (if using OpenGL), and generally put your application into a sleep state. For example, if you are a game developer, you would use this notification to pause the game. While in the sleep state, your application should not do any significant work.

3. The system displays an alert panel with information about the phone call. The system also prompts the user to accept or ignore the call.

4. If the user ignores the call, the system calls your application delegate's `applicationDidBecomeActive:` method.

At this point, your application can reenable timers and other timer-related activities and begin running again. The system also resumes the delivery of touch events to your application.

5. If the user takes the call, the system calls your application delegate's `applicationWillTerminate:` method. Your application should terminate as usual, saving any needed contextual information to return the user to the same place in your application upon your next launch.

Depending on what the user does while answering a phone call, the system may launch your application again upon completion of the interruption. For example, if the user simply takes the call and then hangs up, the system relaunches your application. If while on the call, the user goes back to the Home screen and then launches another application, the system does not relaunch your application.

Important: When the user takes a call and then launches an application while on the call, the height of the status bar grows to reflect the fact that the user is on a call. Similarly, when the user ends the call, the status bar height shrinks back to its regular size. Your own applications should be prepared for these changes in the status bar height and adjust their content area accordingly. View controllers handle this behavior for you automatically. If you lay out our user interface programmatically, however, you need to take the status bar height into account when laying out your views and implement the `layoutSubviews` method to handle dynamic layout changes.

If the user presses the Sleep/Wake button on the phone while running your application, the system calls your application delegate's `applicationWillResignActive:` method but does not prompt the user for any information. Your application goes to sleep and should not perform any activity at this time. Upon waking the device, the system similarly calls your application delegate's `applicationDidBecomeActive:` method. At that point, you can restart any timers and processing you were doing.

The Application Bundle

When you build an application, Xcode packages it as a bundle. A **bundle** is a directory in the file system that groups related resources together in one place. The application bundle contains the application executable and any resources used by the application (for instance, the application icon, other images, and localized content). Table 5-1 lists the contents of a typical application bundle, which for demonstration purposes here is called `MyApp`). This example is for illustrative purposes only. Some of the files listed in this table may not appear in your own application bundles.

Table 5-1 A typical application bundle

File	Description
`MyApp`	The executable file containing your application's code. The name of this file is the same as your application name minus the `.app` extension. This file is required.
`Settings.bundle`	The settings bundle is a file package that you use to add application preferences to the Settings application. This bundle contains property lists and other resource files to configure and display your preferences. See "The Settings Bundle" (page 102) for more information.

File	Description
`Icon.png`	The 57 x 57 pixel icon used to represent your application on the device home screen. This icon should not contain any glossy effects. The system adds those effects for you automatically. This file is required. For information about this image file, see "Application Icon and Launch Images" (page 102).
`Icon-Settings.png`	The 29 x 29 pixel icon used to represent your application in the Settings application. If your application includes a settings bundle, this icon is displayed next to your application name in the Settings application. If you do not specify this icon file, the `Icon.png` file is scaled and used instead. For information about this image file, see "The Settings Bundle" (page 102).
`MainWindow.nib`	The application's main nib file contains the default interface objects to load at application launch time. Typically, this nib file contains the application's main window object and an instance of the application delegate object. Other interface objects are then either loaded from additional nib files or created programmatically by the application. (The name of the main nib file can be changed by assigning a different value to the `NSMainNibFile` key in the `Info.plist` file. See "The Information Property List" (page 95) for further information.)
`Default.png`	The 480 x 320 pixel image to display when your application is launched. The system uses this file as a temporary background until your application loads its window and user interface. For information about this image file, see "Application Icon and Launch Images" (page 102).
`Info.plist`	Also known as the information property list, this file is a property list defining key values for the application, such as bundle ID, version number, and display name. See "The Information Property List" (page 95) for further information. This file is required.
`sun.png` (or other resource files)	Nonlocalized resources are placed at the top level of the bundle directory (`sun.png` represents a nonlocalized image file in the example). The application uses nonlocalized resources regardless of the language setting chosen by the user.

`en.lproj`
`fr.lproj`
`es.lproj`
other language-specific project directories
	Localized resources are placed in subdirectories with an ISO language abbreviation for a name plus an `.lproj` suffix. (For example, the `en.lproj`, `fr.lproj`, and `es.lproj` directories contain resources localized for English, French, and Spanish.) For more information, see "Internationalizing Applications" (page 105).

An application might not have nonlocalized resources, but it should be internationalized and have a *language*`.lproj` folder for each language it supports.

You use the methods of the `NSBundle` class or functions of the `CFBundle` opaque type to obtain paths to localized and nonlocalized image and sound resources stored in the application bundle. For example, to get a path to the image file `sun.png` (shown in "Application Interruptions" (page 92)) and create an image file from it would require two lines of Objective-C code:

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

Calling the `mainBundle` class method returns an object representing the application bundle. See *Resource Programming Guide* for information on loading resources.

Application Configuration

The application bundle contains several components that you must configure for your application to execute successfully when launched. You must specify a range of properties that both identify the application to the system and identify the preferences users can make. You must also provide images that the system looks for when it presents your application to users.

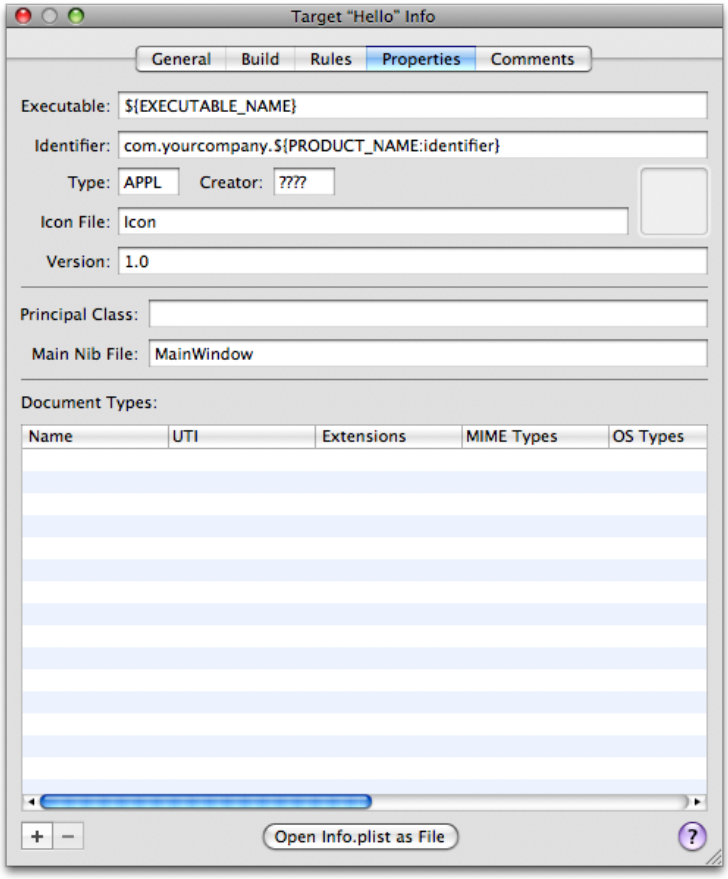
The Information Property List

The information property list is a file named `Info.plist` that is included with every iPhone application project created by Xcode. It is a property list whose key-value pairs specify essential runtime-configuration information for the application. The elements of the information property list are organized in a hierarchy in which each node is an entity such as an array, dictionary, string, or other scalar type.

Note: To learn more about information property lists, see *Runtime Configuration Guidelines*.

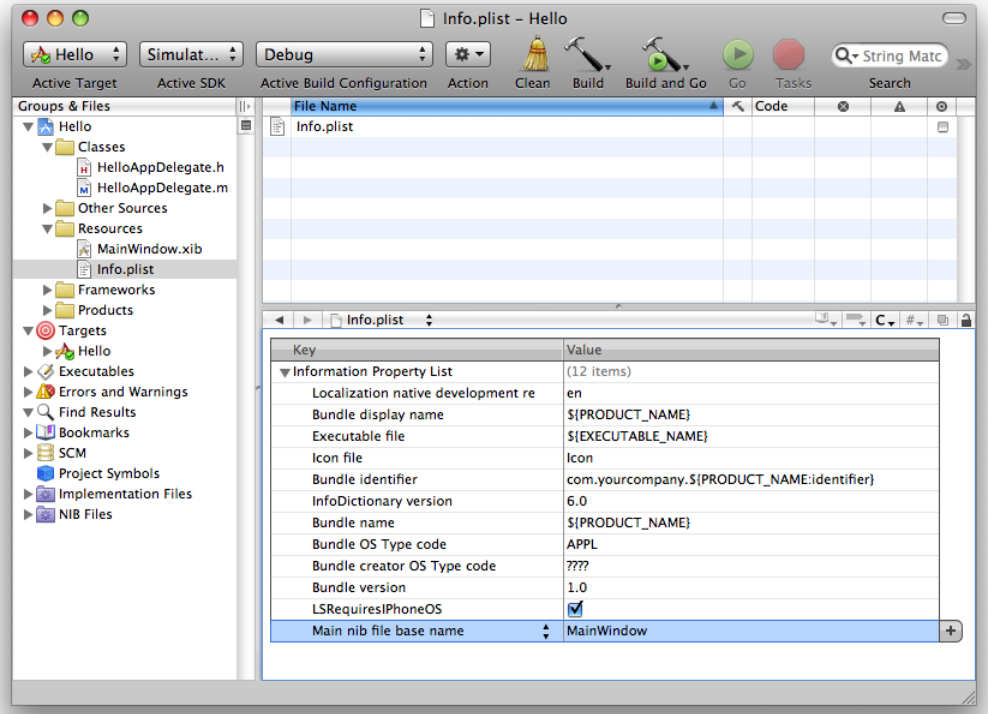
In Xcode, you can access the information property list by choosing "Edit Active Target *TargetName*" from the Project menu. Then in the target's Info window, click the Properties control. Xcode displays a pane of information similar to the example in Figure 5-4.

Figure 5-4 The Properties pane of a target's Info window



The Properties pane shows you some, but not all, of the properties of the application bundle. When you click the "Open Info.plist as File" button, or when you select the Info.plist file in your Xcode project, Xcode displays a property list editor window similar to the one in Figure 5-5. You can use this window to edit the property values and add new key-value pairs.

Figure 5-5 Information property list editor



Xcode automatically sets the value of some of these properties, but others you need to set explicitly. Table 5-2 lists some of the important keys you might want to include in your application's `Info.plist` file. For a complete list of properties you can include in this file, see *Runtime Configuration Guidelines*.

Table 5-2 Important keys in the `Info.plist` file

Key	Value
`CFBundleDisplayName`	The name to display underneath the application icon. This value should be localized for all supported languages.
`CFBundleIdentifier`	An identifier string that specifies the application type of the bundle. This string should be a uniform type identifier (UTI). For example, if your company name is Ajax and the application is named Hello, the bundle identifier would be com.Ajax.Hello. The bundle identifier is used in validating the application signature.
`CFBundleURLTypes`	An array of URL types that the application can handle. Each URL type is a dictionary that defines the schemes (for example , such as "http" or "mailto") that the application can handle. This property allows applications to register custom URL schemes.
`CFBundleVersion`	A string that specifies the build version number of the bundle. This value is a monotonically increased string, comprised of one or more period-separated integers. This value cannot be localized.

Key	Value
LSRequiresIPhoneOS	A Boolean value that indicates whether the bundle can run on iPhone OS only. Xcode adds this key automatically and sets its value to true. You should not change the value of this key.
NSMainNibFile	A string that identifies the name of the application's main nib file. If you want to use a nib file other than the default one created for your project, associate the name of that nib file with this key. The name of the nib file should not include the .nib filename extension.
UIStatusBarStyle	A string that identifies the style of the status bar as the application launches. This value is based on the Status Bar Style Constants constants declared in UIApplication.h header file. The default style is UIBarStyleDefault. The application can change this initial status-bar style when it finishes launching.
UIStatusBarHidden	A Boolean value that determines whether the status bar is initially hidden when the application launches. Set it to true to hide the status bar. The default value is false.
UIInterface-Orientation	A string that identifies the initial orientation of the application's user interface. This value is based on the Interface Orientation Constants constants declared in the UIApplication.h header file. The default style is UIInterfaceOrientationPortrait. For more information on launching your application in landscape mode, see "Launching in Landscape Mode" (page 102).
UIPrerenderedIcon	A Boolean value that indicates whether the application icon already includes gloss and bevel effects. This property is false by default. Set it to true if you do not want the system to add these effects to your artwork.
UIRequiresPersistent-WiFi	A Boolean value that notifies the system that the application uses the WiFi network for communication. Applications that use WiFi for any period of time must set this key to true; otherwise, after 30 minutes, the device shuts down WiFi connections to save power. Setting this flag also lets the system know that it should display the network selection dialog when WiFi is available but not currently being used. The default value is false.

Properties with string values that are displayed in the user interface should be localized. Specifically, the string value in Info.plist should be a key to a localized string in the InfoPlist.strings file of a language-localized subdirectory. See "Internationalizing Applications" (page 105) for details.

Custom URL Schemes and Interapplication Communication

You can register URL types for your application that include custom URL schemes. A custom URL scheme is a mechanism through which third-party applications can interact with each other and with the system. Through a custom URL scheme, an application can make its services available to other applications.

Once you have registered a scheme, another application can send an `openURL:` message to its singleton `UIApplication` object, passing in an `NSURL` object that represents a URL incorporating your registered scheme. The system then launches your application, and your application delegate can handle the URL resource in its `application:handleOpenURL:` method (which is discussed in Table 5-3 (page 99)). The following code fragment illustrates how one application can request the services of another application ("todolist" in this example is a hypothetical custom scheme registered by an application):

```
NSURL *myURL = [NSURL URLWithString:@"todolist://www.acme.com?Quarterly
Report#200806231300"];
[[UIApplication sharedApplication] openURL:myURL];
```

Important: If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided application that handles a URL with that scheme (for example, "mailto") is launched instead of your application. If a URL type registered by your application includes a scheme that conflicts with a scheme registered by another third-party application, the application that launches for a URL with that scheme is undefined.

Registering Custom URL Schemes

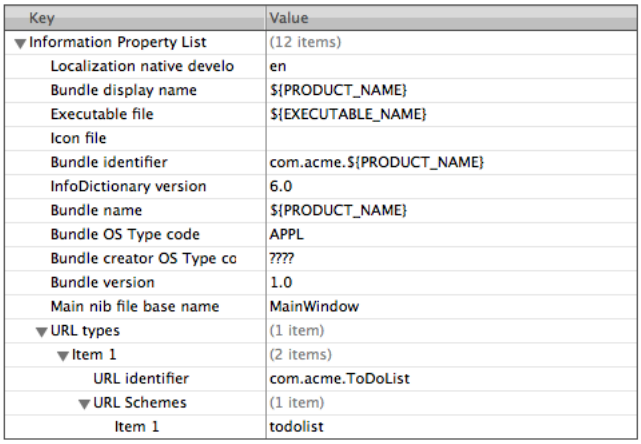
To register a URL type for your application, you must specify the subproperties of the `CFBundleURLTypes` property, which was introduced in "The Information Property List" (page 95). The `CFBundleURLTypes` property is an array of dictionaries in the application's `Info.plist` file, with each dictionary defining a URL type the application supports. Table 5-3 describes the keys and values of a `CFBundleURLTypes` dictionary.

Table 5-3 Keys and values of the `CFBundleURLTypes` property

Key	Value
`CFBundleURLName`	A string that is the abstract name for the URL type. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, "com.acme.Foo". The URL-type name provided here is used as a key to a localized string in the `InfoPlist.strings` file in a language-localized bundle subdirectory. The localized string is the human-readable name of the URL type in a given language.
`CFBundleURLSchemes`	An array of URL schemes for URLs belonging to this URL type. Each scheme is a string. URLs belonging to a given URL type are characterized by their scheme components.

Figure 5-6 shows a definition of the `CFBundleURLTypes` property ("URL types") using the built-in Xcode editor for `Info.plist` files. "URL identifier" is the editor's label for the `CFBundleURLName` property and "URL Schemes" is the label for the `CFBundleURLSchemes` property.

Figure 5-6 Defining a custom URL scheme in the Info.plist file



Once you have registered a URL type with a custom scheme by defining the `CFBundleURLTypes` property, you can test the scheme in the following way:

1. Build, install, and run your application.
2. Go the the Home screen and launch Safari. (In the iPhone simulator you can go to the Home screen by selecting Hardware > Home from the menu.)
3. Type a URL that uses your custom scheme in the address bar of Safari.
4. Verify that your application launches and the application delegate is sent a `application:handleOpenURL:` message.

Handling URL Requests

The delegate of an application handles URL requests routed to the application by implementing the `application:handleOpenURL:` method. You especially need to implement this method if you have registered custom URL schemes for your application.

A URL request based on a custom scheme assumes a kind of protocol understood by those applications requesting the services of your application. The URL contains information of some kind that the scheme-registering application is expected to process or respond to in some way. Objects of the `NSURL` class, which are passed into the `application:handleOpenURL:` method, represent URLs in the Cocoa Touch framework. `NSURL` conforms to the RFC 1808 specification; it includes methods that return the various parts of a URL as defined by RFC 1808, including user, password, query, fragment, and parameter string. The "protocol" for your custom scheme can use these URL parts for conveying various kinds of information.

In the example implementation of `application:handleOpenURL:` shown in Listing 5-2, the passed-in URL object conveys application-specific information in its query and fragment parts. The delegate extracts this information—in this case, the name of a to-do task and the date the task is due—and with it creates a model object of the application.

Listing 5-2 Handling a URL request based on a custom scheme

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url {
    if ([[url scheme] isEqualToString:@"todolist"]) {
        ToDoItem *item = [[ToDoItem alloc] init];
        NSString *taskName = [url query];
        if (!taskName || ![self isValidTaskString:taskName]) { // must have a
task name
            [item release];
            return NO;
        }
        taskName = [taskName
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

        item.toDoTask = taskName;
        NSString *dateString = [url fragment];
        if (!dateString || [dateString isEqualToString:@"today"]) {
            item.dateDue = [NSDate date];
        } else {
            if (![self isValidDateString:dateString]) {
                [item release];
                return NO;
            }
            // format: yyyymmddhhmm (24-hour clock)
            NSString *curStr = [dateString substringWithRange:NSMakeRange(0,
4)];
            NSInteger yeardigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(4, 2)];
            NSInteger monthdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(6, 2)];
            NSInteger daydigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(8, 2)];
            NSInteger hourdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(10, 2)];
            NSInteger minutedigit = [curStr integerValue];

            NSDateComponents *dateComps = [[NSDateComponents alloc] init];
            [dateComps setYear:yeardigit];
            [dateComps setMonth:monthdigit];
            [dateComps setDay:daydigit];
            [dateComps setHour:hourdigit];
            [dateComps setMinute:minutedigit];
            NSCalendar *calendar = [NSCalendar currentCalendar];
            NSDate *itemDate = [calendar dateFromComponents:dateComps];
            if (!itemDate) {
                [dateComps release];
                return NO;
            }
            item.dateDue = itemDate;
            [dateComps release];
        }

        [(NSMutableArray *)self.list addObject:item];
        [item release];
        return YES;
    }
    return NO;
}
```

Be sure to validate the input you get from URLs passed to your application; see Validating Input in *Secure Coding Guide* to find out how to avoid problems related to URL handling. To learn about URL schemes defined by Apple, see "Apple Applications URL Schemes" (page 207).

Application Icon and Launch Images

The file for the icon displayed in the user's Home screen has the default name of `Icon.png` (although the `CFBundleIconFile` property in the `Info.plist` file lets you rename it). It should be a PNG image file located in the top level of the application bundle. The application icon should be a 57 x 57 pixel image without any shine or round beveling effects. Typically, the system applies these effects to the icon before displaying it. You can override that behavior, however, by including the `UIPrerenderedIcon` key in your application's `Info.plist` file; for more information, see Table 5-2 (page 97).

The file for the application's launch image is named `Default.png`. This image should closely resemble the application's initial user interface; the system displays the launch image before an application is ready to display its user interface, giving the impression of a quick launch. The launch image should also be a PNG image file, located in the lop level of the application bundle. If the application is launched through a URL, the system looks for a launch image named `Default-`*scheme*`.png`, where *scheme* is the scheme of the URL. If that file is not present, it chooses `Default.png` instead.

To add an image file to a project in Xcode, choose Add to Project from the Project menu, locate the file in the browser, and click Add.

The Settings Bundle

The Settings bundle is named `Settings.bundle` and is located in the top level of the application bundle. The **Settings bundle** is an opaque directory that contains information that the Settings application uses when it displays your application's preferences to users. The Settings bundle contains one or more schema files that provide detailed information about your application's preferences. It may also include other support files needed to display your preferences, such as images or localized strings. When the user supplies values for your application's preferences, the Settings application updates the preferences database. At runtime, your application can retrieve those values using the `NSUserDefaults` or `CFPreferences` APIs.

The file for the icon displayed for your custom application preferences in the Settings applications has the name `Icon-Settings.png`. This file should be a 29 x 29 pixel PNG image file located in the top level of the application bundle. If your application has a settings bundle, but you do not provide this file, the Settings application uses the application icon (`Icon.png`) by default.

For more information on application preferences and the Settings bundle, see "Application Preferences" (page 193).

Launching in Landscape Mode

Applications in iPhone OS normally launch in portrait mode to match the orientation of the Home screen. If you have an application that you want to launch in landscape mode, however, you must perform several steps to make it launch in that orientation.

- Set the `UIInterfaceOrientation` key to an appropriate value in your application's `Info.plist` file. You can set the value for this key to `UIInterfaceOrientationLandscapeLeft` or `UIInterfaceOrientationLandscapeRight`.
- Lay out your content in landscape mode or make sure that your content's autoresizing options are set correctly.
- When your view is loaded, you must manually reorient it to landscape mode.

Important: These steps are only needed if you intend to run your application in landscape mode all the time. If you are creating an application that can toggle between landscape and portrait modes, you should launch your application in portrait mode and then use the built-in view controller support to rotate your interface as needed.

The `UIInterfaceOrientation` key provides a hint to iPhone OS that it should set the orientation of the application status bar (if one is displayed) to the specified orientation at launch time. This is equivalent to calling the `setStatusBarOrientation:animated:` method of `UIApplication` early in the execution of your `applicationDidFinishLaunching:` method. Setting this orientation affects only the status bar orientation, however, and does not affect the orientation of your views or default launch image.

Because the `UIInterfaceOrientation` key does not alter the orientation of your content, you must present that content in the desired orientation when it is loaded. If you are using Interface Builder to lay out your views, you can reorient the design surface for landscape-right mode and lay things out graphically there. You can also lay out your views in portrait mode and let the view autoresizing behavior adjust the layout to landscape mode for you. When rotating the design surface from portrait to landscape mode, Interface Builder respects the autoresizing behaviors set for each view and adjusts their position on the design surface. You can therefore test the behavior of your views inside Interface Builder before loading them into your application.

Regardless of how you orient the design surface in Interface Builder, it is important to remember that nib files themselves have no concept of interface orientation. Views are blank canvases that can be used for many purposes, such as presenting content in a scroll view. Therefore, the size of a view does not change its orientation at load time; it is always loaded in portrait mode. It is up to you to rotate your views to the appropriate orientation after they are loaded. When you do this depends on how you construct your application. If you construct your application using view controllers, you can perform this operation in the `viewDidLoad` method of your view controller objects. If you are not using view controllers, you can perform it in the `applicationDidFinishLaunching:` method after creating your views or after loading them from a nib file.

Important: Interface orientation changes must always be performed on your views and not on your application's main window. The graphics system expects the window to be in portrait orientation at all times. Changing the transform of your window object could prevent your views from drawing their contents correctly.

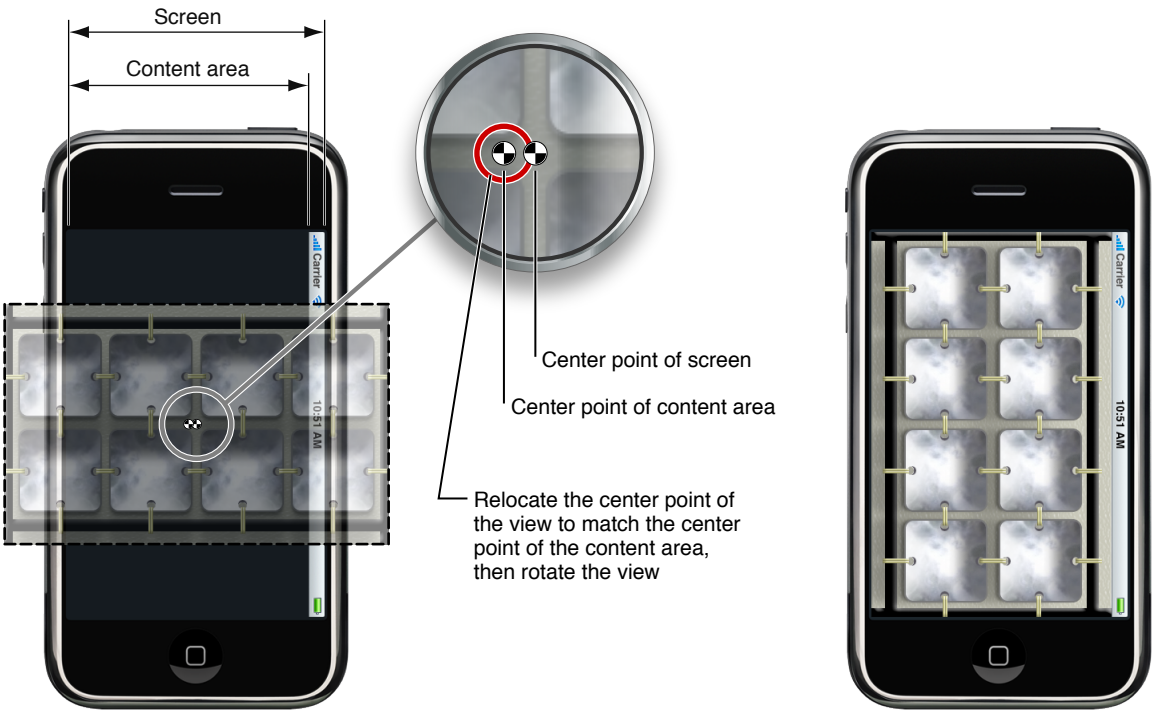
After your views are loaded into memory, you need to reorient them to landscape mode by doing the following:

1. Find the center point of your application's content area.
2. Rotate your views 90 degrees around that center point.

The exact center point of your application's content area can vary and is dependent on your interface design. Full-screen applications or applications that display a translucent status bar typically have a content area that matches the screen bounds. Applications that display an opaque status bar typically have a content area that does not include the area occupied by the status bar. This latter scenario is more common and is also a little more difficult to implement correctly. As a result, the rest of this section shows you how to perform a rotation on an interface that uses an opaque status bar.

When your landscape view is loaded from its nib file, its center point is initially set to match the center point of the screen, as shown in Figure 5-7. Because the view in the example does not occupy the area under the status bar, however, the initial center point is incorrect. Determining the correct center point requires computing the actual rectangle that the view will fill and determining the center of it. This is where the frame of the status bar can help. The x-origin point and height of the status bar correspond to the width and height of the rectangle the view will fill. The origin of this rectangle is fixed at (0, 0) because it corresponds to the origin of the window's coordinate system. From the resulting rectangle, you can get the center point, assign it to the `center` property of your view, and then apply the rotation transform.

Figure 5-7 Coordinates used when laying out in landscape mode



Listing 5-3 shows the `viewDidLoad` method of a controller object whose views are oriented for landscape-right mode at launch. This method uses the status bar frame rectangle to compute the visible content area and then get the center of that rectangle. It then shifts the center of the view to that location and rotates the view into the proper orientation.

Listing 5-3 Reorienting a view to landscape mode

```
- (void)viewDidLoad {
```

```
    UIInterfaceOrientation orientation = [[UIApplication sharedApplication]
statusBarOrientation];

    [super viewDidLoad];

    if (orientation == UIInterfaceOrientationLandscapeRight) {
        CGAffineTransform transform = self.view.transform;

       // Use the status bar frame to determine the center point of the window's
 content area.
        CGRect statusBarFrame = [[UIApplication sharedApplication]
statusBarFrame];
        CGRect bounds = CGRectMake(0, 0, statusBarFrame.size.height,
statusBarFrame.origin.x);
       CGPoint center = CGPointMake(bounds.size.height / 2.0, bounds.size.width
 / 2.0);

        // Set the center point of the view to the center point of the window's
 content area.
        self.view.center = center;

        // Rotate the view 90 degrees around its new center point.
        transform = CGAffineTransformRotate(transform, (M_PI / 2.0));
        self.view.transform = transform;
    }
}
```

Internationalizing Applications

Ideally, the text, images, and other content that iPhone applications display to users should be localized for multiple languages. The text that an alert dialog displays, for example, should be in the preferred language of the user. The process of preparing a project for content localized for particular languages is called **internationalization**. Project components that are candidates for localization include:

- Code-generated text, including locale-specific aspects of date, time, and number formatting
- Static text—for example, an HTML file loaded into a web view for displaying application help
- Icons and other images when those images either contain text or have some culture-specific meaning
- Sound files containing spoken language
- Nib files

Using the Settings application, users select the language they want to see in applications' user interfaces from the Language preferences view (see Figure 5-8) They get to this view from the International group of settings, accessed from General settings.

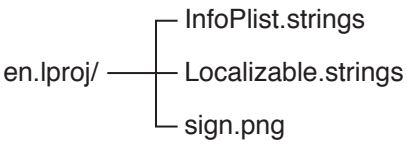
Figure 5-8 The Language preference view

The chosen language is associated with a subdirectory of the bundle. The subdirectory name has two components: an ISO 639-1 language code and a `.lproj` suffix. The language code may also modified with a particular region by appending (after an underscore) an ISO 3166-1 region designator. For example, to designate resources localized to English as spoken in the United States, the bundle would be named `en_US.lproj`. By convention, these language-localized subdirectories are called `lproj` folders.

Note: You may use ISO 639-2 language codes instead of those defined by IOS 639-1. See "Language and Locale Designations" in *Internationalization Programming Topics* for descriptions of language and region codes.

An `lproj` folder contains all the localized content for a given language and, possibly, a given region. You use the facilities of the `NSBundle` class or the `CFBundle` opaque type to locate (in one of the application's `lproj` folders) resources localized for the currently selected language. Figure 5-9 gives an example of such a directory containing content localized for English (`en`).

Figure 5-9 The contents of a language-localized subdirectory



This subdirectory example has the following items:

- The `InfoPlist.strings` file contains strings assigned as localized values of certain properties in the project's `Info.plist` file (such as `CFBundleDisplayName`). For example, the `CFBundleDisplayName` key for an application named Battleship in the English version would have this entry in `InfoPlist.strings` in the `fr.lproj` subdirectory:

  ```
  CFBundleDisplayName = "Cuirassé";
  ```

- The `Localizable.strings` file contains localized versions of strings generated by application code.
- The `sign.png` file in this example is a file containing a localized image.

To internationalize strings in your code for localization, use the `NSLocalizedString` macro in place of the string. This macro has the following declaration:

```
NSString *NSLocalizedString(NSString *key, NSString *comment);
```

The first parameter is a unique key to a localized string in a `Localizable.strings` file in a given `lproj` folder. The second parameter is comment to assist in translation. For example, suppose you are setting the content of a label (`UILabel` object) in your user interface. The following code would internationalize the label's text:

```
label.text = NSLocalizedString(@"City", @"Label for City text field");
```

You can then create a `Localizable.strings` file for a given language and add it to the proper `lproj` folder. For the above key, this file would have an entry similar to the following:

```
"City" = "Ville";
```

Note: Alternatively, you can insert `NSLocalizedString` calls in your code where appropriate and then run the `genstrings` command-line tool. This tool generates a `Localizable.strings` template that includes the key and comment for each string requiring translation. For further information about `genstrings`, see the `genstrings(1)` man page.

To find out more about internationalization, see *Internationalization Programming Topics*.

Windows and Views

Windows and views are the visual components you use to construct the interface of your iPhone application. Windows provide the background platform for displaying content but views do most of the work of drawing that content and responding to user interactions. Although this chapter covers the concepts associated with both windows and views, it focuses more on views because of their importance to the system.

Because views play such a vital role in iPhone applications, there is no way to cover every aspect of them in a single chapter. This chapter focuses on the basic properties of windows and views, their relationships to each other, and how you create and manipulate them in your application. This chapter does not cover how views respond to touch events or draw custom content. For more information about those subjects, see "Event Handling" (page 139) and "Graphics and Drawing" (page 151) respectively.

What Are Windows and Views?

Like Mac OS X, iPhone OS uses windows and views to present graphical content on the screen. Although there are many similarities between the window and view objects on both platforms, the roles played by both windows and views are slightly different on each platform.

The Role of UIWindow

Although windows play an important role in the development of Mac OS X applications, in iPhone applications, that role is reduced significantly. A typical iPhone application has only one **window**, represented by an instance of the `UIWindow` class. Your application creates this window at launch time (or loads it from a nib file), adds one or more views to it, and displays it. After that, you rarely need to refer to the window object again.

A window object has no visual adornments such as a close box or title bar and cannot be closed or manipulated directly by the user. All manipulations to a window occur through its programmatic interfaces. The application also uses the window to facilitate the delivery of events to your application. For example, the window object keeps track of its current first responder object and dispatches events to it when asked to do so by the `UIApplication` object.

One thing that experienced Mac OS X developers may find unusual about the `UIWindow` class is its inheritance. In Mac OS X, the parent class of `NSWindow` is `NSResponder`. In iPhone OS, the parent class of `UIWindow` is `UIView`. Thus, in iPhone OS, a window is also a view object. Despite its parentage, you typically treat windows in iPhone OS the same as you would in Mac OS X. You typically do not manipulate the view-related properties of a `UIWindow` object directly.

When creating your application window, you should always set its initial frame size to fill the entire screen. If you load your window from a nib file, Interface Builder does not permit you to create a window smaller than the screen size. If you create your window programmatically, however, you must specifically pass in the desired frame rectangle at creation time. There is no reason to pass in any rectangle other than the screen rectangle, which you can get from the `UIScreen` object as shown here:

```
UIWindow* aWindow = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]] autorelease];
```

Although iPhone OS supports layering windows on top of each other, your application should never create more than one window. The system itself uses additional windows to display the system status bar, important alerts, and other types of messages on top of your application's windows. If you want to display alerts on top of your content, use the alert views provided by UIKit rather than creating additional windows.

The Role of UIView

A **view**, an instance of the `UIView` class, defines a rectangular area on the screen. In iPhone applications, views play a key role in both presenting your interface and responding to interactions with that interface. Each view object has the responsibility of rendering content within its rectangular area and for responding to touch events in that area. This dual behavior means that views are the primary mechanism for interacting with the user in your application. In a Model-View-Controller application, view objects obviously are the View portion of the application.

In addition to displaying its own contents and handling events, a view may also manage one or more subviews. A **subview** is simply a view object embedded inside the frame of the original view object, which is referred to as the parent view or **superview**. Views arranged in this manner form what is known as a **view hierarchy** and may contain any number of views. Views can also be nested at arbitrarily deep levels by adding subviews to subviews. The organization of views inside the view hierarchy controls what appears on screen, as each subview is displayed on top of its parent view. The organization also controls how the views react to events and changes. Each parent view is responsible for managing its direct subviews, by adjusting their position and size as needed and even responding to events that its subviews do not handle.

Because view objects are the main way your application interacts with the user, they have a number of responsibilities. Among the responsibilities of views are the following:

- Drawing and animation
  - Views draw content in their rectangular area.
  - Some view properties can be animated to new values.
- Layout and subview management
  - Views manage a list of subviews.

- Views define their own resizing behaviors in relation to their parent view.
- Views can manually change the size and position of their subviews as needed.
- Views can convert points in their coordinate system to the coordinate systems of other views or the window.

- Event handling
  - Views receive touch events.
  - Views participate in the responder chain.

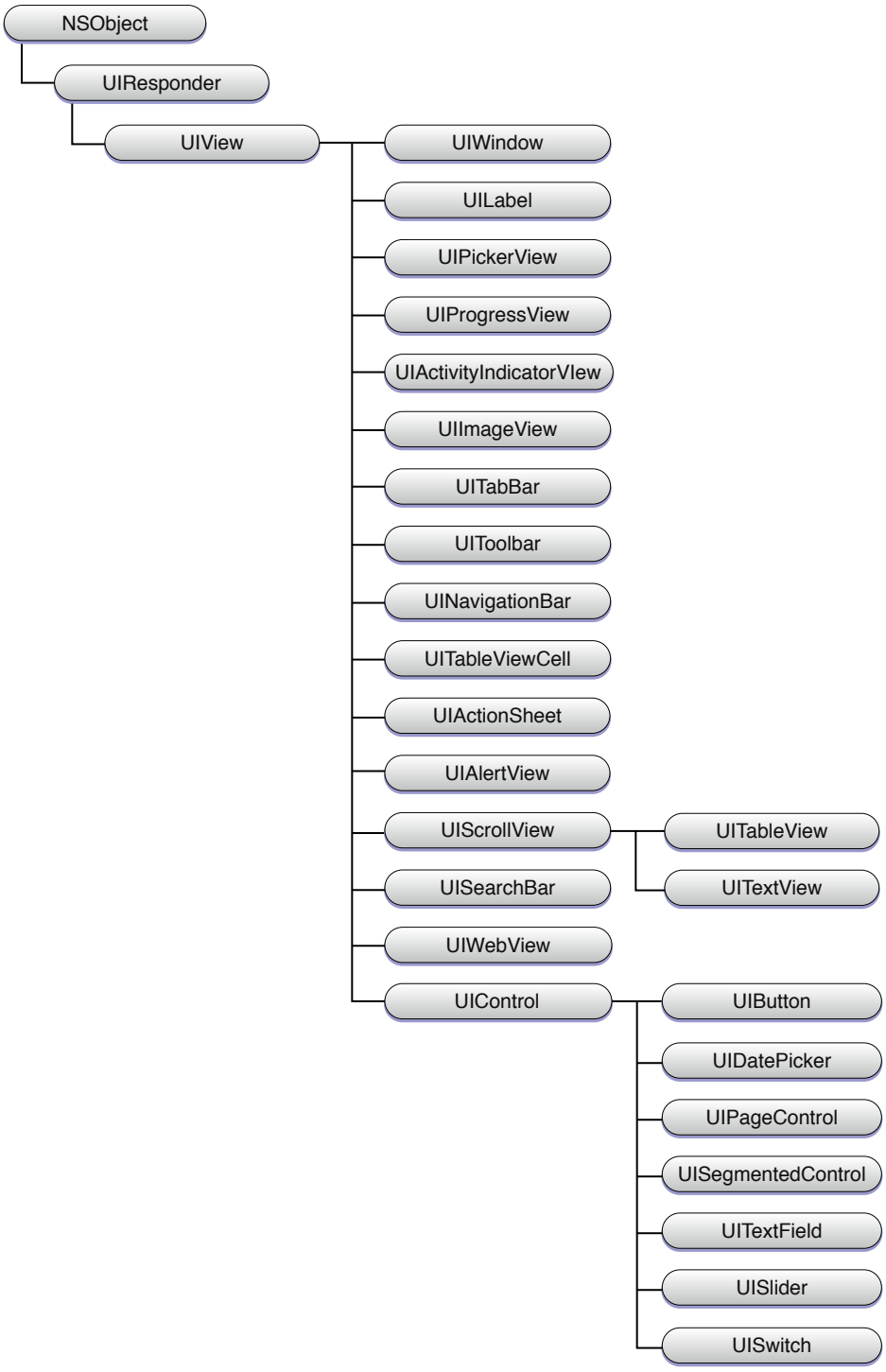
In iPhone applications, views work closely with view controllers to manage several aspects of the views behavior. View controllers handle the loading and unloading of views, interface rotations caused by the user physically rotating the device, and interactions with the high-level navigation objects used to construct complex user interfaces. For more information, see "The Role of View Controllers" (page 114).

Most of this chapter is dedicated to explaining these responsibilities and showing you how to tie your own custom code into the existing `UIView` behaviors.

UIKit View Classes

The `UIView` class defines the basic properties of a view but does not define any specific visual representation for views. Instead, UIKit uses subclasses to define the specific appearance and behavior for standard system elements such as text fields, buttons, and toolbars. Figure 6-1 shows the class hierarchy diagram for all of the views in UIKit. With the exception of the `UIView` and `UIControl` classes, most of the views in this hierarchy are designed to be used as-is or in conjunction with a delegate object.

Figure 6-1 View class hierarchy



The preceding set of views can be broken down into the following broad categories:

- Containers

Container views enhance the function of other views, or provide additional visual separation of the content. For example, the `UIScrollView` class is used to display views whose contents are too large to fit onscreen all at once. The `UITableView` class is a subclass of `UIScrollView` that manages lists of data. Because table rows are selectable, tables are commonly used for hierarchical navigation too—for example, to drill down into a hierarchy of objects.

- Controls

  The majority of a typical application's user interface is created using controls. A control is a special type of view that inherits from the `UIControl` superclass. Controls typically display a specific value and handle all of the user interactions required to modify that value. Controls also use standard system paradigms, such as target-action and delegation, to notify your application when user interactions occur. Controls include buttons, text fields, sliders, and switches.

- Display views

  Although controls and many other types of views provide interactive behavior, UIKit does provide some views for simply displaying information. These classes include `UIImageView`, `UILabel`, `UIProgressView`, and `UIActivityIndicatorView`.

- Text and web views

  Text and web views provide a more sophisticated way to display multiline text content in your application. The `UITextView` class supports the display and editing of multiple lines of text in a scrollable area. The `UIWebView` class provides a way to display HTML content, which lets you incorporate graphics and advanced text-formatting options and lay out your content in custom ways.

- Alert views and action sheets

  Alert views and action sheets are used to get the user's attention immediately. They present a message to the user along with one or more optional buttons that the user can use to respond to the message. Alert views and action sheets are similar in function but look and behave differently. The `UIAlertView` class displays a blue alert box that pops up on the screen; the `UIActionSheet` class displays a box that slides in from the bottom of the screen.

- Navigation views

  Tab bars and navigation bars work in conjunction with view controllers to provide tools for navigating from one screen of your user interface to another. You typically do not create `UITabBar` and `UINavigationBar` items directly but configure them through the appropriate controller interface or using Interface Builder instead.

  Unlike tab bars and navigation bars, you do create and manage toolbars in your user interface. A `UIToolbar` object displays buttons for commonly used commands along the bottom of the screen. The Safari, Mail, and Photos applications all use toolbars for portions of their user interfaces. Toolbars can be shown all the time or only as needed by the application. You can also display toolbars in conjunction with tab bars and navigation bars.

- The window

  A window provides a surface for drawing content and is the root container for all other views. There is typically only one window per application. For more information, see "The Role of UIWindow" (page 109).

In addition to views, UIKit provides view controllers to manage those objects. For more information, see "The Role of View Controllers" (page 114).

The Role of View Controllers

Applications running in iPhone OS have many options for organizing their content and presenting it to the user. An application with multiple screens worth of information presents multiple sets of views to display those screens. Managing those views behind the scenes is a view controller object.

A view controller object, an instance of the `UIViewController` class, provides the underlying logic for managing and displaying a screen's worth of content. The base view controller class provides a workflow for creating a set of views programmatically or loading them from a nib file. It also works with the operating system to flush those views from memory during low-memory situations and then reconstitute them if they are needed later. View controllers provide automatic support for some standard system behaviors, such as interface orientation changes caused by the user rotating the device and the temporary display of other views on top of the current content.

In addition to the base `UIViewController` class, UIKit includes more advanced subclasses for handling some of the sophisticated interface behaviors common to the platform. For example, navigation controllers manage the display of multiple hierarchical screens worth of content. Tab bar controllers let the user switch between different sets of screens, each of which represents a different operating mode for the application.

For information on how to use view controllers to manage the views in your user interface, see *View Controller Programming Guide for iPhone OS*.

View Architecture and Geometry

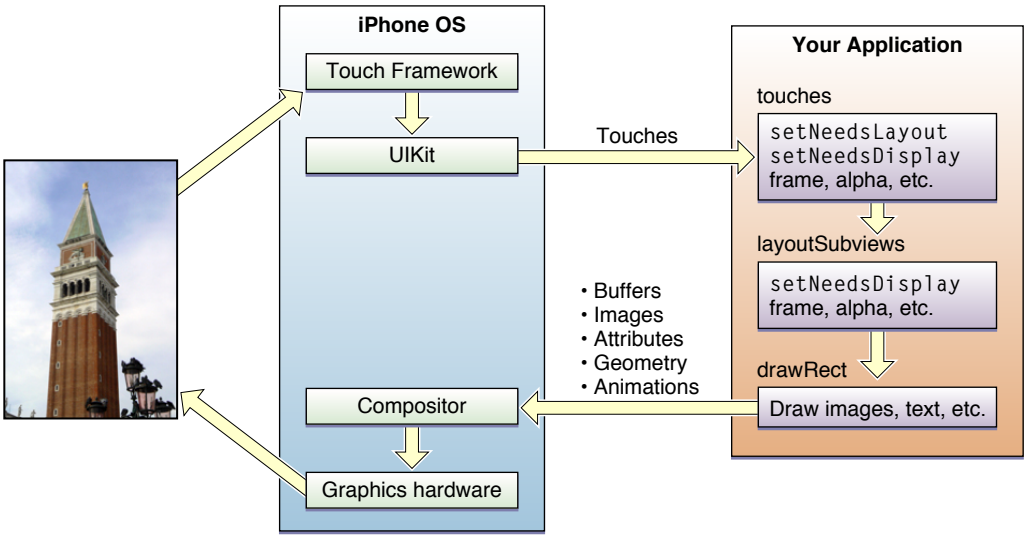
Because they are focal objects in iPhone applications, it is important to understand a little about how views interact with other parts of the system. The standard view classes in UIKit provide a considerable amount of behavior "for free" to your application. They also provide well-defined integration points where you can customize that behavior and do what you need to do for your application.

The following sections explain the standard behavior of views and call out the high-level integration points. For information about the integration points of specific classes, see the reference document for that class. You can get a list of all the class reference documents in *UIKit Framework Reference*.

The View Interaction Model

Any time the user interacts with your user interface, or your own code programmatically changes something, a complex sequence of events takes place inside of UIKit to handle that interaction. At specific points during that sequence, UIKit calls out to your view classes and gives them a chance to respond on behalf of your application. Understanding these callout points is important to understanding where your views fit into the system. Figure 6-2 shows the basic sequence of events that starts with the user touching the screen and ends with the graphics system updating the screen content in response. Programmatic events follow the same basic steps without the initial user interaction.

Figure 6-2 UIKit interactions with your view objects



The following steps break the event sequence down even further and explain what happens at each stage and how your application might want to react in response.

1. The user touches the screen.
2. The hardware reports the touch event to the UIKit framework.
3. The UIKit framework packages the touch into a `UIEvent` object and dispatches it to the appropriate view. (For a detailed explanation of how UIKit delivers events to your views, see "Event Delivery" (page 141).)
4. The event-handling methods of your view might respond to the event by doing any of the following:

   - Adjust the properties (frame, bounds, alpha, and so on) of the view or its subviews.
   - Mark the view (or its subviews) as needing a change in its layout.
   - Mark the view (or its subviews) as needing to be redrawn.
   - Notify a controller about changes to some piece of data.

   Of course, it is up to the view to decide which of these things must be done and call the appropriate methods to do it.

5. If a view is marked as requiring layout, UIKit calls the view's `layoutSubviews` method.

   You can override this method in your custom views and use it to adjust the position and size of any subviews. For example, a view that provides a large scrollable area would need to use several subviews as "tiles" rather than create one large view, which is not likely to fit in memory anyway. Overriding this method would give that view the opportunity to hide subviews that are now offscreen or reposition them and use them to draw newly exposed content. As part of this process, the view could also mark the new tiles as needing to be redrawn.

6. If any part of the view is marked as needing to be redrawn, UIKit calls the view's `drawRect:` method.

   UIKit calls this method for only those views that need it. Each view's implementation of this method should redraw the specified area as quickly as possible. Each view should draw only its own contents and not the contents of any subviews. Views should not attempt to make any further changes to their properties or layout at this point.

7. Any updated views are composited with the rest of visible content and sent to the graphics hardware for display.

8. The graphics hardware transfers the rendered content to the screen.

Note: The preceding update model applies primarily to applications that use native views and drawing techniques. If your application draws its content using OpenGL ES, you would typically configure a single full-screen view and then draw directly to your OpenGL graphics context. Your view would still handle touch events, but it would not need to lay out subviews or implement a `drawRect:` method. For more information about using OpenGL ES, see "Drawing with OpenGL ES" (page 162).

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

1. The event-handling methods:
   - `touchesBegan:withEvent:`
   - `touchesMoved:withEvent:`
   - `touchesEnded:withEvent:`
   - `touchesCancelled:withEvent:`
2. The `layoutSubviews` method
3. The `drawRect:` method

These are the methods that most custom views implement to get the behavior they want. Depending on what you are trying to do, you may not need to override all of these methods. For example, if you are implementing a view whose size never changes, you might not need to override the `layoutSubviews` method. Similarly, views that display simple content, such as text and images, can often avoid drawing altogether by simply embedding `UIImageView` and `UILabel` objects as subviews.

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

The View Rendering Architecture

Although you use views to represent content on the screen, the `UIView` class itself actually relies heavily on another object for much of its basic behavior. Each view object in UIKit is backed by a Core Animation layer object, which is an instance of the `CALayer` class. This layer class provides the fundamental support for the layout and rendering of the view's contents and for compositing and animating the contents of the view.

Unlike Mac OS X, where Core Animation support is optional, in iPhone OS, it is integrated into the heart of the view rendering implementation. Although it plays a central role, UIKit hides that implementation wherever possible to streamline the programming experience for developers. As a result, most developers can forget that layers even exist and can simply manage their visual content using the methods and properties of the `UIView` class. Where Core Animation becomes important, however, is when the `UIView` class simply does not provide everything you need. At that point, you can dive down into the Core Animation layers and do some pretty sophisticated rendering for your application.

The following sections provide an introduction to Core Animation and describe some of the features it provides to you for free through the `UIView` class. For more detailed information about how to use Core Animation for advanced rendering, see *Core Animation Programming Guide*.

Core Animation Basics

Core Animation takes advantage of hardware acceleration and an optimized architecture to implement fast rendering and real-time animations. The first time a view's `drawRect:` method is called, the results are captured into a bitmap, which is then managed by the underlying layer. Subsequent redraw calls use this cached bitmap whenever possible to avoid calling the `drawRect:` method, which can be expensive. This process allows Core Animation to optimize its compositing operations and deliver the desired performance.

The layers associated with your view objects are stored in a hierarchy referred to as the **layer tree**. Like views, each layer in the layer tree has a single parent and can have any number of embedded sublayers. By default, the organization of objects in the layer tree matches the organization of the views in your view hierarchy. You can add layers, however, without adding a corresponding view. You might do this to implement special visual effects for which a view is not required.

Layer objects are actually the driving force behind the rendering and layout system in iPhone OS, and most view properties are actually thin wrappers for properties on the underlying layer object. When you change the property of a layer in the layer tree (directly using the `CALayer` object), the changed value is reflected immediately in the layer object. If the change triggers a corresponding animation, however, that change may not be reflected onscreen immediately; instead, it must be animated onto the screen over time. To manage these sorts of animations, Core Animation maintains two additional sets of layer objects in what are referred to as the **presentation tree** and the **render tree**.

The presentation tree reflects the state of the layers as they are currently presented to the user. When you animate the changing of a layer value, the presentation layer reflects the old value until the animation commences. As the animation progresses, Core Animation updates the value in the presentation-tree layer based on the current frame of the animation. The render tree then works together with the presentation tree to render the changes on the screen. Because the render tree runs in a separate process or thread, the work it does does not impact your application's main run loop. While both the layer tree and the presentation tree are public, the render tree is a private API.

The placement of layer objects behind your views has many important implications for the performance of your drawing code. The most important is that most geometry changes to your views do not require redrawing. For example, changing the position and size of a view does not require the system to redraw the contents of a view; it can simply reuse the cached bitmap created by the layer. Animating this cached content is significantly more efficient than trying to redraw that content every time.

The downside to using layers is that the additional cached data can add memory pressure to your application. If your application creates too many views or creates very large views, you could run out of memory quickly. You should not be afraid to use views in your application, but do not create new view objects if you have existing views that can be reused. In other words, pursue approaches that minimize the number of views you keep in memory at the same time.

For a more detailed overview of Core Animation, the object trees, and how you create animations, see *Core Animation Programming Guide*.

Changing the Layer of a View

Because views are required to have an associated layer object in iPhone OS, the `UIView` class creates this layer automatically at initialization time. You can access the layer that is created through the `layer` property of the view, but you cannot change the layer object after the view is created.

If you want a view to use a different type of layer, you must override the view's `layerClass` class method and return the class object for the layer you want it to use. The most common reason to return a different layer class is to implement an OpenGL-based application. To use OpenGL drawing commands, the layer for the underlying view must be an instance of the `CAEAGLLayer` class. This type of layer interacts with the OpenGL rendering calls to present the desired content on the screen.

You should never modify the delegate property of a view's layer, it references the view and should be considered private. Similarly, a view can only operate as a delegate for a single layer. You must not create additional layer instance and set the view object as their delegate, doing so will cause your application to crash.

Animation Support

One of the benefits of having a layer object behind every view is that it makes it easier to animate content in iPhone OS. It is important to remember that animation is not necessarily about creating visual eye candy. Animations provide the user with a context for any changes that occur in your application's user interface. For example, the use of transitions from one screen to another implies the relationship between those screens. The system provides automatic support for many of the most commonly used animations, but you can also create animations for other parts of your interface.

Many properties of the `UIView` class are considered to be animatable. An **animatable** property is one for which there is semiautomatic support for animating from one value to another. You must still tell UIKit that you want to perform the animation, but Core Animation assumes full responsibility for running the animation once it has begun. Among the properties you can animate on a `UIView` object are the following:

- `frame`
- `bounds`
- `center`
- `transform`

- `alpha`

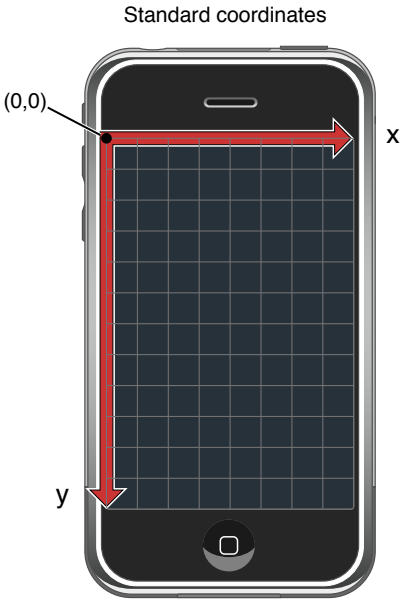
Even though other view properties are not directly animatable, you can create explicit animations for some of them. Explicit animations require you to do more of the work in managing the animation and the rendered contents but they still use the underlying Core Animation infrastructure to obtain good performance.

For more information about creating animations using the `UIView` class, see "Animating Views" (page 131). For more information about creating explicit animations, see *Core Animation Programming Guide*.

View Coordinate Systems

Coordinates in UIKit are based on a coordinate system whose origin is in the top-left corner and whose coordinate axes extend down and to the right from that point. Coordinate values are represented using floating-point numbers, which allows for precise layout and positioning of content and allows for resolution independence. Figure 6-3 (page 119) shows this coordinate system relative to the screen, but this coordinate system is also used by the `UIWindow` and `UIView` classes. This particular orientation was chosen to make it easier to lay out controls and content in user interfaces, even though it differs from the default coordinate systems in use by Quartz and Mac OS X.

Figure 6-3 View coordinate system



As you write your interface code, you need to be aware of the coordinate system currently in effect. Every window and view object maintains its own local coordinate system. All drawing in a view occurs relative to the view's local coordinate system. The frame rectangle for each view, however, is specified using the coordinate system of its parent view, and coordinates delivered as part of an event object are specified relative to the enclosing window's coordinate system. For convenience, the `UIWindow` and `UIView` classes each provide methods to convert back and forth between the coordinate systems of different objects.

Although the coordinate system used by Quartz does not use the top-left corner as the origin point, for many Quartz calls this is not a problem. Before invoking your view's `drawRect:` method, UIKit automatically configures the drawing environment to use a top-left origin. Quartz calls made within this environment draw correctly in your view. The only time you need to worry about different coordinate systems is when you set up the drawing environment yourself using Quartz.

For more information about coordinate systems, Quartz, and drawing in general, see "Graphics and Drawing" (page 151).

The Relationship of the Frame, Bounds, and Center

A view object tracks its size and location using its `frame`, `bounds`, and `center` properties. The `frame` property contains a rectangle that specifies the view's location and size relative to its parent view's coordinate system. The `bounds` property contains a rectangle that defines the view's position and size relative to its own local coordinate system and whose origin is typically set to (0, 0) but need not be. The `center` property contains the center point of the frame rectangle.

You use the `frame`, `bounds`, and `center` properties for different purposes in your code. Because the bounds rectangle represents the view's local coordinate system, you use it most often during drawing or event-handling code when you need to know where in your view something happened. The center point represents the known center point of your view and is always the best way to manipulate the position of your view. The frame rectangle is a convenience value that is computed using the `bounds` and `center` point and is valid only when the view's transform is set to the identity transform.

Figure 6-4 shows the relationship between the frame and bounds rectangles. The complete image on the right is drawn drawn in the view starting at (0, 0). Because the size of the bounds does not match the full size of the image, however, only part of the image outside the bounds rectangle is clipped automatically. When the view is composited with its parent view, the position of the view inside its parent is determined by the origin of the view's frame rectangle, which in this case is (5, 5). As a result, the view's contents appear shifted down and to the right from the parent view's origin.

Figure 6-4 Relationship between a view's frame and bounds

Frame rectangle at (5.0, 5.0), size (73.0, 98.0)

Bounds rectangle at (0.0, 0.0), size (73.0, 98.0)

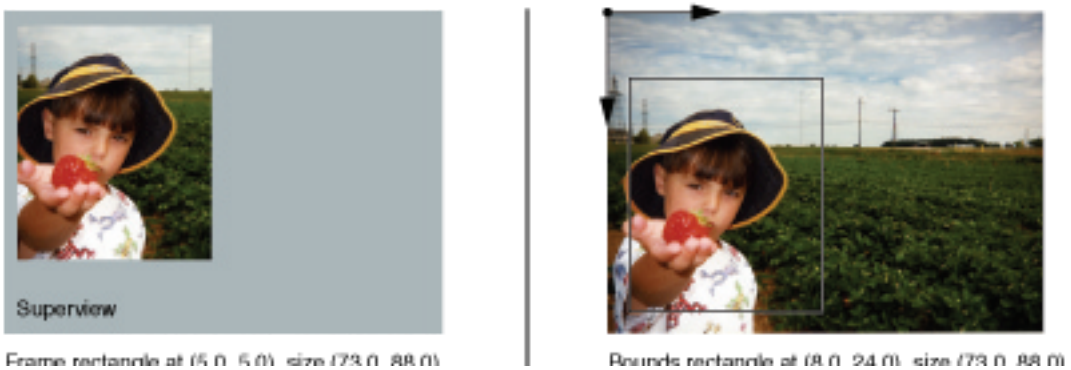
When there is no transform applied to the view, the location and size of the view is determined by these three interrelated properties. The `frame` property of a view is set when a view object is created programmatically using the `initWithFrame:` method. That method also initializes the `bounds` rectangle to originate at (0.0, 0.0) and have the same size as the view's frame. The `center` property is then set to the center point of the frame.

Although you can set the values of these properties independently, setting the value for one changes the others in the following ways:

- When you set the `frame` property, the size of the `bounds` property is set to match the size of the `frame` property. The `center` property is also adjusted to match the center point of the new frame.
- When you set the `center` property, the origin of the `frame` changes accordingly.
- When you set the size of the `bounds` rectangle, the size of the `frame` rectangle changes to match.

You can set the `bounds` origin without changing the other two properties. When you change the `bounds` origin, you change which part of the view you want to display. In Figure 6-4 (page 120), the original `bounds` origin is set to (0.0, 0.0). In Figure 6-5, that origin is moved to (8.0, 24.0). As a result, a different portion of the underlying image is displayed by the view. Because the frame rectangle did not change, however, the new content is displayed in the same location inside the parent view as before.

Figure 6-5 Altering a view's bounds



Note: By default, a view's frame is not clipped to its parent view's frame. If you want to force a view to clip its subviews, set the view's `clipsToBounds` property to `YES`.

Coordinate System Transformations

Although coordinate system transformations are commonly used in a view's `drawRect:` method to facilitate drawing, in iPhone OS, you can also use them to implement visual effects for your view. The `UIView` class includes a `transform` property that lets you apply different types of translation, scaling, and zooming effects to the entire view. By default, the value of this property is the identity transform, which causes no changes to the view. To add transformations, get the `CGAffineTransform` structure stored in this property, use the corresponding Core Graphics functions to apply the transformations, and then assign the modified transform structure back to the view's `transform` property.

Note: When applying transforms to a view, all transformations are performed relative to the `center` point of the view.

Translating a view shifts all subviews along with the drawing of the view's content. Because their coordinate systems inherit and build on these alterations, scaling also affects the drawing of the subviews. For more information about how to control the scaling of view content, see "Content Modes and Scaling" (page 122).

Warning: If the `transform` property is not the identity transform, the value of the `frame` property is undefined and must be ignored. After setting the transform, use the `bounds` and `center` properties to get the position and size of the view.

For information about using transforms in conjunction with your `drawRect:` method, see "The Native Coordinate System" (page 152). For information about the functions you use to modify the `CGAffineTransform` struct, see *CGAffineTransform Reference*.

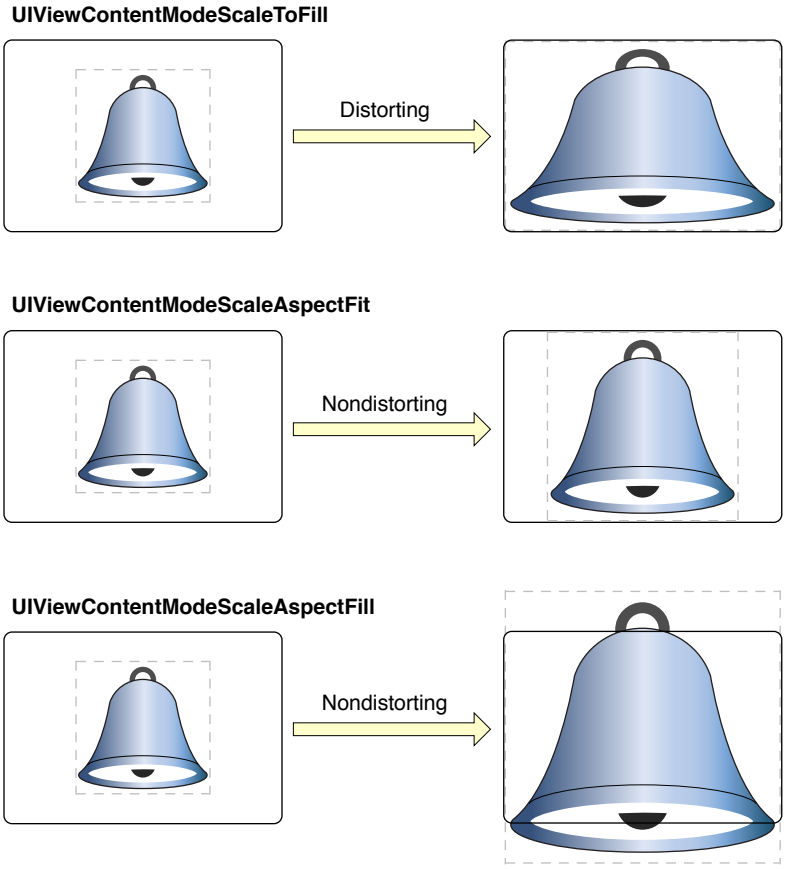
Content Modes and Scaling

When you change the bounds of a view or apply a scaling factor to the `transform` property of a view, the frame rectangle is changed by a commensurate amount. Depending on the content mode associated with the view, the view's content may also be scaled or repositioned to account for the changes. The view's `contentMode` property determines the effect that bounds changes and scaling operations have on the view. By default, the value of this property is set to `UIViewContentModeScaleToFill`, which always causes the view's contents to be scaled to fit the new frame size. For example, Figure 6-6 shows what happens when the horizontal scaling factor of the view is doubled.

Figure 6-6 View scaled using the scale-to-fill content mode

Scaling of your view's content occurs because the first time a view is shown, its rendered contents are cached in the underlying layer. Rather than force the view to redraw itself every time its bounds change or a scaling factor is applied, UIKit uses the view's content mode to determine how to display the cached content. Figure 6-7 compares the results of changing the bounds of a view or applying a scaling factor to it using several different content modes.

Figure 6-7 Content mode comparisons



Although applying a scaling factor always scales the view's contents, there are content modes that do not scale the view's contents when the bounds of the view change. Several `UIViewContentMode` constants (such as `UIViewContentModeTop` and `UIViewContentModeBottomRight`) display the current content in different corners or along different edges of the view. There is also a mode for displaying the content centered inside the view. Changing the bounds rectangle with one of these content modes in place simply moves the existing contents to the appropriate location inside the new bounds rectangle.

One place where you might use content modes is in the implementation of resizable controls. Buttons and segmented controls typically use several images to create the appearance of the control. In addition to two fixed-size end cap images, a button that can grow horizontally uses a stretchable center image that is only 1 pixel wide. By displaying each image in its own image view and setting the content mode of the stretchable middle image to `UIViewContentModeScaleToFill`, the button can grow in size without distorting the appearance of the end caps. More importantly, the images associated with each image view can be cached by Core Animation and animated without any custom drawing code, which results in much better performance.

Although content modes are good to avoid redrawing the contents of your view, you can also use the `UIViewContentModeRedraw` content mode in situations where you specifically want control over the appearance of your view during scaling and resizing operations. Setting your view's content mode to this value forces Core Animation to invalidate your view's contents and call your view's `drawRect:` method rather than scale or resize them automatically.

Autoresizing Behaviors

When you change the frame rectangle of a view, the position and size of embedded subviews often needs to change to match the new size of the original view. If the `autoresizesSubviews` property of a view is set to `YES`, its subviews are automatically resized according to the values in the `autoresizingMask` property. (By default, the `autoresizesSubviews` property is `NO` so you need set this to `YES` to use this feature.) In many cases simply configuring the autoresizing mask for a view provides the appropriate behavior for an application. Otherwise, it is the application's responsibility to reposition and resize the subviews by overriding the `layoutSubviews` method.

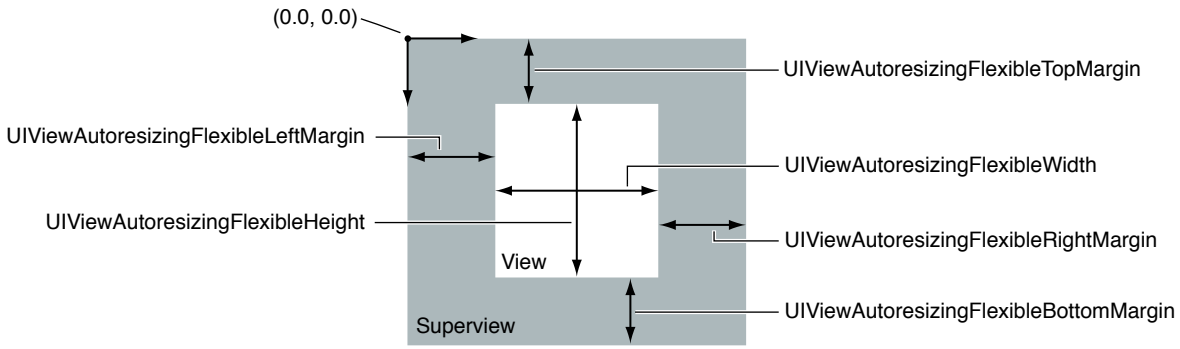
To set a view's autoresizing behaviors, combine the desired autoresizing constants using a bitwise OR operator and assign the resulting value to the view's `autoresizingMask` property. Table 6-1 lists the autoresizing constants and describes how each one affects the size and placement of a given view. For example, to keep a view pinned to the lower-left corner of its superview, add the `UIViewAutoresizingFlexibleRightMargin`and `UIViewAutoresizingFlexibleTopMargin` constants and assign them to the `autoresizingMask` property. When more than one aspect along an axis is made flexible, the resize amount is distributed evenly among them.

Table 6-1 Autoresizing mask constants

Autoresizing mask	Description
`UIViewAutoresizingNone`	If set, the view doesn't autoresize.
`UIViewAutoresizing-FlexibleHeight`	If set, the view's height changes proportionally to the change in the superview's height. Otherwise, the view's height does not change relative to the superview's height.
`UIViewAutoresizing-FlexibleWidth`	If set, the view's width changes proportionally to the change in the superview's width. Otherwise, the view's width does not change relative to the superview's width.
`UIViewAutoresizing-FlexibleLeftMargin`	If set, the view's left edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's left edge remains in the same position relative to the superview's left edge.
`UIViewAutoresizing-FlexibleRightMargin`	If set, the view's right edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's right edge remains in the same position relative to the superview.
`UIViewAutoresizing-FlexibleBottomMargin`	If set, the view's bottom edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's bottom edge remains in the same position relative to the superview.
`UIViewAutoresizing-FlexibleTopMargin`	If set, the view's top edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's top edge remains in the same position relative to the superview.

Figure 6-8 provides a graphical representation of the position of the constant values. When one of these constants is omitted, the view's layout is fixed in that aspect; when a constant is included in the mask, the view's layout is flexible in that aspect.

Figure 6-8 View autoresizing mask constants



If you are using Interface Builder to configure your views, you can set the autoresizing behavior for each view using the Autosizing controls in the Size inspector. Although the flexible width and height constants from the preceding figure have the same behavior as the Interface Builder springs located in the same position have, the behavior of the margin constants is effectively reversed. In other words, to apply the flexible right margin autoresizing behavior to a view in Interface Builder, you must leave the space on that side of the Autosizing control empty, not place a strut there. Fortunately, Interface Builder provides an animation to show you how changes to the autoresizing behaviors affect your view.

If the `autoresizesSubviews` property of a view is set to `NO`, any autoresizing behaviors set on the immediate subviews of that view are ignored. Similarly, if a subview's autoresizing mask is set to `UIViewAutoresizingNone`, it does not change size and so its immediate subviews are never resized either.

Note: For autoresizing to work correctly, the view's `transform` property must be set to the identity transform. The behavior is undefined if it is not.

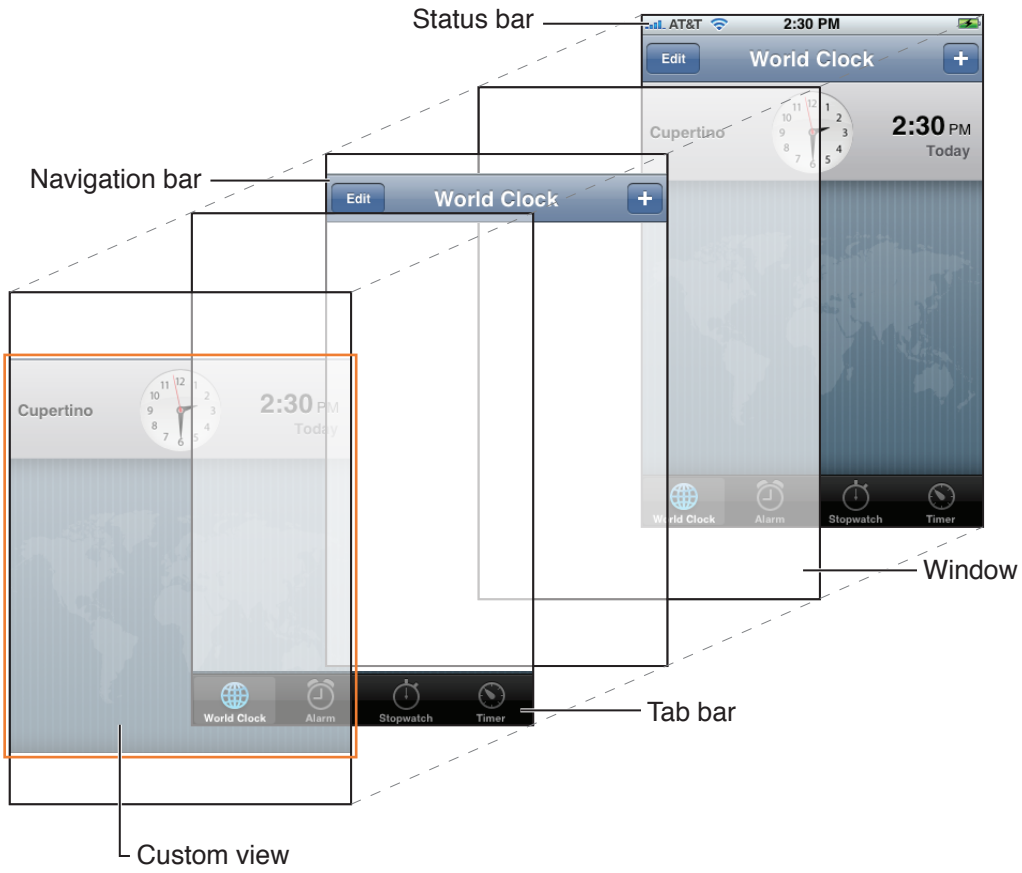
Although autoresizing behaviors may be suitable for some layout needs, if you want more control over the layout of your views, you should override the `layoutSubviews` method in the appropriate view classes. For more information about managing the layout of your views, see "Responding to Layout Changes" (page 133).

Creating and Managing the View Hierarchy

Managing the view hierarchy of your user interface is a crucial part of developing your application's user interface. The organization of your views defines not only the way your application appears visually but also how your application responds to changes. The parent-child relationships in the view hierarchy help define the chain of objects that is responsible for handling touch events in your application. When the user rotates the device, parent-child relationships also help define how each view's size and position are altered by changes to the user interface orientation.

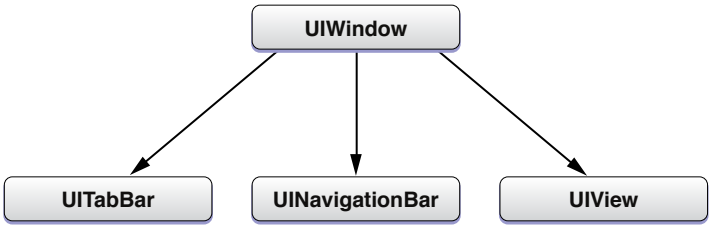
Figure 6-9 shows a simple example of how the layering of views creates a desired visual effect. In the case of the Clock application, tab-bar and navigation-bar views are mixed together with a custom view to implement the overall interface.

Figure 6-9 Layered views in the Clock application



If you look at the object relationships for the views in the Clock application, you see that they look something like the relationships shown in "Changing the Layer of a View." The window object acts as the root view for the tab bar, navigation bar, and custom view of the application.

Figure 6-10 View hierarchy for the Clock application



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

Creating a View Object

The simplest way to create views is to use Interface Builder and load them from the resulting nib file. From Interface Builder's graphical environment, you can drag new views out of the library and drop them onto a window or another view and build your view hierarchies quickly. Because Interface Builder uses live view objects, building your interface graphically shows you exactly how it will look when you load it at runtime. It also eliminates the need for you to write tedious code to allocate and initialize each view in your view hierarchy.

If you prefer not to use Interface Builder and nib files to create your views, you can create them programmatically. To create a new view object, allocate memory for the view object and send that object an `initWithFrame:` message to initialize it. For example, to create a new instance of the `UIView` class, which you could use as a container for other views, you would use the following code:

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

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

The frame rectangle you specify when you initialize the view represents the position and size of the view relative to its intended parent view. You must add views to a window or to another view to make them appear on the screen. When doing so, UIKit uses the frame rectangle you specify to place the view inside its parent. For information on how to add views to your view hierarchy, see "Adding and Removing Subviews" (page 127).

Adding and Removing Subviews

Interface Builder is the most convenient way to build view hierarchies because it lets you see exactly how those views will appear at runtime. It then saves the view objects and their hierarchical relationships in a nib file, which the system uses at runtime to recreate the objects and relationships in your application. When a nib file is loaded, the system automatically calls the `UIView` methods needed to recreate the view hierarchy.

If you prefer not to use Interface Builder and nib files to create your view hierarchies, you can create them programmatically instead. A view that has required subviews should create them in its own `initWithFrame:` method to ensure that they are present and initialized with the view. Subviews that are part of your application design (and not required for the operation of your view) should be created outside of your view's initialization code. In iPhone applications, the two most common places to create views and subviews programmatically are the `applicationDidFinishLaunching:` method of your application delegate and the `loadView` method of your view controllers.

To manipulate views in the view hierarchy, you use the following methods:

- To add a subview to a parent, call the `addSubview:` method of the parent view. This method adds the subview to the end of the parent's list of subviews.
- To insert a subview in the middle of the parent's list of subviews, call any of the `insertSubview:...` methods of the parent view.

- To reorder existing subviews inside their parent, call the `bringSubviewToFront:`, `sendSubviewToBack:`, or `exchangeSubviewAtIndex:withSubviewAtIndex:` methods of the parent view. These methods are faster than removing the subviews and reinserting them.
- To remove a subview from its parent, call the `removeFromSuperview` method of the subview (not the parent view).

When adding subviews, the current frame rectangle of the subview is used as the initial position of that view inside its parent. You can change that position at any time by changing the `frame` property of the subview. Subviews whose frame lies outside of their parent's visible bounds are not clipped by default. To enable clipping, you must set the `clipsToBounds` property of the parent view to `YES`.

Listing 6-1 shows a sample `applicationDidFinishLaunching:` method of an application delegate object. In this example, the application delegate creates its entire user interface programmatically at launch time. The interface consists of two generic `UIView` objects, which display primary colors. Each view is then embedded inside a window, which is also a subclass of `UIView` and can therefore act as a parent view. Because parents retain their subviews, this method releases the newly created views to prevent them from being overretained.

Listing 6-1 Creating a window with views

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Create the window object and assign it to the
    // window instance variable of the application delegate.
    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    window.backgroundColor = [UIColor whiteColor];

    // Create a simple red square
    CGRect redFrame = CGRectMake(10, 10, 100, 100);
    UIView *redView = [[UIView alloc] initWithFrame:redFrame];
    redView.backgroundColor = [UIColor redColor];

    // Create a simple blue square
    CGRect blueFrame = CGRectMake(10, 150, 100, 100);
    UIView *blueView = [[UIView alloc] initWithFrame:blueFrame];
    blueView.backgroundColor = [UIColor blueColor];

    // Add the square views to the window
    [window addSubview:redView];
    [window addSubview:blueView];

    // Once added to the window, release the views to avoid the
    // extra retain count on each of them.
    [redView release];
    [blueView release];

    // Show the window.
    [window makeKeyAndVisible];
}
```

Important: When you're considering memory management, think of the subviews as any other collection object. Specifically, when you insert a view as a subview using `addSubview:`, that subview is retained by its superview. Inversely, when you remove the subview from its superview using the `removeFromSuperview` method, the subview is autoreleased. Releasing views after adding them to your view hierarchy prevents them being overretained, which could cause memory leaks.

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

When you add a subview to a parent view, UIKit sends several messages to both the parent and child to let them know what is happening. You can override methods such as `willMoveToSuperview:`, `willMoveToWindow:`, `willRemoveSubview:`, `didAddSubview:`, `didMoveToSuperview` and `didMoveToWindow` in your custom views to process changes before and after they occur and to update the state information in your view accordingly.

After you create a view hierarchy, you can use the `superview` property of a view to get its parent or the `subviews` property to get its children. You can also use the `isDescendantOfView:` method to determine whether a view is in the view hierarchy of a parent view. Because the root view in a view hierarchy has no parent, its `superview` property is set to `nil`. For views currently onscreen, the window object is typically the root view of the hierarchy.

You can use the `window` property of a view to get a pointer to the window that currently contains the view (if any). This property is set to `nil` if the view is not currently attached to a window.

Converting Coordinates in the View Hierarchy

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

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

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

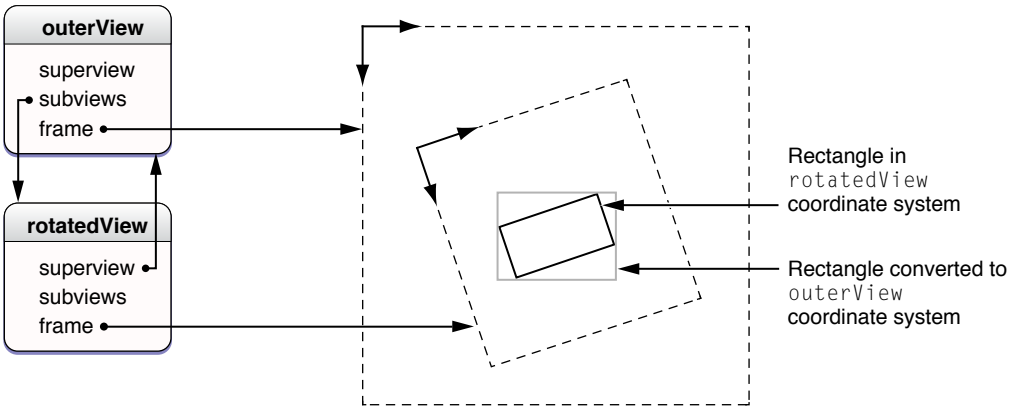
In addition to the `UIView` conversion methods, the `UIWindow` class also defines several conversion methods. These methods are similar to the `UIView` versions except that instead of converting to and from a view's local coordinate system, these methods convert to and from the window's coordinate system.

- `convertPoint:fromWindow:`

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

Coordinate conversions are straightforward when neither view is rotated or when dealing only with points. When converting rectangles or sizes between views with different rotations, the geometric structure must be altered in a reasonable way so that the resulting coordinates are correct. When converting a rectangle, the `UIView` class assumes that you want to guarantee coverage of the original screen area. To this end, the converted rectangle is enlarged so that when located in the appropriate view, it completely covers the original rectangle. Figure 6-11 shows the conversion of a rectangle in the `rotatedView` object's coordinate system to that of its superview, `outerView`.

Figure 6-11 Converting values in a rotated view



When converting size information, `UIView` simply treats it as a delta offset from (0.0, 0.0) that you need to convert from one view to another. Though the offset distance remains the same, the balance along the two axes shifts according to the rotation. When converting sizes, UIKit always returns sizes that consist of positive numbers.

Tagging Views

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

To search for a tagged view, use the `viewWithTag:` method of `UIView`. This method searches the receiver's subviews using a depth-first search, starting with the receiver itself.

Modifying Views at Runtime

As applications receive input from the user, they adjust their user interface in response to that input. An application might rearrange the views in its interface, refresh existing views that contain changed data, or load an entirely new set of views. Which techniques you choose to use depends on your interface and what you are trying to achieve. How you initiate these techniques, however, is the same for all applications. The following sections describe these techniques and how you use them to update your user interface at runtime.

Note: For background information about how UIKit moves events and messages between itself and your custom code, see "The View Interaction Model" (page 114) before proceeding.

Animating Views

Animations are a way to provide fluid visual transitions between different states of your user interface. In iPhone OS, animations are used extensively to reposition views, change their size, and even change their alpha value to make them fade in or out. Because this support is crucial for making easy-to-use applications, UIKit simplifies the process of creating animations by integrating support for them directly into the `UIView` class.

The `UIView` class defines several properties that are inherently **animatable**—that is, the view provides built-in support for animating changes in the property from their current value to a new value. Although the work needed to perform the animation is handled for you automatically by the `UIView` class, you must still let the view know that you want the animation to happen. You do this by wrapping changes to the given property in an animation block.

An **animation block** starts with a call to the `beginAnimations:context:` class method of `UIView` and ends with a call to the `commitAnimations` class method. In between these calls is where you configure the animation parameters and change the properties you want to animate. As soon as you call the `commitAnimations` method, UIKit kicks off the animations, animating any changes from their current values to the new values you just set. Animation blocks can be nested, but nested animations do not start until the outermost animation block is committed.

Table 6-2 lists the animatable properties of the `UIView` class.

Table 6-2 Animatable properties

Property	Description
`frame`	The view's frame rectangle, in superview coordinates.
`bounds`	The view's bounding rectangle, in view coordinates.
`center`	The center of the frame, in superview coordinates.
`transform`	The transform applied to the view, relative to the center of its bounds.
`alpha`	The view's alpha value, which determines the view's level of transparency.

Configuring Animation Parameters

In addition to changing property values inside an animation block, you can configure additional parameters that determine how you want the animation to proceed. You do this by calling the following class methods of `UIView`:

- Use the `setAnimationStartDate:` method to set the start date of the animations after the `commitAnimations` method returns. The default behavior starts animations immediately.
- Use the `setAnimationDelay:` method to set a delay between the time the `commitAnimations` method returns and the animations actually begin.
- Use the `setAnimationDuration:` method to set the number of seconds over which the animations occur.
- Use the `setAnimationCurve:` method to set the relative speed of the animations over their course. For example, the animations can gradually speed up at the beginning, gradually slow down near the end, or remain the same speed throughout.
- Use the `setAnimationRepeatCount:` method to set the number of times the animations repeat.
- Use the `setAnimationRepeatAutoreverses:` method to set whether the animations reverse automatically when they reach their target value. Combined with the `setAnimationRepeatCount:` method, you can use this method to toggle each property between its initial and final values smoothly over a period of time.

The `commitAnimations` class method returns immediately and before the animations begin. UIKit runs animations in a separate thread and away from your application's main event loop. The `commitAnimations` method posts its animations to this separate thread where they are queued up until they are ready to execute. By default, Core Animation finishes the currently running animation block before starting animations currently on the queue. You can override this behavior and start your animation immediately, however, by passing `YES` to the `setAnimationBeginsFromCurrentState:` class method within your animation block. This causes the current in-flight animation to stop and the new animation to begin from the current state.

By default, all animatable property changes within an animation block are animated. If you want to prevent some changes made within the block from being animated, use the `setAnimationsEnabled:` method to disable animations temporarily, make your changes, and then reenable them. Any changes made after a `setAnimationsEnabled:` call with the value `NO` are not animated until a matching call with the value `YES` occurs or you commit the animation block. Use the `areAnimationsEnabled` method to determine if animations are currently enabled.

Configuring an Animation Delegate

You can assign a delegate to an animation block and use that delegate to receive messages when the animations begin and end. You might do this to perform additional tasks immediately before and after the animation. You set the delegate using the `setAnimationDelegate:` class method of `UIView` and use the `setAnimationWillStartSelector:` and `setAnimationDidStopSelector:` methods to specify the selectors that will receive the messages. The signatures of the corresponding methods are as follows:

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

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

- *animationID*—an application-supplied string used to identify animations in an animation block.
- *context*—another application-supplied object you can use to pass additional information to the delegate.

The `setAnimationDidStopSelector:` selector method has an additional argument—a Boolean value that is `YES` if the animation ran to completion and was not canceled or stopped prematurely by another animation.

Responding to Layout Changes

Whenever the layout of your views changes, UIKit applies each view's autoresizing behaviors and then calls its `layoutSubviews` method to give it a chance to adjust the geometry of its contained subviews further. Layout changes can occur when any of the following happens:

- The size of a view's bounds rectangle changes.
- The content offset value (the origin of the visible content region) of a scroll view changes.
- The transform associated with the view changes.
- The set of sublayers associated with the view's layer change.
- Your application forces layout to occur by calling the `setNeedsLayout` or `layoutIfNeeded` methods of the view.
- Your application forces layout by calling the `setNeedsLayout` method of the view's underlying layer.

A view's autoresizing behaviors handle the initial job of positioning any subviews. Applying these behaviors guarantees that your views are close to their intended size. For information about how autoresizing behaviors affect the size and position of your views, see "Autoresizing Behaviors" (page 124).

There are a few situations where you might want to adjust the layout of subviews manually using `layoutSubviews`, rather than rely exclusively on autoresizing behaviors. If you are implementing a custom control that is built from several subview elements, adjusting the subviews manually would let you precisely configure the appearance for your control over a range of sizes. Alternately, a view representing a large scrollable content area could display that content by tiling a set of subviews. During scrolling, views going off one edge of the screen would be recycled and repositioned at the incoming screen edge along with any new content.

Note: You can also use the `layoutSubviews` method to adjust the size and position of custom `CALayer` objects attached as sublayers to your view's layer. Managing custom layer hierarchies behind your view lets you perform advanced animations directly using Core Animation. For more information about using Core Animation to manage layer hierarchies, see *Core Animation Programming Guide*.

When writing your layout code, be sure to test your code in each of your application's supported interface orientations. Applications that support both landscape and portrait orientations should verify that layout is handled properly in each orientation. Similarly, your application should be

prepared to deal with other system changes, such as the height of the status bar changing. This occurs when a user uses your application while on an active phone call and then hangs up. At hang up time, the managing view controller may resize its view to account for the shrinking status bar size. Such a change would then filter down to the rest of the views in your application.

Redrawing Your View's Content

When changes to your application's data model require changes to the corresponding user interface, the way you make those changes is to mark the corresponding views as dirty and in need of an update using either the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods. Marking views as dirty, as opposed to simply creating a graphics context and drawing, gives the system a chance to process drawing operations more efficiently. For example, if you mark several regions of the same view as dirty during a given cycle, the system coalesces the dirty regions into a single call to the view's `drawRect:` method. This results in only one graphics context being created to draw all of the affected regions, which is much more efficient than creating several graphics contexts in quick succession.

Views that implement a `drawRect:` method should always check the rectangle passed to the method and use it to limit the scope of their drawing operations. Because drawing is a relatively expensive operation, limiting drawing in this way is a good way to improve performance.

By default, geometry changes to a view do not automatically cause the view to be redrawn. Instead, most geometry changes are handled automatically by Core Animation. Specifically, when you change the `frame`, `bounds`, `center`, or `transform` properties of the view, Core Animation applies the geometry changes to the cached bitmap associated with the view's layer. In many cases, this approach is perfectly acceptable, but if you find the results undesirable, you can force UIKit to redraw your view instead. To prevent Core Animation from applying geometry changes implicitly, set your view's `contentMode` property to `UIViewContentModeRedraw`. For more information about content modes, see "Content Modes and Scaling" (page 122).

Hiding Views

You can hide or show a view by changing the value in the view's `hidden` property. Setting this property to `YES` hides the view whereas setting it to `NO` shows it. Hiding a view also hides any embedded subviews as if their own their `hidden` property were set.

When you hide a view, it remains in the view hierarchy, but its contents are not drawn and it does not receive touch events. Because it remains in the view hierarchy, a hidden view continues to participate in autoresizing and other layout operations. If you hide a view that is currently the first responder, the view does not automatically resign its first responder status. Events targeted at the first responder are still delivered to the hidden view.

Creating a Custom View

The `UIView` class provides the underlying support for displaying content on the screen and for handling touch events, but its instances won't draw anything but a background color using an alpha value and its subviews. If your application needs to display custom content or handle touch events in a specific manner, you must create a custom subclass of `UIView`.

The following sections describe some of the key methods and behaviors you might implement in your custom view objects. For additional subclassing information, see *UIView Class Reference*.

Initializing Your Custom View

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

Listing 6-2 shows a skeletal implementation of a standard `initWithFrame:` method. This method calls the inherited implementation of the method first and then initializes the instance variables and state information of the class before returning the initialized object. Calling the inherited implementation is traditionally performed first so that if there is a problem, you can simply abort your own initialization code and return `nil`.

Listing 6-2 Initializing a view subclass

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

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

Even if your view adopts the `NSCoding` protocol, Interface Builder does not know about your view's custom properties and therefore does not encode those properties into the nib file. As a result, your own `initWithCoder:` method does not have the information it needs to properly initialize the class when it is loaded from a nib file. To solve this problem, you can implement the `awakeFromNib` method in your class and use it to initialize your class specifically when it is loaded from a nib file.

Drawing Your View's Content

As you make changes to your view's content, you notify the system that parts of that view need to be redrawn using the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods. When the application returns to its run loop, it coalesces any drawing requests and computes the specific parts of your interface that need to be updated. It then begins traversing your view hierarchy and sending `drawRect:` messages to the views that require updates. The traversal starts with the root view of your hierarchy and proceeds down through the subviews, processing them from back to front. Views that display custom content inside their visible bounds must implement the `drawRect:` method to render that content.

Prior to calling your view's `drawRect:` method, UIKit configures the drawing environment for your view. It creates a graphics context and adjusts its coordinate system and clipping region to match the coordinate system and bounds of your view. Thus, by the time your `drawRect:` method is called, you

can simply begin drawing using UIKit classes and functions, Quartz functions, or a combination of them all. If you need to access the current graphics context, you can get a pointer to it using the `UIGraphicsGetCurrentContext` function.

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

Listing 6-3 shows a simple implementation of a `drawRect:` method that draws a 10-pixel wide red border around the view. Because UIKit drawing operations use Quartz for their underlying implementations, you can mix drawing calls as shown here and still get the results you expect.

Listing 6-3 Drawing method

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

    CGContextSetLineWidth(context, 10);

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

If you know that your view's drawing code always covers the entire surface of the view with opaque content, you can improve the overall efficiency of your drawing code by setting the `opaque` property of your view to `YES`. Marking a view as opaque lets UIKit avoid drawing content that is located immediately behind your view. This not only reduces the amount of time spent drawing but also minimizes the work that must be done to composite that content together. You should set this property to `YES` only if you know your view provides opaque content. If your view cannot guarantee that its contents are always opaque, you should set the property to `NO`.

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

Responding to Events

The `UIView` class is a subclass of `UIResponder` and is therefore capable of receiving touch events corresponding to user interactions with the view's contents. Touch events start at the view in which the touch occurred and are passed up the responder chain until they are handled. Because views are themselves responders, they participate in the responder chain and are therefore capable of receiving touch events dispatched to them from any of their associated subviews.

Views that handle touch events typically implement all of the following methods, which are described in more detail in "Event Handling" (page 139).

- `touchesBegan:withEvent:`

- touchesMoved:withEvent:
- touchesEnded:withEvent:
- touchesCancelled:withEvent:

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

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

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

Cleaning Up After Your View

If your view class allocates any memory, stores references to any custom objects, or holds resources that must be released when the view is released, you must implement a `dealloc` method. The system calls the `dealloc` method when your view's retain count reaches zero and your view is about to be deallocated itself. Your implementation of this method should release the objects and resources it holds and then call the inherited implementation, as shown in Listing 6-4.

Listing 6-4 Implementing the dealloc method

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

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

Event Handling

Events in iPhone OS are based on a Multi-Touch model. Instead of using a mouse and a keyboard, users touch the screen of the device to manipulate objects, enter data, and otherwise convey their intentions. iPhone OS recognizes one or more fingers touching the screen as part of a **Multi-Touch sequence**. This sequence begins when the first finger touches down on the screen and ends when the last finger is lifted from the screen. iPhone OS tracks fingers touching the screen throughout a multi-touch sequence and records the characteristics of each of them, including the location of the finger on the screen and the time the touch occurred. Applications often recognize certain combinations of touches as gestures and respond to them in ways that are intuitive to users, such as zooming in on content in response to a pinching gesture and scrolling through content in response to a flicking gesture.

Note: A finger on the screen affords a much different level of precision than a mouse pointer. When a user touches the screen, the area of contact is actually elliptical and tends to be offset below the point where the user thinks he or she touched. This "contact patch" also varies in size and shape based on which finger is touching the screen, the size of the finger, the pressure of the finger on the screen, the orientation of the finger, and other factors. The underlying Multi-Touch system analyzes all of this information for you and computes a single touch point.

Many classes in UIKit handle multi-touch events in ways that are distinctive to objects of the class. This is especially true of subclasses of `UIControl`, such as `UIButton` and `UISlider`. Objects of these subclasses—known as control objects—are receptive to certain types of gestures, such as a tap or a drag in a certain direction; when properly configured, they send an action message to a target object when that gesture occurs. Other UIKit classes handle gestures in other contexts; for example, `UIScrollView` provides scrolling behavior for table views, text views, and other views with large content areas.

Some applications may not need to handle events directly; instead, they can rely on the classes of UIKit for that behavior. However, if you create a custom subclass of `UIView`—a common pattern in iPhone OS development—and if you want that view to respond to certain touch events, you need to implement the code required to handle those events. Moreover, if you want a UIKit object to respond to events differently, you have to create a subclass of that framework class and override the appropriate event-handling methods.

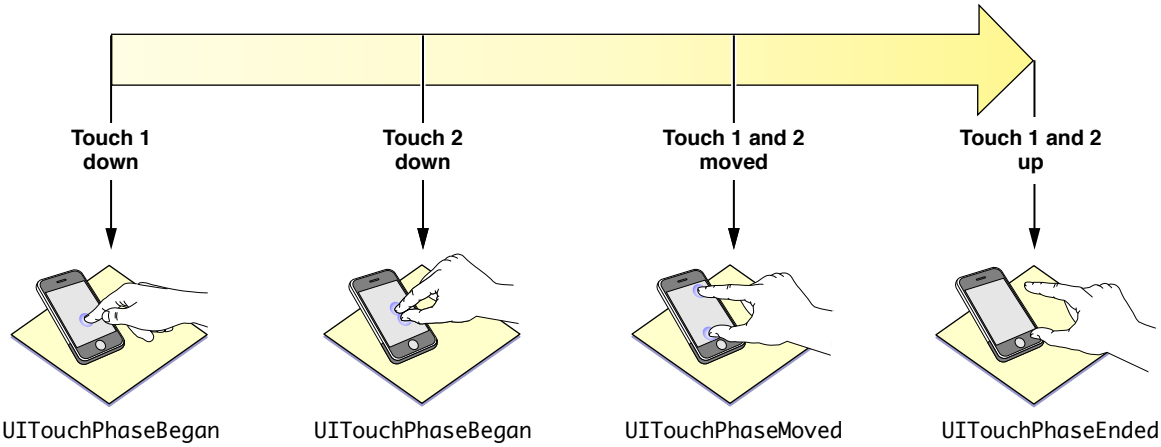
Events and Touches

In iPhone OS, a **touch** is the presence or movement of a finger on the screen that is part of a unique **multi-touch sequence**. For example, a pinch-close gesture has two touches: two fingers on the screen moving toward each other from opposite directions. There are simple single-finger gestures, such as a tap, or a double-tap, or a flick (where the user quickly swipes a finger across the screen). An application might recognize even more complicated gestures; for example, an application might have a custom control in the shape of a dial that users "turn" with multiple fingers to fine-tune some variable.

An **event** is an object that the system continually sends to an application as fingers touch the screen and move across its surface. The event provides a snapshot of all touches during a multi-touch sequence, most importantly the touches that are new or have changed for a particular view. A multi-touch sequence begins when a finger first touches the screen. Other fingers may subsequently touch the screen, and all fingers may move across the screen. The sequence ends when the last of these fingers is lifted from the screen. An application receives event objects during each phase of any touch.

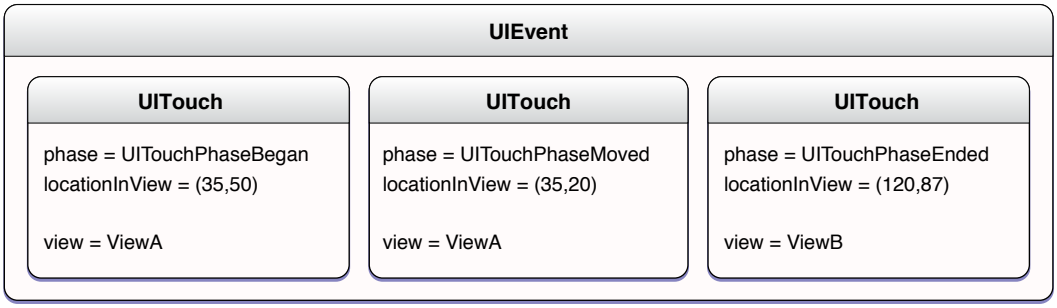
Touches have both temporal and spatial aspects. The temporal aspect, called a phase, indicates when a touch has just begun, whether it is moving or stationary, and when it ends—that is, when the finger is lifted from the screen (see Figure 7-1). A touch also has the current location in a view or window and the previous location (if any). When a finger touches the screen, the touch is associated with a window and a view and maintains that association throughout the life of the event. If multiple touches arrive at once, they are treated together only if they are associated with the same view. Likewise, if two touches arrive in quick succession, they are treated as a multiple tap only if they are associated with the same view.

Figure 7-1 A multi-touch sequence and touch phases



In iPhone OS, a `UITouch` object represents a touch, and a `UIEvent` object represents an event. An event object contains all touch objects for the current multi-touch sequence and can provide touch objects specific to a view or window (see Figure 7-2). A touch object is persistent for a given finger during a sequence, and UIKit mutates it as it tracks the finger throughout it. The touch attributes that change are the phase of the touch, its location in a view, its previous location, and its timestamp. Event-handling code evaluates these attributes to determine how to respond to the event.

Figure 7-2 Relationship of a `UIEvent` object and its `UITouch` objects



The system can cancel a multi-touch sequence at any time and an event-handling application must be prepared to respond appropriately. Cancellations can occur as a result of overriding system events, such as an incoming phone call.

Event Delivery

The delivery of an event to an object for handling occurs along a specific path. As described in "Core Application Architecture" (page 87), when users touch the screen of a device, iPhone OS recognizes the set of touches and packages them in a `UIEvent` object that it places in the current application's event queue. The event object encapsulates the touches for a given moment of a multi-touch sequence. The singleton `UIApplication` object that is managing the application takes an event from the top of the queue and dispatches it for handling. Typically, it sends the event to the application's key window—the window currently the focus for user events—and the `UIWindow` object representing that window sends the event to the first responder for handling. (The first responder is described in "Responder Objects and the Responder Chain.")

An application uses hit-testing to find the first responder for an event; it recursively calls `hitTest:withEvent:` on the views in the view hierarchy (going down the hierarchy) to determine the subview in which the touch took place. The touch is associated with that view for its lifetime, even if it subsequently moves outside the view. "Event-Handling Techniques" (page 148) discusses some of the programmatic implications of hit-testing.

The `UIApplication` object and each `UIWindow` object dispatches events in the `sendEvent:` method. (Both classes declare an identically named method). Because these methods are funnel points for events coming into an application, you can subclass `UIApplication` or `UIWindow` and override the `sendEvent:` method to monitor events or perform special event handling. However, most applications have no need to do this.

Responder Objects and the Responder Chain

A **responder object** is an object that can respond to events and handle them. `UIResponder` is the base class for all responder objects. It defines the programmatic interface not only for event handling but for common responder behavior. `UIApplication`, `UIView`, and all UIKit classes that descend from `UIView` (including `UIWindow`) inherit directly or indirectly from `UIResponder`.

The **first responder** is the responder object in the application (usually a `UIView` object) that is the current recipient of touches. A `UIWindow` objects sends the first responder an event in a message, giving it the first shot at handling the event. If the first responder doesn't handle the event, it passes the event (via message) to the next responder in the responder chain to see if it can handle it.

The **responder chain** is a linked series of responder objects. It allows responder objects to delegate responsibility for handling an event to other, higher-level objects. An event proceeds up the responder chain as the application looks for an object capable of handling the event. The responder chain consists of a series of "next responders" in the following sequence:

1. The first responder passes the event to its view controller (if it has one) and then on to its superview.
2. Each subsequent view in the hierarchy similarly passes to its view controller first (if it has one) and then to its superview.
3. The topmost enclosing view passes the event to the `UIWindow` object.
4. The `UIWindow` object passes the event to the singleton `UIApplication` object.

If the application finds no responder object to handle the event, it discards the event.

Any responder object in the responder chain may implement a `UIResponder` event-handling method and thus receive an event message. But a responder may decline to handle a particular event or may handle it only partially. In that case, it can forward the event message to the next responder in a message similar to the following one:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch* touch = [touches anyObject];
    NSUInteger numTaps = [touch tapCount];
    if (numTaps <= 2) {
        [self.nextResponder touchesBegan:touches withEvent:event];
   } else {
        [self handleDoubleTap:touch];
   }
}
```

Note: If a responder object forwards event-handling messages to the next responder for the initial phase of multi-touch sequence (in `touchesBegan:withEvent:`), it should forward all other event-handling messages for that sequence.

Action messages also make use of the responder chain. When users manipulate a `UIControl` object such as button or page control, the control object (if properly configured) sends an action message to a target object. But if `nil` is specified as the target, the application initially routes the message as it does an event message: to the first responder. If the first responder doesn't handle the action message, it sends it to its next responder, and so on up the responder chain.

Regulating Event Delivery

UIKit gives applications programmatic means to simplify event handling or to turn off the stream of events completely. The following list summarizes these approaches:

- **Turning off delivery of events**. By default, a view receives touch events, but you can set its `userInteractionEnabled` property to `NO` to turn off delivery of events. A view also does not receive events if it's hidden or if it's transparent.
- **Turning off delivery of events for a period**. An application can call the `UIApplication` method `beginIgnoringInteractionEvents` and later call the `endIgnoringInteractionEvents` method. The first method stops the application from receiving touch event messages entirely; the second method is called to resume the receipt of such messages. You sometimes want to turn off event delivery when your code is performing animations.
- **Turning on delivery of multiple touches**. By default, a view ignores all but the first touch during a multi-touch sequence. If you want the view to handle multiple touches you must enable multiple touches for the view. This can be done programmatically by setting the `multipleTouchEnabled` property of your view to `YES`, or in Interface Builder using the inspector for the related view.
- **Restricting event delivery to a single view**. By default, a view's `exclusiveTouch` property is set to `NO`. If you set the property to `YES`, you mark the view so that, if it is tracking touches, it is the only view in the window that is tracking touches. Other views in the window cannot receive those touches. However, a view that is marked "exclusive touch" does not receive touches that are associated with other views in the same window. If a finger contacts an exclusive-touch view, then that touch is delivered only if that view is the only view tracking a finger in that window. If a finger touches a non-exclusive view, then that touch is delivered only if there is not another finger tracking in an exclusive-touch view.
- **Restricting event delivery to subviews**. A custom `UIView` class can override `hitTest:withEvent:` to restrict the delivery of multi-touch events to its subviews. See "Event-Handling Techniques" (page 148) for a discussion of this technique.

Handling Multi-Touch Events

To handle multi-touch events, your custom `UIView` subclass (or, less frequently, your custom `UIApplication` or `UIWindow` subclass), must implement at least one of the `UIResponder` methods for event handling. The following sections describe these methods, discuss approaches for handling common gestures, show an example of a responder object that handles a complex sequence of multi-touch events, and suggest some techniques for event handling.

The Event-Handling Methods

During a multi-touch sequence, the application dispatches a series of event messages. To receive and handle these messages, the class of a responder object must implement at least one of the following methods declared by `UIResponder`:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

The application sends these messages when there are new or changed touches for a given touch phase:

- It sends the `touchesBegan:withEvent:` message when one or more fingers touch down on the screen.

- It sends the `touchesMoved:withEvent:` message when one or more fingers move.
- It sends the `touchesEnded:withEvent:` message when one or more fingers lift up from the screen.
- It sends the `touchesCancelled:withEvent:` message when the touch sequence is cancelled by a system event, such as an incoming phone call.

Each of these methods is associated with a touch phase (for example, `UITouchPhaseBegan`), which for any `UITouch` object you can find out by evaluating its `phase` property.

Each message that invokes an event-handling method passes in two parameters. The first is a set of `UITouch` objects that represent new or changed touches for the given phase. The second parameter is a `UIEvent` object representing this particular event. From the event object you can get all touch objects for the event (`allTouches`) or a subset of those touch objects filtered for specific views or windows. Some of these touch objects represent touches that have not changed since the previous event message or that have changed but are in different phases.

A responder object frequently handles an event for a given phase by getting one or more of the `UITouch` objects in the passed-in set and then evaluating their properties or getting their locations. (If any of the touch objects will do, it can send the `NSSet` object an `anyObject` message.) One important method is `locationInView:`, which, if passed a parameter of `self`, yields the location of the touch in the responder object's coordinate system (assuming the responder is a `UIView` object and the view passed as a parameter is not `nil`). A parallel method tells you the previous location of the touch (`previousLocationInView:`). Properties of the `UITouch` instance tell you how many taps have been made (`tapCount`), when the touch was created or last mutated (`timestamp`), and what phase it is in (`phase`).

A responder class does not have to implement all three of the event methods listed above. For example, if it is looking for only fingers when they're lifted from the screen, it need only implement `touchesEnded:withEvent:`.

If a responder creates persistent objects while handling events during a multi-touch sequence, it should implement `touchesCancelled:withEvent:` to dispose of those objects when the system cancels the sequence. Cancellation often occurs when an external event—for example, an incoming phone call—disrupts the current application's event processing. Note that a responder object should also dispose of those same objects when it receives the last `touchesEnded:withEvent:` message for a multi-touch sequence. (See "Event-Handling Techniques" (page 148) to find out how to determine the last touch-up in a sequence.)

Handling Single and Multiple Tap Gestures

A very common gesture in iPhone applications is the tap: the user taps an object with his or her finger. A responder object can respond to a single tap in one way, a double-tap in another, and possibly a triple-tap in yet another way. To determine the number of times the user tapped a responder object, you get the value of the `tapCount` property of a `UITouch` object.

The best places to find this value are the methods `touchesBegan:withEvent:` and `touchesEnded:withEvent:`. In many cases, the latter method is preferred because it corresponds to the touch phase in which the user lifts a finger from a tap. By looking for the tap count in the touch-up phase (`UITouchPhaseEnded`), you ensure that the finger is really tapping and not, for instance, touching down and then dragging.

In Listing 7-1, the `touchesEnded:withEvent:` method implementation responds to a double-tap gesture by zooming in on (or out from) the content shown in a scroll view.

Listing 7-1 Handling a double-tap gesture

```
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event
{
    UIScrollView  *scrollView = (UIScrollView*)[self superview];
    UITouch       *touch = [touches anyObject];
    CGSize        size;
    CGPoint       point;

    if([touch tapCount] == 2) {
        if(![_viewController _isZoomed]) {
            point = [touch locationInView:self];
            size = [self bounds].size;
            point.x /= size.width;
            point.y /= size.height;

            [_viewController _setZoomed:YES];

            size = [scrollView contentSize];
            point.x *= size.width;
            point.y *= size.height;
            size = [scrollView bounds].size;
            point.x -= size.width / 2;
            point.y -= size.height / 2;
            [scrollView setContentOffset:point animated:NO];
        }
        else
        [_viewController _setZoomed:NO];
    }
}
```

A complication arises when a responder object wants to handle a single-tap *and* a double-tap gesture in different ways. For example, a single tap might select the object and a double tap might display a view for editing the item that was double-tapped. How is the responder object to know that a single tap is not the first part of a double tap? Here is how a responder object could handle this situation using the event-handling methods just described:

1. In `touchesEnded:withEvent:`, when the tap count is one, the responder object sends itself a `performSelector:withObject:afterDelay:` message. The selector identifies another method implemented by the responder to handle the single-tap gesture; the object for the second parameter is the related `UITouch` object; the delay is some reasonable interval between a single- and a double-tap gesture.

2. In `touchesBegan:withEvent:`, if the tap count is two, the responder object cancels the pending delayed-perform invocation by sending itself a `cancelPreviousPerformRequestsWithTarget:` message. If the tap count is not two, the method identified by the selector in the previous step for single-tap gestures is invoked after the delay.

3. In `touchesEnded:withEvent:`, if the tap count is two, the responder performs the actions necessary for handling double-tap gestures.

Detecting Swipe Gestures

Horizontal and vertical swipes are a simple type of gesture that you can track easily from your own code and use to perform actions. To detect a swipe gesture, you have to track the movement of the user's finger along the desired axis of motion, but it is up to you to determine what constitutes a swipe. In other words, you need to determine if the user's finger moved far enough, if it moved in a straight enough line, and if it went fast enough. You do that by storing the initial touch location and comparing it to the location reported by subsequent touch-moved events.

Listing 7-2 shows some basic tracking methods you could use to detect horizontal swipes in a view. In this example, the view stores the initial location of the touch in a `startTouchPosition` member variable. As the user's finger moves, the code compares the current touch location to the starting location to determine if it is a swipe. If the touch moves too far vertically, it is not considered to be a swipe and is processed differently. If it continues along its horizontal trajectory, however, the code continues processing the event as if it were a swipe. The processing routines could then trigger an action once the swipe had progressed far enough horizontally to be considered a complete gesture. To detect swipe gestures in the vertical direction, you would use similar code but would swap the x and y components.

Listing 7-2 Tracking a swipe gesture in a view

```
#define HORIZ_SWIPE_DRAG_MIN  12
#define VERT_SWIPE_DRAG_MAX    4

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

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = touches.anyObject;
    CGPoint currentTouchPosition = [touch locationInView:self];

    // If the swipe tracks correctly.
    if (fabsf(startTouchPosition.x - currentTouchPosition.x) >=
HORIZ_SWIPE_DRAG_MIN &&
        fabsf(startTouchPosition.y - currentTouchPosition.y) <=
VERT_SWIPE_DRAG_MAX)
    {
        // It appears to be a swipe.
        if (startTouchPosition.x < currentTouchPosition.x)
            [self myProcessRightSwipe:touches withEvent:event];
        else
            [self myProcessLeftSwipe:touches withEvent:event];
    }
    else
    {
        // Process a non-swipe event.
    }
}
```

Handling a Complex Multi-Touch Sequence

Taps and swipes are simple gestures. Handling a multi-touch sequence that is more complicated—in effect, interpreting an application-specific gesture—depends on what the application is trying to accomplish. You may have to track all touches through all phases, recording the touch attributes that have changed and altering internal state appropriately.

The best way to convey how you might handle a complex multi-touch sequence is through an example. Listing 7-3 shows how a custom `UIView` object responds to touches by animating the movement of a "Welcome" placard around the screen as a finger moves it and changing the language of the welcome when the user makes a double-tap gesture. (The code in this example comes from the *MoveMe* sample code project, which you can examine to get a better understanding of the event-handling context.)

Listing 7-3 Handling a complex multi-touch sequence

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [[event allTouches] anyObject];
    // Only move the placard view if the touch was in the placard view
    if ([touch view] != placardView) {
        // On double tap outside placard view, update placard's display string
        if ([touch tapCount] == 2) {
            [placardView setupNextDisplayString];
        }
        return;
    }
    // "Pulse" the placard view by scaling up then down
    // Use UIView's built-in animation
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    CGAffineTransform transform = CGAffineTransformMakeScale(1.2, 1.2);
    placardView.transform = transform;
    [UIView commitAnimations];

    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    transform = CGAffineTransformMakeScale(1.1, 1.1);
    placardView.transform = transform;
    [UIView commitAnimations];

    // Move the placardView to under the touch
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.25];
    placardView.center = [self convertPoint:[touch locationInView:self]
fromView:placardView];
    [UIView commitAnimations];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [[event allTouches] anyObject];

    // If the touch was in the placardView, move the placardView to its location
    if ([touch view] == placardView) {
        CGPoint location = [touch locationInView:self];
        location = [self convertPoint:location fromView:placardView];
```

```
        placardView.center = location;
        return;
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [[event allTouches] anyObject];

    // If the touch was in the placardView, bounce it back to the center
    if ([touch view] == placardView) {
        // Disable user interaction so subsequent touches don't interfere with
 animation
        self.userInteractionEnabled = NO;
        [self animatePlacardViewToCenter];
        return;
    }
}
```

Note: Custom views that redraw themselves in response to events they handle generally should only set drawing state in the event-handling methods and perform all of the drawing in the `drawRect:` method. To learn more about drawing view content, see "Graphics and Drawing" (page 151).

Event-Handling Techniques

Here are some event-handling techniques you can use in your code.

- **Tracking the mutations of UITouch objects**

  In your event-handling code you can store relevant bits of touch state for later comparison with the mutated `UITouch` instance. As an example, say you want to compare the final location of each touch with its original location. In the `touchesBegan:withEvent:` method, you can obtain the original location of each touch from the `locationInView:` method and store those in a `CFDictionary` object using the addresses of the `UITouch` objects as keys. Then, in the `touchesEnded:withEvent:` method you can use the address of each passed-in `UITouch` object to obtain the object's original location and compare that with its current location. (You should use a `CFDictionary` object rather than an `NSDictionary` object; the latter copies its keys, but the `UITouch` class does not adopt the `NSCopying` protocol, which is required for object copying.)

- **Hit-testing for a touch on a subview or layer**

  A custom view can use the `hitTest:withEvent:` method of `UIView` or the `hitTest:` method of `CALayer` to find the subview or layer that is receiving a touch, and handle the event appropriately. The following example detects when an "Info" image in a layer of the custom view is tapped.

  ```
  - (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
      CGPoint location = [[touches anyObject] locationInView:self];
      CALayer *hitLayer = [[self layer] hitTest:[self convertPoint:location
  fromView:nil]];

      if (hitLayer == infoImage) {
          [self displayInfo];
      }
  }
  ```

If you have a custom view with subviews, you need to determine whether you want to handle touches at the subview level or the superview level. If the subviews do not handle touches by implementing `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, or `touchesMoved:withEvent:`, then these messages propagate up the responder chain to the superview. However, because multiple taps and multiple touches are associated with the subviews where they first occurred, the superview won't receive these touches. To ensure reception of all kinds of touches, the superview should override `hitTest:withEvent:` to return itself rather than any of its subviews.

- **Determining when the last finger in a multi-touch sequence has lifted**

  When you want to know when the last finger in a multi-touch sequence is lifted from a view, compare the number of `UITouch` objects in the passed in set with the number of touches for the view maintained by the passed-in `UIEvent` object. For example:

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    if ([touches count] == [[event touchesForView:self] count]) {
        // last finger has lifted....
    }
}
```

Graphics and Drawing

High-quality graphics are an important part of your application's user interface. Providing high-quality graphics not only makes your application look good, but it also makes your application look like a natural extension to the rest of the system. iPhone OS provides two primary paths for creating high-quality graphics in your system: OpenGL or native rendering using Quartz, Core Animation, and UIKit.

The OpenGL frameworks are geared primarily toward game development or applications that require high frame rates. OpenGL is a C-based interface used to create 2D and 3D content on desktop computers. iPhone OS supports OpenGL drawing through the OpenGL ES framework, which is based on the OpenGL ES v1.1 specification and is designed specifically for use on embedded hardware systems. This version differs in many ways from desktop versions of OpenGL, so you should be sure to follow the advice for using it later in this chapter.

For developers who want a more object-oriented drawing approach, iPhone OS provides Quartz, Core Animation, and the graphics support in UIKit. 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 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 iPhone applications, along with specific drawing techniques for each of supported drawing technologies. This chapter also provides tips and guidance on how to optimize your drawing code for the iPhone OS platform.

Quartz Concepts and Terminology

In iPhone OS, all drawing—regardless of whether it involves OpenGL, Quartz, UIKit, or Core Animation—occurs within the confines of a `UIView` object. 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. For applications that draw using OpenGL, once you set up your rendering surface, you use the drawing model specified by OpenGL.

For Quartz, Core Animation, and UIKit, you use the drawing concepts described in the following sections.

The View Drawing Cycle

The basic drawing model for `UIView` objects 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.

Whenever a portion of your view needs to be redrawn, the `UIView` object's built-in drawing code calls its `drawRect:` method. It passes this method a rectangle indicating the portion of your view that needs to be redrawn. You override this method in your custom view subclasses and use it to draw the contents of your view. The first time your view is drawn, the rectangle passed to the `drawRect:` method contains your view's entire visible area. During subsequent calls, however, this rectangle represents only the portion of the view that actually needs to be redrawn. There are several actions that can trigger a view update:

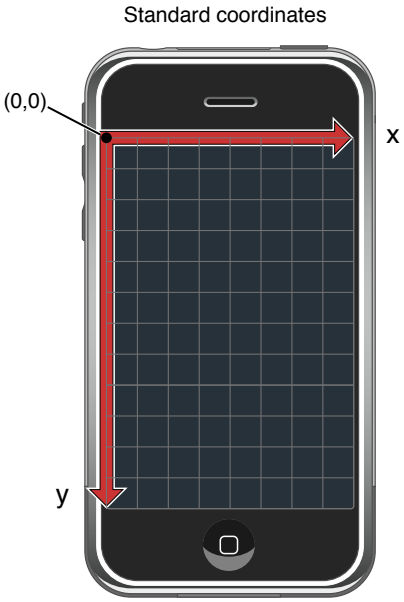
- Moving or removing another view that was partially obscuring your view
- Making a previously hidden view visible again by setting its `hidden` property to `NO`
- Scrolling a view off the screen and then back on
- Explicitly calling the `setNeedsDisplay` or `setNeedsDisplayInRect:` method of your view

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 update your view's content periodically, however, you must determine when to 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.

The Native Coordinate System

In UIKit, the origin of a window or view is located in its top-left corner, and positive coordinate values extend down and to the right of this origin. Figure 8-1 shows this coordinate system in action. When you write your drawing code, you use this coordinate system to specify the location of individual points.

Figure 8-1 The default coordinate system



You can make changes to the default coordinate system by modifying the current transformation matrix. The **current transformation matrix (CTM)** is a mathematical matrix that maps points in your view's coordinate system to points on the device's screen. 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 and is oriented as shown in Figure 8-1 (page 153). You can modify 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 screen.

Modifying the CTM is the standard technique used to draw content in your view because it involves much less work. If you want to draw a 10 x 10 square starting at the point (20, 20) in the current drawing system, you could create a path that moves to (20, 20) and then draw the needed set of lines to complete the square. If you decide later that you want to move that square to the point (10, 10), however, you would have to recreate the path with the new starting point. In fact, you would have to recreate the path every time you changed the origin. Creating paths is a relatively expensive operation, but creating a square whose origin is at (0, 0) and modifying the CTM to match the desired drawing origin is cheap by comparison.

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` opaque type, 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 *CGAffineTransform Reference*.

Graphics Contexts

Before calling your custom `drawRect:` method, the view object automatically configures its drawing environment so that your code can start drawing immediately. As part of this configuration, the `UIView` object creates a graphics context (a `CGContextRef` opaque type) for the current drawing

environment. This graphics context contains the information the drawing system needs to perform any subsequent drawing commands. It defines basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and several others.

You can create custom graphics context objects in situations where you want to draw somewhere other than your view. In Quartz, you primarily do this when you want to capture a series of drawing commands and use them to create an image or a PDF file. To create the context, you use the `CGBitmapContextCreate` or `CGPDFContextCreate` function. Once you have the context, you can pass it to the drawing functions needed to create your content.

When creating custom contexts, the coordinate system for those contexts is different than the native coordinate system used by iPhone OS. Instead of the origin being in the upper-left corner of the drawing surface, it is in the lower-left corner and the axes point up and to the right. The coordinates you specify in your drawing commands must take this into consideration or the resulting image or PDF file may appear wrong when rendered.

Important: Because you use a lower-left origin when drawing into a bitmap or PDF context, you must compensate for that coordinate system when rendering the resulting content into a view. In other words, if you create an image and draw it using the `CGContextDrawImage` function, the image will appear upside down by default. To correct for this, you must invert the y axis of the CTM (by multiplying it by `-1`) and shift the origin from the lower-left corner to the upper-left corner of the view.

If you use a `UIImage` object to wrap a `CGImageRef` you create, you do not need to modify the CTM. The `UIImage` object automatically compensates for the inverted coordinate system of the `CGImageRef` type.

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*.

Points Versus Pixels

The Quartz drawing system uses a vector-based drawing model. Compared to a raster-based drawing model, in which drawing commands operate on individual pixels, drawing commands in Quartz are specified using a fixed-scale drawing space, known as the **user coordinate space**. iPhone OS then maps the coordinates in this drawing space onto the actual pixels of the device. The advantage of this model is that graphics drawn using vector commands continue to look good when scaled up or down using an affine transform.

In order to maintain the precision inherent in a vector-based drawing system, drawing coordinates are specified using floating-point values instead of integers. The use of floating-point values for coordinates makes it possible for you to specify the location of your program's content very precisely. For the most part, you do not have to worry about how those values are eventually mapped to the device's screen.

The user coordinate space is the environment that you use for all of your drawing commands. The units of this space are measured in points. The **device coordinate space** refers to the native coordinate space of the device, which is measured in pixels. By default, one point in user coordinate space is equal to one pixel in device space, which results in 1 point equaling 1/160th of an inch. You should not assume that this 1-to-1 ratio will always be the case, however.

Color and Color Spaces

iPhone OS supports the full range of color spaces available in Quartz; however, most applications should need only the RGB color space. Because iPhone OS is designed to run on embedded hardware and display graphics on a screen, 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.

Supported Image Formats

Table 8-1 lists the image formats supported directly by iPhone OS. Of these formats, the PNG format is the one most recommended for use in your applications.

Table 8-1 Supported image formats

Format	Filename extensions
Portable Network Graphic (PNG)	`.png`
Tagged Image File Format (TIFF)	`.tiff`, `.tif`
Joint Photographic Experts Group (JPEG)	`.jpeg`, `.jpg`
Graphic Interchange Format (GIF)	`.gif`
Windows Bitmap Format (DIB)	`.bmp`, `.BMPf`
Windows Icon Format	`.ico`
Windows Cursor	`.cur`
XWindow bitmap	`.xbm`

Drawing Tips

The following sections provide tips on how to write quality drawing code while ensuring that your application looks appealing to end users.

Deciding When to Use Custom Drawing Code

Depending on the type of application you are creating, it may be possible to use little or no custom drawing code. Although immersive applications typically make extensive use of custom drawing code, utility and productivity applications 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 application would need to use custom drawing code to track the user's drawing commands and a game would be updating the screen constantly to reflect the changing game environment. In those situations, you would need to 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 application's interface is fixed, you can render the interface in advance to one or more image files and display those images at runtime using `UIImageView` objects. You can layer image views with other content as needed to build your interface. For example, you could use `UILabel` objects to display configurable text and include buttons or other controls to provide interactivity.

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 8-2 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 8-2 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.
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`.

Tip	Action
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.
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.
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`.
Minimize graphics state changes while drawing	Changing the graphics state requires effort by the window server. 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.

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 application, keep the following guidelines in mind:

- **Use the PNG format for images.** The PNG format provides high-quality image content and is the preferred image format for iPhone OS. In addition, iPhone OS includes an optimized drawing path for PNG images that is typically more efficient than other formats.
- **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.

Drawing with Quartz and UIKit

Quartz is the general name for the native window server and drawing technology in iPhone OS. 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
- Functions for generating a JPEG or PNG representation of a `UIImage` object
- 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

By the time your `drawRect:` method is called, your view's built-in drawing code has already created and configured a default graphics context for you. 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 8-3 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 8-3 Core graphics functions for modifying graphics state

Graphics state	Core Graphics functions	UIKit alternatives

Current transformation matrix (CTM)	`CGContextRotateCTM`
	`CGContextScaleCTM`
	`CGContextTranslateCTM`
	`CGContextConcatCTM`
		`CGAffineTransform` class

Clipping area	`CGContextClipToRect`	None

Graphics state	Core Graphics functions	UIKit alternatives

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

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

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 Images

iPhone OS 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 8-4 lists some of the usage scenarios and the recommended options for handling them.

Table 8-4 Usage scenarios for images

Scenario	Recommended usage
Display an image as the content of a view	Use a `UIImageView` class to load and 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	Use the `CGBitmapContextCreate` function to create a bitmap graphics context and draw the image contents into it. Then use the `CGBitmapContextCreateImage` function to create a `CGImageRef` from the bitmap context. You can then use this type to initialize a `UIImage` object.
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.

The following example shows how to load an image from your application's bundle. You can subsequently use this image object to initialize a `UIImageView` object, or you can store it and draw it explicitly in your view's `drawRect:` method.

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

To draw an 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. Assuming you stored the previously loaded image in a member variable called `anImage`, the following example draws that image at the point (10, 10) in the view.

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

Creating and Drawing Paths

A path is a description of a 2D geometric scene that uses a sequence of lines and Bézier curves to represent that scene. UIKit includes the `UIRectFrame` and `UIRectFill` functions (among others) for creating 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 functions of the Core Graphics framework.

To create a path, you use the `CGContextBeginPath` function to configure the graphics context to receive path commands. After calling that function, you use other path-related functions to set the path's starting point, draw lines and curves, add rectangles and ellipses, and so on. When you are done specifying the path geometry, you can paint the path directly or create a `CGPathRef` or `CGMutablePathRef` data type to store a reference to that path for later use.

When you want to draw a path in your view, you can stroke it, fill it, or do both. Stroking a path with a function such as `CGContextStrokePath` creates a line centered on the path using the current stroke color. Filling the path with the `CGContextFillPath` function uses the current fill color or fill pattern to fill the area enclosed by the path's line segments.

For more information on how to draw paths, 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*.

Drawing Text

When you need to draw custom text strings in your interface, use the methods of `NSString` to do so. UIKit includes extensions to the basic `NSString` class that allow you to draw strings in your views. These methods allow you to adjust the position of the rendered text precisely and blend it with the rest of your view's content. The methods of this class also let you compute the bounding rectangle for your text in advance based on the desired font and style attributes.

For information about the drawing methods of `NSString`, see *NSString UIKit Additions 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*.

Drawing with OpenGL ES

The **Open Graphics Library (OpenGL)** is a cross-platform C-based interface used to create 2D and 3D content on desktop systems. It is typically used by games developers or anyone needing to perform drawing with high frame rates. You use OpenGL functions to specify primitive structures such as points, lines, and polygons and the textures and special effects to apply to those structures to enhance their appearance. The functions you call send graphics commands to the underlying hardware, where they are then rendered. Because rendering is done mostly in hardware, OpenGL drawing is usually very fast.

OpenGL for Embedded Systems is a pared-down version of OpenGL that is designed for mobile devices and takes advantage of modern graphics hardware. If you want to create OpenGL content for iPhone OS–based devices—that is, iPhone or iPod Touch—you'll use OpenGL ES. The OpenGL ES framework (`OpenGLES.framework`) provided with iPhone OS conforms to the OpenGL ES v1.1 specification. You can find out more about OpenGL ES by reading Polygons In Your Pocket: Introducing OpenGL ES.

This section is designed to get you started writing OpenGL ES applications for iPhone OS–based devices. "Setting Up a Rendering Surface" (page 162) provides step-by-step instructions for creating a surface that you can draw to using OpenGL ES. But before you start writing the OpenGL ES portion of your application, you'll want to read "Implementation Details" (page 166) to learn about the capabilities of iPhone OS–based devices and the specifics of the OpenGL ES implementation in iPhone OS. "Best Practices" (page 164) provides coding guidelines that can help your application perform optimally.

Setting Up a Rendering Surface

The setup for drawing with OpenGL ES is straightforward and requires the same types of tasks you'd perform to draw to surfaces on a Macintosh computer. The primary difference is that you'll use the EAGL API to set up the window surface instead of an API such as CGL or AGL. The EAGL API provides the interface between the OpenGL ES renderer and the windows and views of an iPhone application. (See *OpenGL ES Framework Reference*.)

When you set up your Xcode project, make sure you link to `OpenGLES.framework`. Then, set up a surface for rendering by following these steps:

1. Subclass `UIView` and set up the view appropriately for your iPhone application.
2. Override the `layerClass` method of the `UIView` class so that it returns a `CAEAGLLayer` object rather than a `CALayer` object.

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

3. Get the layer associated with the view by calling the `layer` method of `UIView`.

```
myEAGLLayer = (CAEAGLLayer*)self.layer;
```

4. Set the layer properties.

   For optimal performance, it's recommended that you mark the layer as opaque by setting the `opaque` property provided by the `CALayer` class. See "Best Practices" (page 164).

5. Optionally configure the surface properties of the rendering surface by assigning a new dictionary of values to the `drawableProperties` property of the `CAEAGLLayer` object.

   The EAGL framework lets you specify custom color formats and whether or not the native surface retains its contents after presenting them. You identify these properties in the dictionary using the `kEAGLDrawablePropertyColorFormat` and `kEAGLDrawablePropertyRetainedBacking` keys. For a list of values for these keys, see the `EAGLDrawable` protocol.

6. Create a new `EAGLContext` object to manage the drawing context. You typically create and initialize this object as follows:

   ```
   EAGLContext* myContext = [[EAGLContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES1];
   ```

   If you want to share objects (texture objects, vertex buffer objects, and so forth) between multiple contexts, use the `initWithAPI:sharegroup:` initialization method instead. For each context that should share a given set of objects, pass the same `EAGLSharegroup` object in the *`sharegroup`* parameter.

7. Make your `EAGLContext` object the context for the current thread using the `setCurrentContext:` class method.

   You can have one current context per thread.

8. Create and bind a new render buffer to the `GL_RENDERBUFFER_OES` target. (Typically you would do this in a two-step process using the `glGenRenderbufferOES` function to allocate an unused name and the `glBindRenderbufferOES` functions create and bind the render buffer to that name.)

9. Attach the newly created render buffer target to your view's layer object using the `renderBufferStorage:fromDrawable:` method of your `EAGLContext` object. (The layer provides the underlying storage for the render buffer.) For example, given the context created previously, you would use the following code to bind the buffer to the view's layer (obtained previously and stored in the `myEAGLLayer` variable):

   ```
   [myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];
   ```

   The width, height, and format of the render buffer storage are derived from the bounds and properties of the `CAEAGLLayer` object at the moment you call the `renderbufferStorage:fromDrawable:` method. If you change the layer's bounds later, Core Animation scales the content by default. To avoid scaling, you must recreate the renderbuffer storage by calling `renderbufferStorage:fromDrawable:`.

10. Configure your frame buffer as usual and bind your render buffer to the attach points of your frame buffer.

Best Practices

To create great OpenGL ES applications that run optimally, whenever possible follow the guidelines discussed in this section. You'll also want to read "Implementation Details" (page 166) to see how your code can take advantage of the specific features of the GPU.

General Guidelines

When developing OpenGL ES applications for iPhone OS–based devices:

- Use an EAGL surface that is the same size as size of the screen. Read the `bounds` property of the `UIScreen` class to get the screen size.
- Do not apply any Core Animation transforms to the `CAEAGLLayer` object that contains the EAGL window surface.
- Set the `opaque` property of the `CALayer` class to mark the `CAEAGLLayer` object as opaque.
- Do not place any other Core Animation layers or UIKit views above the `CAEAGLLayer` object that you're rendering to.
- If your application needs to present landscape content, avoid transforming the layer. Instead, set the OpenGL ES state to rotate everything by changing the Model/View/Projection transform, and swapping the width and height arguments to the `glViewport` and `glScissor` functions.
- Limit interactions between OpenGL ES and UIKit or Core Animation rendering. For example, avoid rendering with OpenGL ES while you render notifications, messages, or any other user interface controls provided by UIKit or Core Animation.
- Economize memory usage. iPhone OS–based devices use a shared memory system. Memory used by your application's graphics is not available for the system. For example, after loading GL textures, free your copy of the pixel data if you no longer need to use it. (See "Memory" (page 168))
- To implement transparency, use alpha blending instead of alpha testing. Alpha testing is considerably more expensive than alpha blending.
- Disable unused or unnecessary features. For instance, do not enable lighting or blending if you do not need them.
- Minimize scissor state changes. They are considerably more expensive on OpenGL ES for iPhone OS than they are on OpenGL for Mac OS X.
- Minimize the number of times you call the `EAGLContext` class method `setCurrentContext:` during a rendering frame. Changing the current surface on OpenGL ES for iPhone OS is considerably more expensive than it is on OpenGL for Mac OS X. Rather than using multiple contexts, consider using multiple frame buffer objects instead.
- Operations that depend on completing previous rendering commands (such as `glTexSubImage`, `glCopyTexImage`, `glCopyTexSubImage`, `glReadPixels`, `glFlush`, `glFinish`, or `setCurrentContext:`) can be very expensive if you perform them in the middle of a frame. If you need these operations, perform them at the beginning or end of a frame.

CPU Usage

Respecifying OpenGL ES state can cause the CPU to perform unnecessary work. Use techniques to reduce CPU usage. For example, use a texture atlas. This allows you to perform additional draw calls without changing the texture binding. (Or even better, collapse multiple draw calls into one.) If you create a texture atlas, you also need to sort state calls to avoid rebinding the texture. Otherwise, you won't see a performance benefit.

Vertex Data

When creating geometry:

- Reduce overall geometry by doing such things as using indexed triangle strips and providing only as much detail as the user can see.
- To achieve fine detail in your drawing without increasing the vertex count, consider using DOT3 lighting or textures.
- Memory bandwidth is limited, so use the smallest acceptable type for your data. Specify vertex colors using 4 unsigned byte values. Specify texture coordinates with 2 or 4 unsigned byte or short values instead of floating-point values, if you can.

Textures

To get the best performance with textures:

- Use textures with the smallest size per pixel that you can afford. If possible, use `565` textures instead of `8888`.
- Use compressed textures stored in the PVRTC format. See the specification for the extension `GL_IMG_texture_compression_pvrtc`.
- Use mipmapping with the `LINEAR_MIPMAP_NEAREST` option.
- Create and load all textures prior to rendering. Don't upload or modify textures during a frame. Specifically, avoid calling the functions `glTexSubImage` or `glCopyTexSubImage` in the middle of a frame.
- Use multitexturing rather than applying textures over multiple passes.

Drawing Order

Drawing order is important for hardware that uses tile based deferred rendering. (See "Rendering Path" (page 168).)

- Don't waste CPU time sorting objects front to back. The tile based deferred rendering model used by the GPU makes sorting unnecessary.
- Draw opaque objects first; draw alpha blended objects last.

Lighting

Simplify lighting as much as possible.

- Use the fewest lights possible and the simplest lighting type for your application. For example, consider using directional lights instead of spot lighting, which incurs a higher performance cost.
- If you can, precompute lighting. Static lighting gives better performance than dynamic. You can compute the lighting ahead of time and store the result in textures or color arrays that you can look up later.

Debugging and Tuning

The Instruments application includes an OpenGL ES instrument that you can use to gather information about the runtime behavior of your OpenGL code. In addition, you can set a breakpoint on the `opengl_error_break` symbol in GDB to see when OpenGL errors are generated.

Implementation Details

Understanding the features of the hardware and the specifics of the implementation of OpenGL ES can help you tailor your code to get the best performance. Use the information in this section, along with "Best Practices" (page 164), as you design your OpenGL ES application.

OpenGL ES Implementation

The OpenGL ES implementation in iPhone OS differs from other implementations of OpenGL ES in the following ways:

- The maximum texture size is 1024 x 1024.
- 2D texture targets are supported; other texture targets are not.
- Stencil buffers aren't available.

Hardware Capabilities

When developing any OpenGL application, it's important to check for the functionality that your application uses and have a contingency in place if the hardware doesn't support the particular feature or extension that you want to use. This is true for Macintosh computers and it is even more important when writing OpenGL ES applications for iPhone OS–based devices. It's essential that you understand the capabilities of the specific hardware that you are writing for. Keep in mind that, unlike OpenGL on Macintosh computers, there is no software rendering fallback option for iPhone OS–based devices.

The graphics hardware for iPhone OS–based devices has the following limitations:

- The texture magnification and magnification filters (within a texture level) must match. For example:
  - Supported: `GL_TEXTURE_MAG_FILTER =GL_LINEAR`, `GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR`
  - Supported: `GL_TEXTURE_MAG_FILTER = GL_NEAREST`, `GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_NEAREST`
  - Not Supported: `GL_TEXTURE_MAG_FILTER = GL_NEAREST`, `GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR`

There are a few, rarely used texture environment operations that aren't available:

- If the value of `GL_COMBINE_RGB` is `GL_MODULATE`, only one of the two operands may read from an `GL_ALPHA` source.
- If the value of `GL_COMBINE_RGB` is `GL_INTERPOLATE`, `GL_DOT3_RGB`, or `GL_DOT3_RGBA`, then several combinations of `GL_CONSTANT` and `GL_PRIMARY_COLOR` sources and `GL_ALPHA` operands do not work properly.
- If the value of `GL_COMBINE_RGB` or `GL_COMBINE_ALPHA` is `GL_SUBTRACT`, then `GL_SCALE_RGB` or `GL_SCALE_ALPHA` must be `1.0`.
- If the value of `GL_COMBINE_ALPHA` is `GL_INTERPOLATE` or `GL_MODULATE`, only one of the two sources can be `GL_CONSTANT`.
- The value of `GL_TEXTURE_ENV_COLOR` must be the same for all texture units.

Supported Extensions

These are the OpenGL ES extensions that you can use when developing OpenGL ES applications for iPhone OS–based devices:

- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_texture_mirrored_repeat
- GL_EXT_texture_filter_anisotropic
- GL_EXT_texture_lod_bias
- `GL_IMG_read_format`
- `GL_IMG_texture_compression_pvrtc`
- `GL_IMG_texture_format_BGRA8888`

Memory

OpenGL ES applications should use no more than 24 MB of memory for both textures and surfaces. This 24 MB is not dedicated graphics memory but comes from the main system memory. Because main memory is shared with other iPhone applications and the system, your application should use as little of it as possible. "Best Practices" (page 164) provides several guidelines for ways to use memory economically. In particular, see "General Guidelines" (page 164) and "Vertex Data" (page 165).

Rendering Path

The GPU in the iPhone and iPod touch is a PowerVR MBX Lite. This GPU uses a technique known as **Tile Based Deferred Rendering** (TBDR). When you submit OpenGL ES commands for rendering, TBDR behaves very differently from a streaming renderer. A streaming renderer simply executes rendering commands in order, one after another. In contrast, a TBDR defers any rendering until it accumulates a large number of rendering commands, and then operates on this command list as a single scene. The framebuffer is divided up into a number of tiles, and the scene is drawn once for each tile, each time drawing only the content that is actually visible within that tile. The TBDR approach has several advantages and disadvantages compared to streaming renderers. Understanding these differences will help you write better performing software.

The most significant advantage of TBDR is that it can make much more efficient use of available bandwidth to memory. Constraining rendering to only one tile allows the GPU to more effectively cache the framebuffer, making depth testing blending much more efficient. Otherwise, the memory bandwidth consumed by these framebuffer operations often becomes a significant performance bottleneck.

When using deferred rendering, some operations become more expensive. For example, if you call the function `glTexSubImage` in the middle of a frame, the accumulated command list may include commands from both before and after the call to `glTexSubImage`. This command list needs to reference both the old and new version of the texture image at the same time, forcing the entire texture to be duplicated even if only a small portion of the texture is updated. Duplication can make functions such as `glTexSubImage` significantly more expensive on a deferred renderer than a streaming renderer.

The PowerVR GPU relies on more than just TBDR to optimize performance; it performs hidden surface removal before fragment processing. If the GPU determines that a pixel won't be visible, it discards the pixel without performing texture sampling or fragment color calculations. Removing hidden pixels can significantly improve performance for scenes that have obscured content. To gain the most benefit from this feature, you should try to draw as much of the scene with opaque content as possible and minimize use of blending and alpha testing.

For more information on exactly how these features are implemented and how your application can best take advantage of them, see PowerVR Technology Overview and PowerVR MBX 3D Application Development Recommendations.

Simulator Capabilities

The iPhone simulator includes a complete and conformant implementation of OpenGL ES 1.1 that you can use for your application development. This implementation differs in a few ways from the implementation found in iPhone OS–based devices. In particular, the simulator does not have the same limitations regarding texture magnification filters or texture environment operations that are described in "Hardware Capabilities" (page 166). In addition, the simulator supports antialiased lines while iPhone OS–based devices do not.

Important: It is important to understand that the rendering performance of OpenGL ES in the simulator has no relation to the performance of OpenGL ES on an actual device. The simulator provides an optimized software rasterizer that takes advantage of the vector processing capabilities of your Macintosh computer. As a result, your OpenGL ES code may run faster or slower in Mac OS X (depending on your computer and what you are drawing) than on an actual device. Therefore, you should always profile and optimize your drawing code on a real device and not assume that the simulator reflects real-world performance.

The following sections provide additional details about the OpenGL ES support available in the iPhone simulator.

Supported Extensions

The iPhone simulator supports all of the OpenGL ES 1.1 core functionality and most of the extensions supported by iPhone OS–based devices. The following extensions are not supported by the simulator, however:

- GL_OES_draw_texture
- GL_OES_matrix_palette
- GL_EXT_texture_filter_anisotropic
- `GL_IMG_texture_compression_pvrtc`

For a list of the extensions supported by the hardware, see "Supported Extensions" (page 167).

Memory

On a device, OpenGL ES applications can use no more than 24 MB of memory for both textures and surfaces. The simulator does not enforce this limit. As a result, your code can allocate as much memory as your computer's rendering hardware supports. Be sure to keep track of the size of your assets during development.

Rendering Path

In contrast to the Tile Based Deferred Rendering technique used in devices, the simulator's software rasterizer uses a traditional streaming model for OpenGL ES commands. Objects are transformed and rendered immediately as you specify them. Consequently, the performance of some operations can differ significantly from that on actual devices.

As with any two different implementations of OpenGL ES, there may be small differences between the pixels rendered by the simulator and those rendered by the device. For example, OpenGL ES allows some calculations, such as color interpolation and texture mipmap filtering, to be approximated. In general, the two implementations will produce similar results, but do not rely on them to be bit-for-bit identical.

For More Information

You may want to consult these resources as you develop OpenGL ES applications for iPhone OS–based devices:

- OpenGL ES 1.X Specification is the official definition of this technology provided by the Khronos Group. You'll also find other useful information on this website.
- PowerVR MBX OpenGL ES 1.x SDK page provides information about the specific OpenGL ES implementation supported by the PowerVR MBX graphics hardware.
- OpenGL ES 1.1 Reference Pages provides a complete reference to OpenGL ES specification, indexed alphabetically as well as by theme.
- *OpenGL ES Framework Reference* describes the functions and constants that provide the interface between OpenGL ES and the iPhone user interface.

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 applications can benefit from using Core Animation in some form in iPhone OS. Animations provide feedback to the user about what is happening. For example, when the user navigates through the Settings application, 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 application.

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" (page 131).

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 iPhone OS. 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*.

Audio and Video Technologies

iPhone OS supports audio features in your application through the Core Audio and OpenAL frameworks, and provides video playback support using the Media Player framework. Core Audio provides an advanced interface for playing, recording, and manipulating sound and for parsing streamed audio. If you are a game developer and already have code that takes advantage of OpenAL, you can use your code in iPhone OS.

This chapter provides a quick introduction to audio in iPhone OS. For a more in-depth description, see *Core Audio Overview*.

Using Sound in iPhone OS

Core Audio offers a rich set of tools for working with sound in your application. These tools are arranged into three frameworks: The Audio Toolbox framework, the Audio Unit framework, and the Core Audio framework (which is not an umbrella framework but rather provides data types used by all Core Audio services).

In addition to Core Audio, you can use the OpenAL framework in iPhone OS to provide positional audio playback in your application. OpenAL 1.1 support in iPhone OS is built on top of Core Audio.

Together, and in concert with other iPhone OS frameworks, the Core Audio interfaces enable you to:

- Play audio (see "Playing Short Sounds Using System Sound Services" (page 174), "Playing Sounds with Control Using Audio Queue Services" (page 176), and "Playing Sounds with Positioning Using OpenAL" (page 179))
- Record audio (see "Recording Audio" (page 179))
- Parse audio streams from a network (see "Parsing Streamed Audio" (page 180))
- Mix sounds, control levels, and position sounds in a stereo field (see "Mixing and Processing Sounds" (page 180))
- Trigger vibration for tactile feedback (iPhone only) (see "Triggering Vibration" (page 181)

You select among the various Core Audio APIs based on the needs of your application. This chapter describes how to do all of these sound-related tasks in iPhone OS, and directs you to further information.

Be sure to read the following two sections as well:

- "Tips for Manipulating Audio" (page 181) offers important guidelines for success with using sound in iPhone OS.
- "Preferred Audio Formats in iPhone OS" (page 182) lists the audio formats and file formats to use for best performance and best user experience.

Audio Sessions

In iPhone OS, your application runs on a device that sometimes has more important things to do, such as take a phone call. If your application is playing sound and a call comes in, or an event alarm comes due, the iPhone needs to do the right thing. An **audio session** is the iPhone OS software abstraction that represents the audio behavior of your application, in context, on an iPhone or iPod touch.

Every iPhone OS application that uses audio should adopt Audio Session Services. To learn how, read Core Audio Essentials in *Core Audio Overview*.

Playing Short Sounds Using System Sound Services

For playback, Core Audio offers two mechanisms, both available in the Audio Toolbox framework:

- To play short sound files of under 30 seconds duration when you do not need level control or other control, use System Sound Services, as described in this section.
- To play longer sound files, to exert control over playback including level adjustments, or to play multiple sounds simultaneously, use Audio Queue Services, described in the next section.

Use the `AudioServicesPlaySystemSound` function from System Sound Services to very simply play short sound files. The simplicity carries with it a few restrictions. Your sound files must be:

- Shorter than 30 seconds in duration
- In linear PCM or IMA/ADPCM (IMA4) format
- Packaged in a `.caf`, `.aif`, or `.wav` file

In addition, when you use the `AudioServicesPlaySystemSound` function:

- Sounds play at the current system audio level
- Sounds play immediately
- Looping and stereo positioning are unavailable

Note: System-supplied alert sounds and system-supplied user-interface sound effects are not available in iPhone OS. For example, using the `kSystemSoundID_UserPreferredAlert` constant as a parameter to the `AudioServicesPlayAlertSound` function will not play anything.

To play a sound with the `AudioServicesPlaySystemSound` function, you first register your sound file as a system sound by creating a sound ID object. You can then play the sound. In typical use, which includes playing a sound occasionally or repeatedly, retain the sound ID object until your

application quits. If you know that you will use a sound only once—for example, in the case of a startup sound—you can destroy the sound ID object immediately after playing the sound, freeing memory.

When you need lowest-latency playback, for example when playing sound effects in games or playing time-critical user feedback sounds in applications, make use of the IO audio unit directly. (Audio units are plug-ins that can add audio features to your application at run time. See "Audio Unit Support in iPhone OS" (page 181).) The most straightforward way to use the IO unit is to use OpenAL, which employs this audio unit for low-latency playback. See "Playing Sounds with Positioning Using OpenAL" (page 179).

Listing 9-1 shows a minimal program that uses the interfaces in System Sound Services to play a sound. The sound completion callback, and the call that installs it, are primarily useful when you want to free memory after playing a sound in cases where you know you will not be playing the sound again.

Listing 9-1 Playing a short sound

```
#include <AudioToolbox/AudioToolbox.h>
#include <CoreFoundation/CoreFoundation.h>

// Define a callback to be called when the sound is finished
// playing. Useful when you need to free memory after playing.
static void MyCompletionCallback (
    SystemSoundID  mySSID,
    void * myURLRef
) {
        AudioServicesDisposeSystemSoundID (mySSID);
        CFRelease (myURLRef);
        CFRunLoopStop (CFRunLoopGetCurrent());
}

int main (int argc, const char * argv[]) {
    // Set up the pieces needed to play a sound.
    SystemSoundID    mySSID;
    CFURLRef        myURLRef;
    myURLRef = CFURLCreateWithFileSystemPath (
        kCFAllocatorDefault,
        CFSTR ("../../ComedyHorns.aif"),
        kCFURLPOSIXPathStyle,
        FALSE
    );

    // create a system sound ID to represent the sound file
    OSStatus error = AudioServicesCreateSystemSoundID (myURLRef, &mySSID);

    // Register the sound completion callback.
    // Again, useful when you need to free memory after playing.
    AudioServicesAddSystemSoundCompletion (
        mySSID,
        NULL,
        NULL,
        MyCompletionCallback,
        (void *) myURLRef
    );

    // Play the sound file.
```

```
    AudioServicesPlaySystemSound (mySSID);

    // Invoke a run loop on the current thread to keep the application
    // running long enough for the sound to play; the sound completion
    // callback later stops this run loop.
    CFRunLoopRun ();
    return 0;
}
```

Playing Sounds with Control Using Audio Queue Services

The preceding examples using System Sound Services are sufficient for playing a short sound when you do not need to control playback. Use Audio Queue Services instead if you want to do any of the following:

- Play a sound that is longer than 30 seconds in duration
- Precisely schedule when a sound plays
- Play multiple sounds simultaneously
- Loop a sound
- Control relative playback level
- Position a sound in a stereo field

Audio Queue Services lets you play sound in any audio format available in iPhone OS. These formats include:

- AAC
- AMR (Adaptive Multi-Rate, a format for speech)
- Apple Lossless (ALAC)
- iLBC (internet Low Bitrate Codec, another format for speech)
- IMA/ADPCM (IMA-4)
- linear PCM
- µ-law and a-law
- MP3

This section provides an overview of using Audio Queue Services for playback. For more information, see *Audio Queue Services Programming Guide* and *Audio Queue Services Reference*. For sample code, see the *SpeakHere* sample in the iPhone Dev Center. (For a Mac OS X implementation, see the AudioQueueTools project available in the Core Audio SDK. When you install Xcode tools in Mac OS X, the AudioQueueTools project is available at `/Developer/Examples/CoreAudio/SimpleSDK/AudioQueueTools`.)

Creating an Audio Queue Object

To create an audio queue object for playback, perform these three steps:

1. Create a data structure to manage information needed by the audio queue, such as the audio format for the data you want to play.
2. Define a callback function for managing audio queue buffers. The callback uses Audio File Services to read the file you want to play.
3. Instantiate the playback audio queue using the `AudioQueueNewOutput` function.

Listing 9-2 illustrates these steps:

Listing 9-2 Creating an audio queue object

```
static const int kNumberBuffers = 3;
// Create a data structure to manage information needed by the audio queue
struct myAQStruct {
    AudioFileID                   mAudioFile;
    CAStreamBasicDescription      mDataFormat;
    AudioQueueRef                 mQueue;
    AudioQueueBufferRef           mBuffers[kNumberBuffers];
    SInt64                        mCurrentPacket;
    UInt32                        mNumPacketsToRead;
    AudioStreamPacketDescription  *mPacketDescs;
    bool                          mDone;
};
// Define a playback audio queue callback function
static void AQTestBufferCallback(
    void                  *inUserData,
    AudioQueueRef         inAQ,
    AudioQueueBufferRef   inCompleteAQBuffer
) {
    myAQStruct *myInfo = (myAQStruct *)inUserData;
    if (myInfo->mDone) return;
    UInt32 numBytes;
    UInt32 nPackets = myInfo->mNumPacketsToRead;

    AudioFileReadPackets (
        myInfo->mAudioFile,
        false,
        &numBytes,
        myInfo->mPacketDescs,
        myInfo->mCurrentPacket,
        &nPackets,
        inCompleteAQBuffer->mAudioData
    );
    if (nPackets > 0) {
        inCompleteAQBuffer->mAudioDataByteSize = numBytes;
        AudioQueueEnqueueBuffer (
            inAQ,
            inCompleteAQBuffer,
            (myInfo->mPacketDescs ? nPackets : 0),
            myInfo->mPacketDescs
        );
        myInfo->mCurrentPacket += nPackets;
    } else {
        AudioQueueStop (
            myInfo->mQueue,
            false
```

```
        );
        myInfo->mDone = true;
    }
}
// Instantiate an audio queue object
AudioQueueNewOutput (
    &myInfo.mDataFormat,
    AQTestBufferCallback,
    &myInfo,
    CFRunLoopGetCurrent(),
    kCFRunLoopCommonModes,
    0,
    &myInfo.mQueue
);
```

Controlling Playback Level

Audio queue objects give you two ways to control playback level.

To set playback level directly, use the `AudioQueueSetParameter` function with the `kAudioQueueParam_Volume` parameter, as shown in Listing 9-3. Level change takes effect immediately.

Listing 9-3 Setting playback level directly

```
Float32 volume = 1;
AudioQueueSetParameter (myAQstruct.audioQueueObject, kAudioQueueParam_Volume,
volume);
```

You can also set playback level for an audio queue buffer, using the `AudioQueueEnqueueBufferWithParameters` function. This lets you assign audio queue settings that are, in effect, carried by an audio queue buffer as you enqueue it. Such changes take effect when the audio queue buffer begins playing.

In both cases, level changes for an audio queue remain in effect until you change them again.

Indicating Playback Level

You can obtain the current playback level from an audio queue object by:

1. Enabling metering for the audio queue object by setting its `kAudioQueueProperty_EnableLevelMetering` property to `true`
2. Querying the audio queue object's `kAudioQueueProperty_CurrentLevelMeter` property

The value of this property is an array of `AudioQueueLevelMeterState` structures, one per channel. Listing 9-4 shows this structure:

Listing 9-4 The AudioQueueLevelMeterState structure

```
typedef struct AudioQueueLevelMeterState {
    Float32     mAveragePower;
    Float32     mPeakPower;
};  AudioQueueLevelMeterState;
```

Playing Multiple Sounds Simultaneously

To play multiple sounds simultaneously, create one playback audio queue object for each sound. For each audio queue, schedule the first buffer of audio to start at the same time using the `AudioQueueEnqueueBufferWithParameters` function.

Audio format is critical when you play sounds simultaneously on iPhone or iPod touch. To play simultaneous sounds of the same use linear PCM or IMA4 audio. This restriction maximizes CPU performance and battery life.

The following list describes how iPhone OS supports audio formats for individual or multiple playback:

- **Linear PCM and IMA/ADPCM (IMA4) audio** You can play multiple linear PCM or IMA4 format sounds simultaneously in iPhone OS without incurring CPU resource problems. The same is true for the AMR and iLBC speech formats, and for the µ-law and a-law formats.
- **AAC, MP3, and Apple Lossless (ALAC) audio** Playback for AAC, MP3, and Apple Lossless (ALAC) sounds uses efficient hardware-based decoding on iPhone and iPod touch, but these codecs all share a single hardware path. You can play only a single instance of one of these formats at a time.

Playing Sounds with Positioning Using OpenAL

The open-sourced OpenAL audio API, available in iPhone OS in the OpenAL framework, provides an interface optimized for positioning sounds during playback. Playing, positioning, and moving sounds is very simple when you use OpenAL. OpenAL also lets you mix sounds. In addition, OpenAL uses Core Audio's IO audio unit directly, resulting in the lowest latency playback.

For all of the reasons, OpenAL is your best choice for playing sound effects in game applications on iPhone and iPod touch. However, OpenAL is also a good choice for general iPhone OS application audio needs.

OpenAL 1.1 support in iPhone OS is built on top of Core Audio. The iPhone implementation of OpenAL provides a panning model that maximizes performance and battery life.

For OpenAL documentation, see the OpenAL website at http://openal.org.

Recording Audio

Core Audio provides support in iPhone OS for recording audio using Audio Queue Services. This interface does the work of connecting to the audio hardware, managing memory, and employing codecs as needed. You can record audio in the following formats:

- Apple Lossless
- iLBC (internet Low Bitrate Codec, for speech)
- IMA/ADPCM (IMA-4)
- linear PCM
- µ-law and a-law

To record audio, your application configures the recording session, instantiates a recording audio queue object, and provides a callback function. The callback stores the audio data in memory for immediate use or writes it to a file for long-term storage.

Recording takes place at a fixed level in iPhone OS. The system records from the audio source that the user has chosen—namely, the built-in microphone or, if connected, the headset microphone.

Just as with playback, you can obtain the current recording level from an audio queue object by querying its `kAudioQueueProperty_CurrentLevelMeter` property. See "Indicating Playback Level" (page 178).

For detailed examples of how to use Audio Queue Services to record audio, see Recording Audio in *Audio Queue Services Programming Guide*. For sample code, see the *SpeakHere* sample in the iPhone Dev Center.

Parsing Streamed Audio

To play streamed audio content, such as from a network connection, use Audio File Stream Services in concert with Audio Queue Services. You can play streamed audio in any of the formats supported by Audio Queue Services, as listed in "Playing Sounds with Control Using Audio Queue Services" (page 176).

For best performance, network streaming applications should use data from Wi-Fi connections only. iPhone OS lets you determine which networks are reachable and available through its System Configuration framework and its `SCNetworkReachability.h` interfaces. For sample code, see the *Reachability* project in the iPhone OS Reference Library.

Audio File Stream Services works by enabling you to parse audio packets and metadata from a network stream. To connect to a network stream you can use interfaces from Core Foundation in iPhone OS, such as `CFHTTPMesaage` in the CF Network interface. You parse the network packets to recover audio packets using Audio File Stream Services. You then buffer the audio packets and send them to a playback audio queue object.

Audio File Stream Services relies on interfaces from Audio File Services, such as the `AudioFramePacketTranslation` structure and the `AudioFilePacketTableInfo` structure. You can also use Audio File Stream Services to read files from disk.

For more information on using streams, refer to *Audio File Stream Services Reference* and *Audio File Services Reference*. For sample code, see the AudioFileStream sample project located in the <*Xcode*>`/Examples/CoreAudio/Services/` directory, where <*Xcode*> is the path to your developer tools directory.

Mixing and Processing Sounds

iPhone OS offers two interfaces for mixing sounds. You can use OpenAL (as described in "Playing Sounds with Positioning Using OpenAL" (page 179)) or you can use the 3D Mixer audio unit.

The 3D Mixer unit combines sounds into a single stream which you then send to the IO unit. For sample code, see the *oalTouch* sample in the iPhone Dev Center.

Audio Unit Support in iPhone OS

iPhone OS provides a core set of audio plug-ins, known as *audio units*, that you can use in any application. The interfaces in the Audio Unit framework enable you to open, connect, and use audio units in iPhone OS. You cannot, however, create your own audio units for iPhone OS.

"Supported audio units" lists the audio units provided in iPhone OS.

Table 9-1 Supported audio units

Audio unit	Description
3D Mixer unit	The 3D Mixer unit, of type `AU3DMixerEmbedded`, lets you mix multiple audio streams.
Converter unit	The Converter unit, of type `AUConverter`, lets you convert audio data from one format to another.
IO unit	The IO unit, of type `RemoteIO`, lets you connect to audio input and output hardware and supports realtime I/O.
iPod EQ unit	The iPod EQ unit, of type `AUiPodEQ`, provides a simple, preset-based equalizer you can use in your application.
Stereo Mixer unit	The Stereo Mixer unit allows any number of mono or stereo inputs, each of which can be 16-bit linear or 8.24-bit fixed-point PCM.

Triggering Vibration

Applications running on iPhone—but not the iPod touch—can trigger vibration using System Sound Services. You specify the vibrate option with the `kSystemSoundID_Vibrate` identifier. To trigger it, use the `AudioServicesPlaySystemSound` function, as shown here.

```
#import <AudioToolbox/AudioToolbox.h>
#import <UIKit/UIKit.h>
- (void)vibratePhone
{
    AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
}
```

If your application is running on an iPod touch, this code will produce a brief buzzing sound using the device's piezoelectric transducer.

Tips for Manipulating Audio

Table 9-2 lists some basic tips to remember when manipulating audio content in iPhone OS.

Table 9-2 Audio tips

Tip	Action
Use compressed audio appropriately	For AAC, MP3, and Apple Lossless audio, decoding takes place in hardware and, while efficient, is limited to one audio file at a time. If you need to play multiple sounds simultaneously, store those sounds using the IMA4 or linear PCM format.
Convert to the data format and file format you need	The `afconvert` tool in Mac OS X lets you convert to a wide range of audio data formats and file types. See "Preferred Audio Formats in iPhone OS" (page 182) and the `afconvert` man page.
Evaluate audio memory issues	When playing sound with Audio Queue Services, you write a callback that sends short segments of audio data to audio queue buffers. In some cases, loading an entire sound file to memory for playback, which minimizes disk access, is best. In other cases, loading just enough data at a time to keep the buffers full is best. Test and evaluate which strategy is best for your application.
Reduce audio file sizes by limiting sample rates and bit depths	Sample rates and the number of bits per sample have a direct impact on the size of your uncompressed audio. If you need to play many such sounds, consider reducing these values to reduce the memory footprint of the audio data. For example, rather than use 44.1 kHz sampling rate for sound effects, you could use a 32 kHz (or possibly lower) sample rate and still provide reasonable quality.
Pick the appropriate technology	Use Core Audio's System Audio Services to play short sounds when you don't need scheduling or level control. Use OpenAL when you want a convenient, high-level interface for positioning sounds in a stereo field. To parse audio packets from a file or a network stream, use Audio File Stream Services. For all other audio applications, use Audio Queue Services.
Code for low latency	For the lowest possible playback latency, use OpenAL or use the IO Remote audio unit directly.

Preferred Audio Formats in iPhone OS

For uncompressed (highest quality) audio, use 16-bit, little endian, linear PCM audio data packaged in a CAF file. You can convert an audio file to this format in Mac OS X using the `afconvert` command-line tool.

```
/usr/bin/afconvert -f caff -d LEI16 {INPUT} {OUTPUT}
```

The `afconvert` tool lets you convert to a wide range of audio data formats and file types. See the `afconvert` man page, and enter `afconvert -h` at a shell prompt, for more information.

For compressed audio when playing one sound at a time, use the AAC format packaged in a CAF or m4a file.

For less memory usage when you need to play multiple sounds simultaneously, use IMA/ADPCM (IMA4) compression. This reduces file size but entails minimal CPU impact during decompression. As with linear PCM data, package IMA4 data in a CAF file.

Playing Video Files

iPhone OS supports the ability to play back video files directly from your application using the Media Player framework (`MediaPlayer.framework`). Video playback is supported in full screen mode only and can be used by game developers who want to play cut scene animations or by other developers who want to play media files. When you start a video from your application, the media player interface takes over, fading the screen to black and then fading in the video content. You can play a video with or without user controls for adjusting playback; enabling some or all of these controls (shown in Figure 9-1) gives the user the ability to change the volume, change the playback point, or start and stop the video. If you disable all of these controls, the video plays until completion.

Figure 9-1 Media player interface with transport controls

To initiate video playback, you must know the URL of the file you want to play. For files your application provides, this would typically be a pointer to a file in your application's bundle; however, it can also be a pointer to a file on a remote server. You use this URL to instantiate a new instance of the `MPMoviePlayerController` class. This class presides over the playback of your video file and manages user interactions, such user taps in the transport controls (if shown). To initiate playback, simply call the `play` method of the controller.

Listing 9-5 shows a sample method that plays back the video at the specified URL. The play method is an asynchronous call that returns control to the caller while the movie plays. The movie controller loads the movie in a full-screen view, and animates the movie into place on top of the application's existing content. When playback is finished, the movie controller sends a notification to the object, which releases the movie controller now that it is no longer needed.

Listing 9-5 Playing full screen movies.

```
-(void)playMovieAtURL:(NSURL*)theURL
{
    MPMoviePlayerController* theMovie = [[MPMoviePlayerController alloc]
initWithContentURL:theURL];

    theMovie.scalingMode = MPMovieScalingModeAspectFill;
    theMovie.moveControlMode = MPMovieControlModeHidden;
```

```
    // Register for the playback finished notification.
    [[NSNotificationCenter defaultCenter] addObserver:self
                selector:@selector(myMovieFinishedCallback:)
                name:MPMoviePlayerPlaybackDidFinishNotification
                object:theMovie];

    // Movie playback is asynchronous, so this method returns immediately.
    [theMovie play];
}

// When the movie is done, release the controller.
-(void)myMovieFinishedCallback:(NSNotification*)aNotification
{
    MPMoviePlayerController* theMovie = [aNotification object];

    [[NSNotificationCenter defaultCenter] removeObserver:self
                name:MPMoviePlayerPlaybackDidFinishNotification
                object:theMovie];

    // Release the movie instance created in playMovieAtURL:
    [theMovie release];
}
```

For more information about the classes of the Media Player framework, see *Media Player Framework Reference*. For a list of supported video formats, see "Video Technologies" (page 36).

Device Features

iPhone OS supports a variety of features that make the mobile computing experience compelling for users. Through iPhone OS, your applications can access hardware features, such as the accelerometers and camera, and software features, such as the user's photo library. The following sections describe these features and show you how to integrate them into your own applications.

Accessing Accelerometer Events

An accelerometer measures changes in velocity over time along a given linear path. The iPhone and iPod touch each contain three accelerometers, one along each of the primary axes of the device. This combination of accelerometers lets you detect movement of the device in any direction. You can use this data to track both sudden movements in the device and the device's current orientation relative to gravity.

Every application has a single `UIAccelerometer` object that can be used to receive acceleration data. You get the instance of this class using the `sharedAccelerometer` class method of `UIAccelerometer`. Using this object, you set the desired reporting interval and a custom delegate to receive acceleration events. You can set the reporting interval to be as small as 10 milliseconds, which corresponds to a 100 Hz update rate, although most applications can operate sufficiently with a larger interval. As soon as you assign your delegate object, the accelerometer starts sending it data. Thereafter, your delegate receives data at the requested update interval.

Listing 10-1 shows the basic steps for configuring the accelerometer. In this example, the update frequency is 50 Hz, which corresponds to an update interval of 20 milliseconds. The `myDelegateObject` is a custom object that you define; it must support the `UIAccelerometerDelegate` protocol, which defines the method used to receive acceleration data.

Listing 10-1 Configuring the accelerometer

```
#define kAccelerometerFrequency        50 //Hz
-(void)configureAccelerometer
{
    UIAccelerometer*  theAccelerometer = [UIAccelerometer sharedAccelerometer];
    theAccelerometer.updateInterval = 1 / kAccelerometerFrequency;

    theAccelerometer.delegate = self;
    // Delegate events begin immediately.
}
```

The shared accelerometer delivers event data at regular intervals to your delegate's `accelerometer:didAccelerate:` method, shown in Listing 10-2. You can use this method to process the accelerometer data however you want. In general it is recommended that you use some sort of filter to isolate the component of the data in which you are interested.

Listing 10-2 Receiving an accelerometer event

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
    UIAccelerationValue x, y, z;
    x = acceleration.x;
    y = acceleration.y;
    z = acceleration.z;

    // Do something with the values.
}
```

To stop the delivery of acceleration events, set the delegate of the shared `UIAccelerometer` object to nil. Setting the

The acceleration data you receive in your delegate method represents the instantaneous values reported by the accelerometer hardware. Even when a device is completely at rest, the values reported by this hardware can fluctuate slightly. When using these values, you should be sure to account for these fluctuations by averaging out the values over time or by calibrating the data you receive. For example, the Bubble Level sample application provides controls for calibrating the current angle against a known surface. Subsequent readings are then reported relative to the calibrated angle. If your own code requires a similar level of accuracy, you should also include some sort of calibration option in your user interface.

Choosing an Appropriate Update Interval

When configuring the update interval for acceleration events, it is best to choose an interval that minimizes the number of delivered events while still meeting the needs of your application. Few applications need acceleration events delivered 100 times a second. Using a lower frequency prevents your application from running as often and can therefore improve battery life. Table 10-1 lists some typical update frequencies and what you can do with the acceleration data generated at that frequency.

Table 10-1 Common update intervals for acceleration events

Event frequency (Hz)	Usage
10-20	Suitable for use in determining the vector representing the current orientation of the device.
30-60	Suitable for games and other applications that use the accelerometers for real-time user input.
70-100	Suitable for applications that need to detect high-frequency motion. For example, you might use this interval to detect the user hitting the device or shaking it very quickly.

Isolating the Gravity Component From Acceleration Data

If you are using the accelerometer data to detect the current orientation of a device, you need to be able to filter out the portion of the acceleration data that is caused by gravity from the portion that is caused by motion of the device. To do this, you can use a low-pass filter to reduce the influence of sudden changes on the accelerometer data. The resulting filtered values would then reflect the more constant effects of gravity.

Listing 10-3 shows a simplified version of a low-pass filter. This example uses a low-value filtering factor to generate a value that uses 10 percent of the unfiltered acceleration data and 90 percent of the previously filtered value. The previous values are stored in the `accelX`, `accelY`, and `accelZ` member variables of the class. Because acceleration data comes in regularly, these values settle out quickly and respond slowly to sudden but short-lived changes in motion.

Listing 10-3 Isolating the effects of gravity from accelerometer data

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // Use a basic low-pass filter to keep only the gravity component of each
axis.
    accelX = (acceleration.x * kFilteringFactor) + (accelX * (1.0 -
kFilteringFactor));
    accelY = (acceleration.y * kFilteringFactor) + (accelY * (1.0 -
kFilteringFactor));
    accelZ = (acceleration.z * kFilteringFactor) + (accelZ * (1.0 -
kFilteringFactor));

   // Use the acceleration data.
}
```

Isolating Instantaneous Motion From Acceleration Data

If you are using accelerometer data to detect just the instant motion of a device, you need to be able to isolate sudden changes in movement from the constant effect of gravity. You can do that with a high-pass filter.

Listing 10-4 shows a simplified high-pass filter computation. The acceleration values from the previous event are stored in the `accelX`, `accelY`, and `accelZ` member variables of the class. This example computes the low-pass filter value and then subtracts it from the current value to obtain just the instantaneous component of motion.

Listing 10-4 Getting the instantaneous portion of movement from accelerometer data

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // Subtract the low-pass value from the current value to get a simplified
high-pass filter
    accelX = acceleration.x - ( (acceleration.x * kFilteringFactor) + (accelX
* (1.0 - kFilteringFactor)) );
```

```
    accelY = acceleration.y - ( (acceleration.y * kFilteringFactor) + (accelY
* (1.0 - kFilteringFactor)) );
    accelZ = acceleration.z - ( (acceleration.z * kFilteringFactor) + (accelZ
* (1.0 - kFilteringFactor)) );

   // Use the acceleration data.
}
```

Getting the Current Device Orientation

If you need to know only the general orientation of the device, and not the exact vector of orientation, you should use the methods of the `UIDevice` class to retrieve that information. Using the UIDevice interface is simpler and does not require you to calculate the orientation vector yourself.

Before getting the current orientation, you must tell the `UIDevice` class to begin generating device orientation notifications by calling the `beginGeneratingDeviceOrientationNotifications` method. Doing so turns on the accelerometer hardware (which may otherwise be off to conserve power).

Shortly after enabling orientation notifications, you can get the current orientation from the `orientation` property of the shared `UIDevice` object. You can also register to receive `UIDeviceOrientationDidChangeNotification` notifications, which are posted whenever the general orientation changes. The device orientation is reported using the `UIDeviceOrientation` constants, which indicate whether the device is in landscape or portrait mode or whether the device is face up or face down. These constants indicate the physical orientation of the device and need not correspond to the orientation of your application's user interface.

When you no longer need to know the orientation of the device, you should always disable orientation notifications by calling the `endGeneratingDeviceOrientationNotifications` method of `UIDevice`. Doing so gives the system the opportunity to disable the accelerometer hardware if it is not in use elsewhere.

Getting the User's Current Location

The Core Location framework lets you locate the current position of the device and use that information in your application. The framework takes advantage of the device's built-in hardware, triangulating a position fix from available signal information. It then reports the location to your code and occasionally updates that position information as it receives new or improved signals.

If you do use the Core Location framework, be sure to do so sparingly and to configure the location service appropriately. Gathering location data involves powering up the onboard radios and querying the available cell towers, WiFi hotspots, or GPS satellites, which can take several seconds. In addition, requesting more accurate location data may require the radios to remain on for a longer period of time. Leaving this hardware on for extended periods of time can drain the device's battery. Given that position information does not change too often, it is usually sufficient to establish an initial position fix and then acquire updates periodically after that. If you are sure you need regular position updates, you can also configure the service with a minimum threshold distance to minimize the number of position updates your code must process.

To retrieve the user's current location, create an instance of the `CLLocationManager` class and configure it with the desired accuracy and threshold parameters. To begin receiving location notifications, assign a delegate to the object and call the `startUpdatingLocation` method to start the determination of the user's current location. When new location data is available, the location manager notifies its assigned delegate object. If a location update has already been delivered, you can also get the most recent location data directly from the `CLLocationManager` object without waiting for a new event to be delivered.

Listing 10-5 shows implementations of a custom `startUpdates` method and the `locationManager:didUpdateToLocation:fromLocation:` delegate method. The `startUpdates` method creates a new location manager object (if one does not already exist) and uses it to start generating location updates. (In this case, the `locationManager` variable is a member variable declared by the `MyLocationGetter` class, which also conforms to the `CLLocationManagerDelegate` protocol.) The handler method uses the timestamp of the event to determine how recent it is. If it is an old event, the handler ignores it and waits for a more recent one, at which point it disables the location service.

Listing 10-5 Initiating and processing location updates

```
@implementation MyLocationGetter
- (void)startUpdates
{
    // Create the location manager if this object does not
    // already have one.
    if (nil == locationManager)
        locationManager = [[CLLocationManager alloc] init];

    locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;

    // Set a movement threshold for new events
    locationManager.distanceFilter = 500;

    [locationManager startUpdatingLocation];
}


// Delegate method from the CLLocationManagerDelegate protocol.
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    // If it's a relatively recent event, turn off updates to save power
    NSDate* eventDate = newLocation.timestamp;
    NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];
    if (abs(howRecent) < 5.0)
    {
        [manager stopUpdatingLocation];

        printf("latitude %+.6f, longitude %+.6f\n",
                newLocation.coordinate.latitude,
                newLocation.coordinate.longitude);
    }
    // else skip the event and process the next one.
}
@end
```

Checking the timestamp of an event is recommended because the location service often returns the last cached location event immediately. It can take several seconds to obtain a rough location fix so the old data simply serves as a way to reflect the last known location. You can also use the accuracy as a means of determining whether you want to accept an event. As it receives more accurate data, the location service may return additional events, with the accuracy values reflecting the improvements accordingly.

Note: The Core Location framework records timestamp values at the beginning of each location query, not when that query returns. Because Core Location uses several different techniques to get a location fix, queries can sometimes come back in a different order than their timestamps might otherwise indicate. As a result, it is normal for new events to sometimes have timestamps that are slightly older than those from previous events. The framework concentrates on improving the accuracy of the location data with each new event it delivers, regardless of the timestamp values.

For more information about the objects and methods of the Core Location framework, see *Core Location Framework Reference*.

Taking Pictures with the Camera

UIKit provides access to a device's camera through the `UIImagePickerController` class. This class displays the standard system interface for taking pictures using the available camera. It also supports optional controls for resizing and cropping the image after the user takes it. This class can also be used to select photos from the user's photo library.

The view representing the camera interface is a modal view that is managed by the `UIImagePickerController` class. You should never access this view directly from your code. To display it, you must call the `presentModalViewController:animated:` method of the currently active view controller, passing a `UIImagePickerController` object as the new view controller. Upon being installed, the picker controller automatically slides the camera interface into position, where it remains active until the user approves the picture or cancels the operation. At that time, the picker controller notifies its delegate of the user's choice.

Interfaces managed by the `UIImagePickerController` class may not be available on all devices. Before displaying the camera interface, you should always make sure that the interface is available by calling the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class. You should always respect the return value of this method. If this method returns `NO`, it means that the current device does not have a camera or that the camera is currently unavailable for some reason. If the method returns `YES`, you display the camera interface by doing the following:

1. Create a new `UIImagePickerController` object.
2. Assign a delegate object to the picker controller.

   In most cases, the current view controller acts as the delegate for the picker, but you can use an entirely different object if you prefer. The delegate object must conform to the `UIImagePickerControllerDelegate` protocol.
3. Set the picker type to `UIImagePickerControllerSourceTypeCamera`.
4. Optionally, enable or disable the picture editing controls by assigning an appropriate value to the `allowsImageEditing` property.

5. Call the `presentModalViewController:animated:` method of the current view controller to display the picker.

Listing 10-6 shows the code representing the preceding set of steps. As soon as you call the `presentModalViewController:animated` method, the picker controller takes over, displaying the camera interface and responding to all user interactions until the interface is dismissed. To choose an existing photo from the user's photo library, all you have to do is change the value in the `sourceType` property of the picker to `UIImagePickerControllerSourceTypePhotoLibrary`.

Listing 10-6 Displaying the interface for taking pictures

```
-(BOOL)startCameraPickerFromViewController:(UIViewController*)controller
usingDelegate:(id<UIImagePickerControllerDelegate>)delegateObject
{
    if ( (![UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
            || (delegateObject == nil) || (controller == nil))
        return NO;

    UIImagePickerController* picker = [[UIImagePickerController alloc] init];
    picker.sourceType = UIImagePickerControllerSourceTypeCamera;
    picker.delegate = delegateObject;
    picker.allowsImageEditing = YES;

    // Picker is displayed asynchronously.
    [controller presentModalViewController:picker animated:YES];
    return YES;
}
```

When the user taps the appropriate button to dismiss the camera interface, the `UIImagePickerController` notifies the delegate of the user's choice but does not dismiss the interface. The delegate is responsible for dismissing the picker interface. (Your application is also responsible for releasing the picker when done with it, which you can do in the delegate methods.) It is for this reason that the delegate is actually the view controller object that presented the picker in the first place. Upon receiving the delegate message, the view controller would call its `dismissModalViewControllerAnimated:` method to dismiss the camera interface.

Listing 10-7 shows the delegate methods for dismissing the camera interface displayed in Listing 10-6 (page 191). These methods are implemented by a custom `MyViewController` class, which is a subclass of `UIViewController` and, for this example, is considered to be the same object that displayed the picker in the first place. The `useImage:` method is an empty placeholder for the work you would do in your own version of this class and should be replaced by your own custom code.

Listing 10-7 Delegate methods for the image picker

```
@implementation MyViewController (ImagePickerDelegateMethods)

- (void)imagePickerController:(UIImagePickerController *)picker
                    didFinishPickingImage:(UIImage *)image
                    editingInfo:(NSDictionary *)editingInfo
{
    [self useImage:image];

    // Remove the picker interface and release the picker object.
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
```

```
    [picker release];
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
    [picker release];
}

// Implement this method in your code to do something with the image.
- (void)useImage:(UIImage*)theImage
{
}
@end
```

If image editing is enabled and the user successfully picks an image, the `image` parameter of the `imagePickerController:didFinishPickingImage:editingInfo:` method contains the edited image. You should treat this image as the selected image, but if you want to store the original image, you can get it (along with the crop rectangle) from the dictionary in the `editingInfo` parameter.

Picking a Photo from the Photo Library

UIKit provides access to the user's photo library through the `UIImagePickerController` class. This controller displays a photo picker interface, which provides controls for navigating the user's photo library and selecting an image to return to your application. You also have the option of enabling user editing controls, which let the user the pan and crop the returned image. This class can also be used to present a camera interface.

Because the `UIImagePickerController` class is used to display the interface for both the camera and the user's photo library, the steps for using the class are almost identical for both. The only difference is that you assign the `UIImagePickerControllerSourceTypePhotoLibrary` value to the `sourceType` property of the picker object. The steps for displaying the camera picker are discussed in "Taking Pictures with the Camera" (page 190).

Note: As you do for the camera picker, you should always call the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class and respect the return value of the method. You should never assume that a given device has a photo library. Even if the device has a library, this method could still return `NO` if the library is currently unavailable.

Application Preferences

In traditional desktop applications, preferences are application-specific settings used to configure the behavior or appearance of an application. iPhone OS also supports application preferences, although not as an integral part of your application. Instead of each application displaying a custom user interface for its preferences, all application-level preferences are displayed using the system-supplied Settings application.

In order to integrate your custom application preferences into the Settings application, you must include a specially formatted settings bundle in the top-level directory of your application bundle. This settings bundle provides information about your application preferences to the Settings application, which is then responsible for displaying those preferences and updating the preferences database with any user-supplied values. At runtime, your application retrieves these preferences using the standard retrieval APIs. The sections that follow describe both the format of the settings bundle and the APIs you use to retrieve your preferences values.

Guidelines for Preferences

Adding your application preferences to the Settings application is most appropriate for productivity-style applications and in situations where you have preference values that are typically configured once and then rarely changed. For example, the Mail application uses these preferences to store the user's account information and message-checking settings. Because the Settings application has support for displaying preferences hierarchically, manipulating your preferences from the Settings application is also more appropriate when you have a large number of preferences. Providing the same set of preferences in your application might require too many screens and might cause confusion for the user.

When your application has only a few options or has options that the user might want to change regularly, you should think carefully about whether the Settings application is the right place for them. For instance, utility applications provide custom configuration options on the back of their main view. A special control on the view flips it over to display the options and another control flips the view back. For simple applications, this type of behavior provides immediate access to the application's options and is much more convenient for the user than going to Settings.

For games and other full-screen applications, you can use the Settings application or implement your own custom screens for preferences. Custom screens are often appropriate in games because those screens are treated as part of the game's setup. You can also use the Settings application for your preferences if you think it is more appropriate for your game flow.

Note: You should never spread your preferences across the Settings application and custom application screens. For example, a utility application with preferences on the back side of its main view should not also have configurable preferences in the Settings application. If you have preferences, pick one solution and use it exclusively.

The Preferences Interface

The Settings application implements a hierarchical set of pages for navigating application preferences. The main page of the Settings application displays the system and third-party applications whose preferences can be customized. Selecting a third-party application takes the user to the preferences for that application.

Each application has at least one page of preferences, referred to as the main page. If your application has only a few preferences, the main page may be the only one you need. If the number of preferences gets too large to fit on the main page, however, you can add more pages. These additional pages become child pages of the main page. The user accesses them by tapping on a special type of preference, which links to the new page.

Each preference you display must be of a specific type. The type of the preference defines how the Settings application displays that preference. Most preference types identify a particular type of control that is used to set the preference value. Some types provide a way to organize preferences, however. Table 11-1 lists the different element types supported by the Settings application and how you might use each type to implement your own preference pages.

Table 11-1 Preference element types

Element Type	Description
Text Field	The text field type displays an optional title and an editable text field. You can use this type for preferences that require the user to specify a custom string value. The key for this type is `PSTextFieldSpecifier`.
Title	The title type displays a read-only string value. You can use this type to display read-only preference values. (If the preference contains cryptic or nonintuitive values, this type lets you map the possible values to custom strings.) The key for this type is `PSTitleValueSpecifier`.
Toggle Switch	The toggle switch type displays an ON/OFF toggle button. You can use this type to configure a preference that can have only one of two values. Although you typically use this type to represent preferences containing Boolean values, you can also use it with preferences containing non-Boolean values. The key for this type is `PSToggleSwitchSpecifier`.
Slider	The slider type displays a slider control. You can use this type for a preference that represents a range of values. The value for this type is a real number whose minimum and maximum you specify. The key for this type is `PSSliderSpecifier`.

Element Type	Description
Multi value	The multi value type lets the user select one value from a list of values. You can use this type for a preference that supports a set of mutually exclusive values. The values can be of any type. The key for this type is `PSMultiValueSpecifier`.
Group	The group type is a way for you to organize groups of preferences on a single page. The group type does not represent a configurable preference. It simply contains a title string that is displayed immediately before one or more configurable preferences. The key for this type is `PSGroupSpecifier`.
Child Pane	The child pane type lets the user navigate to a new page of preferences. You use this type to implement hierarchical preferences. For more information on how you configure and use this preference type, see "Hierarchical Preferences" (page 197). The key for this type is `PSChildPaneSpecifier`.

For detailed information about the format of each preference type, see *Settings Application Schema Reference*. To learn now to create and edit Setting page files, see "Adding the Settings Bundle to Your Application" (page 198).

The Settings Bundle

In iPhone OS, you specify your application's preferences through a special settings bundle. This bundle has the name `Settings.bundle` and resides in the top-level directory of your application's bundle. This bundle contains one or more Settings Page files that provide detailed information about your application's preferences. It may also include other support files needed to display your preferences, such as images or localized strings. Table 11-2 lists the contents of a typical settings bundle.

Table 11-2 Contents of the `Settings.bundle` directory

Item name	Description
`Root.plist`	The Settings Page file containing the preferences for the root page. The contents of this file are described in more detail in "The Settings Page File Format" (page 196).
Additional `.plist` files.	If you build a set of hierarchical preferences using child panes, the contents for each child pane are stored in a separate Settings Page file. You are responsible for naming these files and associating them with the correct child pane.
One or more `.lproj` directories	These directories store localized string resources for your Settings Page files. Each directory contains a single strings file, whose title is specified in your Settings Page. The strings files provide the localized content to display to the user for each of your preferences.

Item name	Description
Additional images	If you use the slider control, you can store the images for your slider in the top-level directory of the bundle.

In addition to the settings bundle, your application bundle can contain a custom icon for your application settings. If a file with the name `Icon-Settings.png` is located in the top of your application's bundle directory, that icon is used to identify your application preferences in the Settings application. If no such image file is present, the Settings application uses your application's icon file (`Icon.png` by default) instead, scaling it as necessary. Your `Icon-Settings.png` file should be a 29 x 29 pixel image.

When the Settings application launches, it checks each custom application for the presence of a settings bundle. For each custom bundle it finds, it loads that bundle and displays the corresponding application's name and icon in the Settings main page. When the user taps the row belonging to your application, Settings loads the `Root.plist` Settings Page file for your settings bundle and uses that file to display your application's main page of preferences.

In addition to loading your bundle's `Root.plist` Settings Page file, the Settings application also loads any language-specific resources for that file, as needed. Each Settings Page file can have an associated `.strings` file containing localized values for any user-visible strings. As it prepares your preferences for display, the Settings application looks for string resources in the user's preferred language and substitutes them into your preferences page prior to display.

The Settings Page File Format

Each Settings Page file in your settings bundle is stored in the iPhone Settings property-list file format, which is a structured file format. The simplest way to edit Settings Page files is using Xcode's built in editor facilities; see "Editing Settings Pages" (page 199). You can also edit property-list files using the Property List Editor application that comes with the Xcode tools.

Note: Xcode automatically converts any XML-based property files in your project to binary format when building your application. This conversion saves space and is done for you automatically at build time.

The root element of each Settings Page file contains the keys listed in Table 11-3. Only one key is actually required, but it is recommended that you include both of them.

Table 11-3 Root-level keys of a preferences Settings Page file

Key	Type	Value
`PreferenceSpecifiers` (required)	Array	The value for this key is an array of dictionaries, with each dictionary containing the information for a single preference element. For a list of element types, see Table 11-1 (page 194). For a description of the keys associated with each element type, see *Settings Application Schema Reference*.

Key	Type	Value
StringsTable	String	The name of the strings file associated with this file. A copy of this file (with appropriate localized strings) should be located in each of your bundle's language-specific project directories. If you do not include this key, the strings in this file are not localized. For information on how these strings are used, see "Localized Resources" (page 198).

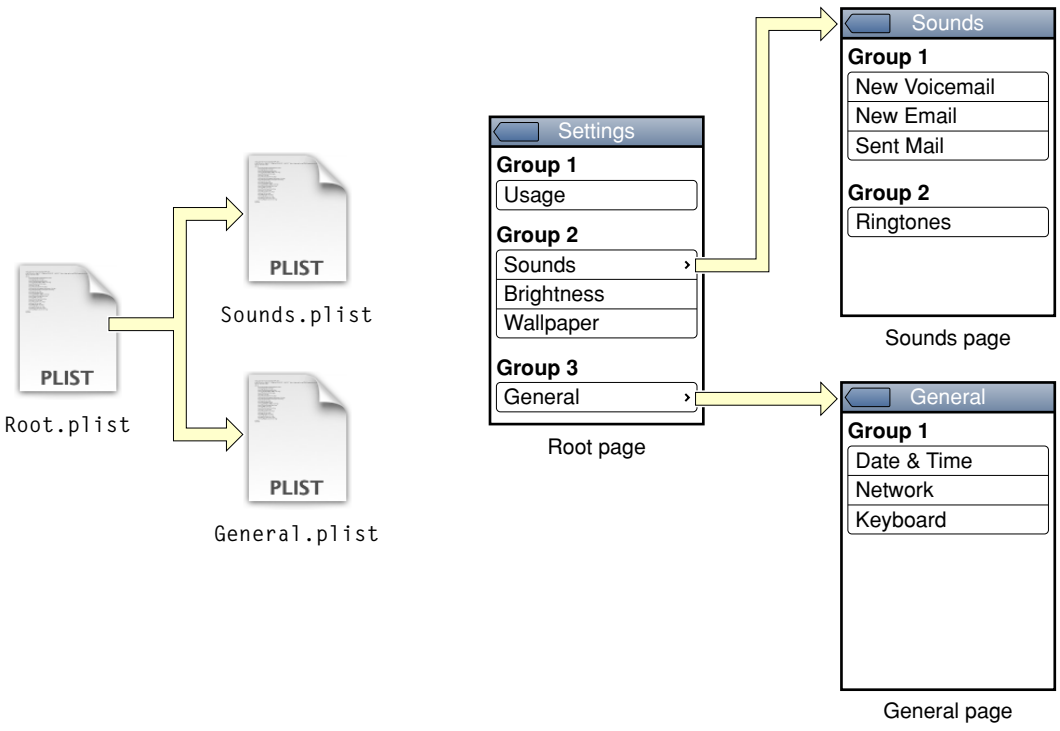
Hierarchical Preferences

If you plan to organize your preferences hierarchically, each page you define must have its own separate `.plist` file. Each `.plist` file contains the set of preferences displayed only on that page. Your application's main preferences page is always stored in the `Root.plist` file. Additional pages can be given any name you like.

To specify a link between a parent page and a child page, you include a child pane element in the parent page. The child pane element contains a property whose value is the name of the `.plist` file for the child page. When the user taps the row represented by the child pane element, the Settings application loads the associated `.plist` file and displays the new page. It also provides navigation controls on the child page that allow the user to navigate back to the parent page.

Figure 11-1 shows how this hierarchical set of pages works. The left side of the figure shows the `.plist` files, and the right side shows the relationships between the corresponding pages.

Figure 11-1 Organizing preferences using child panes



Localized Resources

Because preferences contain user-visible strings, you should provide localized versions of those strings with your settings bundle. Each page of preferences can have an associated `.strings` file for each localization supported by your bundle. When the Settings application encounters a key that supports localization, it checks the appropriately localized `.strings` file for a matching key. If it finds one, it displays the value associated with that key.

When looking for localized resources such as `.strings` files, the Settings application follows the the same rules that Mac OS X applications do. It first tries to find a localized version of the resource that matches the user's preferred language setting. If a resource does not exist for the user's preferred language, an appropriate fallback language is selected.

For information about the format of strings files, language-specific project directories, and how language-specific resources are retrieved from bundles, see *Internationalization Programming Topics*.

Adding the Settings Bundle to Your Application

Xcode provides a template for adding a Settings bundle to your current project. The default settings bundle contains a `Root.plist` file and a default language directory for storing any localized resources. You can then expand this bundle to include additional property list files and resources needed by your Settings bundle.

To add a Settings bundle to your Xcode project, do the following:

1. Choose File > New File.
2. Choose the iPhone OS > Settings > Settings Bundle template.
3. Name the file `Settings.bundle`.

In addition to adding a new Settings bundle to your project, Xcode automatically adds that bundle to the Copy Bundle Resources build phase of your application target. Thus, all you have to do is modify the property list files of your Settings bundle and add any needed resources.

The newly added `Settings.bundle` bundle has the following structure:

```
Settings.bundle/
    Root.plist
    en.lproj/
        Root.strings
```

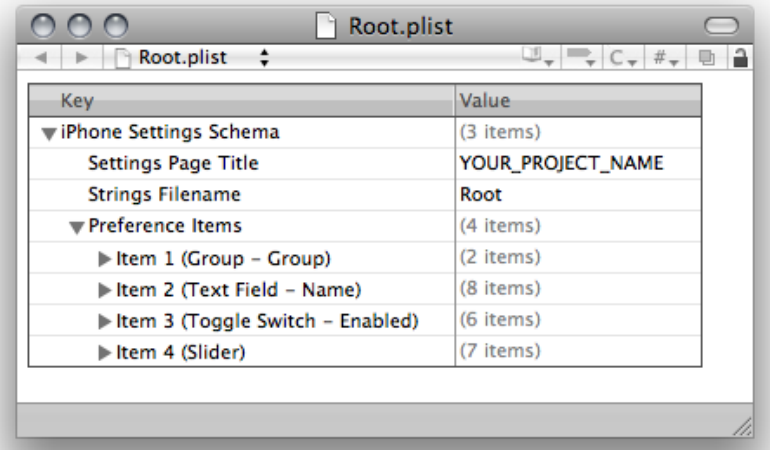
To add additional Settings Page files or other resources to your Settings bundle, add them directly to the the Settings bundle in the Finder or in a shell editor.

Note: You cannot modify the structure of your Settings bundle—by adding or removing files or directories—in the Groups & Files list.

Editing Settings Pages

After creating your Settings bundle using the Settings Bundle template, open the root Settings page file:

1. In the Groups & Files list, select `Settings.bundle`.
2. In the Detail View, double-click `Root.plist`.
3. Choose View > Property List Type > iPhone Settings plist.



Now you can design your own Settings page. The remainder of this section shows how to edit `Root.plist` to create a Settings page like the one in Figure 11-2.

Figure 11-2 A root Settings page

1. You should change the title of the root Settings page to the name of your application. Just double-click `YOUR_PROJECT_NAME` and enter the new title.
2. The first group of settings has a pair of sound-related settings. Therefore, this group should be titled "Sound."
    a. Disclose item 1 of `Preference Items`.
    b. Change the value of the `Title` key from `Group` to `Sound`.

    The Settings application groups the elements that follow this item (except for another group) under a group titled "Sound."
3. The first sound-related setting specifies whether the application should play sounds. Its title is "Play Sounds."
    a. Move item 3 of `Preference Items` to item 2:
        a. Select item 3 of `Preference Items`.
        b. Choose Edit > Cut.
        c. Select item 1.
        d. Choose Edit > Paste.

        You can also use drag-and-drop to move items.
    b. Disclose item 2.

c. Change the value of the `Title` key to `Play Sounds`.

d. Change the value of the `Identifier` key to `play_sounds_preference`.

Item 2 (Toggle Switch - Play Sounds)	(6 items)
Type	Toggle Switch
Title	Play Sounds
Identifier	play_sounds_preference
Default Value	☑
Value for ON	YES
Value for OFF	NO

4. The second sound-related setting is 3D Sound.

   a. Select item 2.

   b. Choose Edit > Copy.

   c. Choose Edit > Paste.

   d. Disclose item 3.

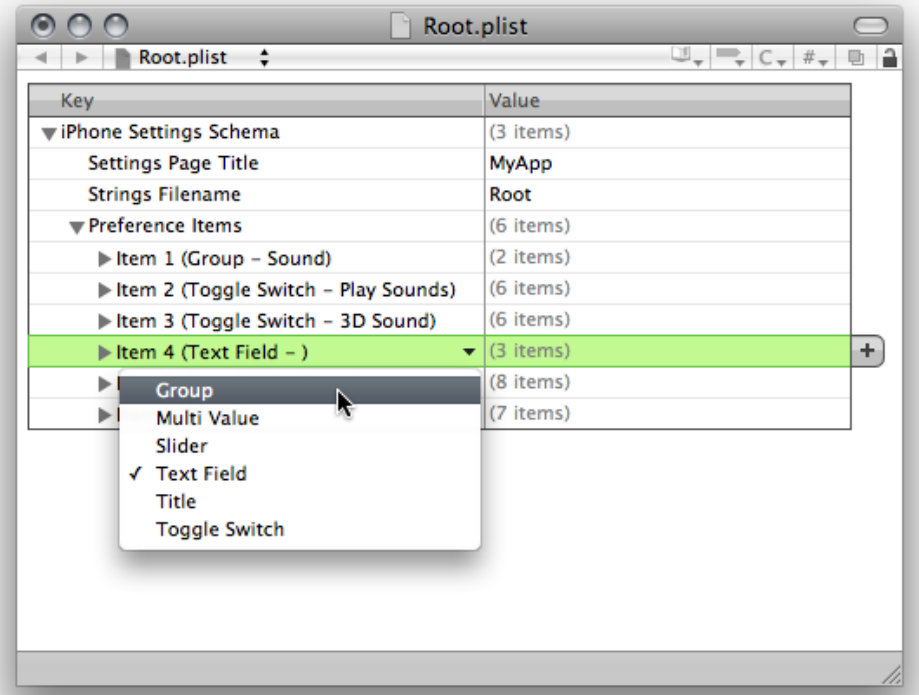
   e. Change the value of the `Title` key to `3D Sound`.

   f. Change the value of the `Identifier` key to `3d_sound_preference`.

5. The second settings group contains user information.

   a. Select item 3 and press Return.

b. Choose Group from the dictionary type menu of item 4.



c. Disclose item 4.

d. Set the value of the `Title` key to `User Info`.

6. The template configuration of item 5 is appropriate for this page.

7. The second user-related setting is Experience Level.

   a. Select item 5 and press Return to create a multi value element.

   b. Set the title to “Experience Level”.

   c. Set the identifier to “experience_preference”.

   d. Set the default value to zero.

   e. Click the plus (+) button (or press Return) to add a `Titles` item.

   f. With the `Titles` item selected, press Option-Return.

   g. Enter `Beginner`, `Expert`, and `Master` as the items of the `Titles` array.

   h. Close the `Titles` item.

   i. Select the `Titles` item and press Return to add the `Values` item.

j. Add `0`, `1`, and `2` as the items of the `Values` array.

8. Select item 6 and press Return to add a `Group` item and set its title to “Gravity”.

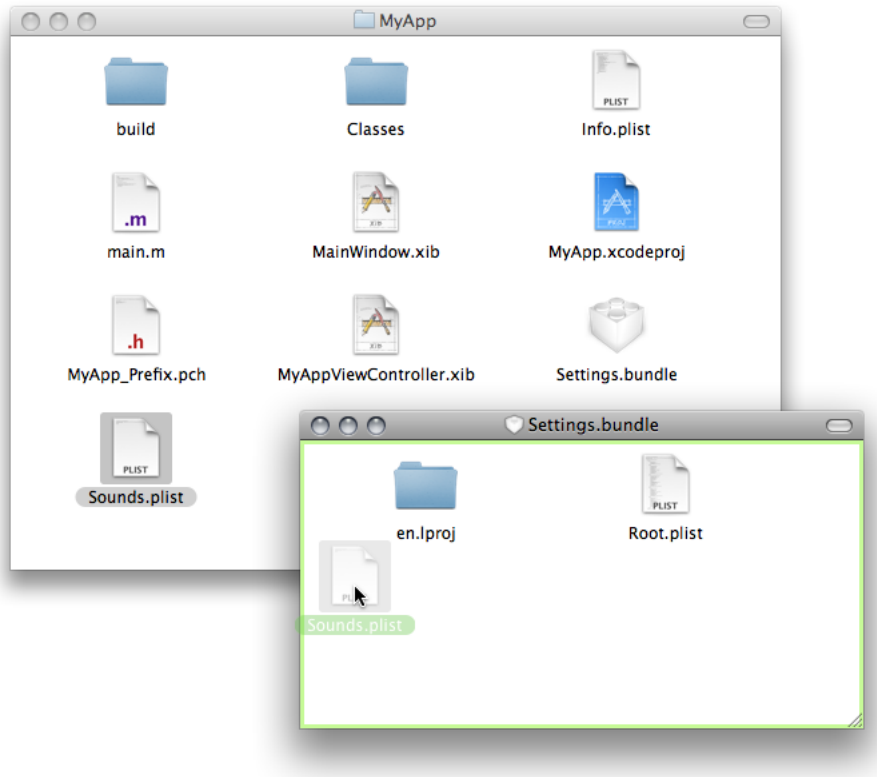
Creating Settings Page Files

The Settings Bundle template includes the `Root.plist` file, which defines your application’s top Settings page. If you need to define additional Settings pages, you need to add additional property list files to your Settings bundle.

To add a property list file to your Settings bundle:

1. Launch Property List Editor.
2. Choose File > New.
3. Save the File to your project directory.
4. Move the new plist file into the `Settings.bundle` directory:
    a. In the Finder, navigate to your project directory.
    b. Select the `Settings.bundle` file.
    c. Choose Show Package Contents from the shortcut menu.

5. Drag the new plist file to the `Settings.bundle` directory.



Accessing Your Preferences

iPhone applications get and set preferences values using either the Foundation and Core Foundation frameworks. In the Foundation framework, you use the `NSUserDefaults` class to get and set preference values. In the Core Foundation framework, you use several preferences-related functions to get and set values.

Listing 11-1 shows a simple example of how to read a preference value from your application. This example uses the `NSUserDefaults` class to read a Boolean value from preferences and assign it to an application-specific instance variable.

Listing 11-1 Accessing preference values in an application

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [self setMyAppBoolProperty:[defaults boolForKey:MY_BOOL_PREF_KEY]];

    // Finish app initialization...
}
```

For information about the `NSUserDefaults` methods used to read and write preferences, see *NSUserDefaults Class Reference*. For information about the Core Foundation functions used to read and write preferences, see *Preferences Utilities Reference*.

Debugging Preferences for Simulated Applications

When running your application, the iPhone Simulator stores any preferences values for your application in `~/Library/Application Support/iPhone Simulator/User/Applications/`<*APP_ID*>`/Library/Preferences`, where <*APP_ID*> is a programmatically generated directory name that iPhone OS uses to identify your application.

Each time you reinstall your application, iPhone OS performs a clean install, which deletes any previous preferences. In other words, building or running your application from Xcode always installs a new version, replacing any old contents. To test preference changes between successive executions, you must run your application directly from the simulator interface and not from Xcode.

Apple Applications URL Schemes

Both Safari on iPhone and native applications can integrate seamlessly with other iPhone applications to provide a rich user experience using URL schemes. You access other applications by adding phone, mail, map, iTunes, and YouTube links to your webpages or by launching the corresponding applications from a native application.

Although most of the examples in this appendix are HTML, iPhone applications can also launch other applications using these same URL schemes by sending the `openURL:` message to their shared `UIApplication` object.

This appendix describes the different URL schemes for Apple applications only.

Mail Links

When the user taps a mail link in Safari on iPhone, an email compose sheet opens in Mail with the address filled in. Simply use the standard `mailto` URL as follows in HTML:

```
<a href="mailto:frank@wwdcdemo.example.com">John Frank</a>
```

If you want to launch Mail from a native application, just pass the URL—for example, `mailto:frank@wwdcdemo.example.com`—to the `openURL:` method.

You can also include multiple recipients, a subject field, a from field, and a message in the `mailto` URL as follows:

```
mailto:foo@example.com?cc=bar@example.com&subject=
Greetings%20from%Cupertino!&body=Wish%20you%20were%20here!
```

Clicking a mail link in Safari on iPhone displays an unsupported dialog if the Mail application is not installed on the device.

For more information on the format of the `mailto` scheme, see RFC 2368.

Phone Links

Safari on iPhone automatically converts any number on your webpage that takes the form of a phone number to a phone link. When the user taps a phone link, a dialog appears asking whether the user wants to dial that phone number. If the user taps the Call button in the dialog, the Phone application launches and dials the phone number. You can also add your own phone links to your webpage or disable telephone number detection.

The syntax of a telephone link in HTML is:

```
<a href="tel:1-408-555-5555">1-408-555-5555</a>
```

Telephone number detection is on by default in Safari on iPhone. Therefore, if your webpage contains numbers that can be interpreted as phone numbers, but are not phone numbers, you can turn off telephone number detection. You might also turn off telephone number detection to prevent the DOM document from being modified when parsed by the browser.

To turn off telephone number detection in Safari on iPhone, use the `format-detection` meta tag as follows:

```
<meta name = "format-detection" content = "telephone=no">
```

For example, in Listing A-1, automatic telephone number detection is off. Therefore, the `408-555-5555` telephone number does not appear as a link when rendered by Safari on iPhone. However, the `1-408-555-5555` number does appear as a link because it is in a phone link.

Listing A-1 Turning telephone number detection off

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" >
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Telephone Number Detection</title>
    <meta name = "viewport" content = "width=device-width">
     <!-- Turn off telephone number detection. -->
    <meta name = "format-detection" content = "telephone=no">
</head>
<body>
     <!-- Then use phone links to explicitly create a link. -->
     <p>A phone number: <a href="tel:1-408-555-5555">1-408-555-5555</a></p>
     <!-- Otherwise, numbers that look like phone numbers are not links. -->
     <p>Not a phone number: 1-408-555-5511</p>
</body>
</html>
```

Clicking a phone link displays an unsupported dialog in Safari on iPhone if the Phone application is not installed on the device.

If you are launching the Phone application from a native application, make sure that the phone number is in the correct format by using the `stringByAddingPercentEscapesUsingEncoding:` method in `NSString`.

Map Links

When the user taps a map link in Safari on iPhone, a Google map opens containing the destination—for example, a map showing Cupertino, California is displayed when the user taps this link:

```
<a href="http://maps.google.com/maps?q=cupertino">Cupertino</a>
```

The following URL provides directions between two points, San Francisco and Cupertino:

```
http://maps.google.com/maps?daddr=San+Francisco,+CA&saddr=cupertino">Directions
```

Clicking a map link redirects to the Google Maps website if the Maps application is not installed on the device.

However, not all Google Map parameters and queries are supported in iPhone OS. The rules for creating a valid map link are:

- The domain must be `google` and the subdomain must be `maps` or `ditu`.
- The path must be `/`, `/maps`, `/local`, or `/m` if the query contains `site` as the key and `local` as the value.
- The path cannot be `/maps/*`.
- All parameters must be supported. See "`Supported Google Map parameters`" for list of supported parameters.
- A parameter cannot be `q=*` if the value is a URL (so KML is not picked up).
- The parameters cannot include `view=text` or `dirflg=r`.

Table A-1 lists the parameters supported by iPhone OS along with a brief description of each. For a complete description of these parameters, see Google Map Parameters.

Table A-1 Supported Google Map parameters

Parameter	Notes
q=	The query parameter. This parameter is treated as if it had been typed into the query box by the user on the maps.google.com page. `q=*` is not supported
near=	The location part of the query.
ll=	The latitude and longitude points (in decimal format, comma separated, and in that order) for the map center point.
sll=	The latitude and longitude points from which the business search should be performed.
spn=	The approximate latitude and longitude span.
sspn=	A custom latitude and longitude span format used by Google.
t=	The type of map to display.
z=	The zoom level.

Parameter	Notes
saddr=	The source address, which is used when generating driving directions
daddr=	The destination address, which is used when generating driving directions.
latlng=	A custom ID format that Google uses for identifying businesses.
cid=	A custom ID format that Google uses for identifying businesses.

YouTube Links

When the user taps a YouTube URL in Safari on iPhone, the YouTube application launches and plays the movie specified in the URL.

The supported YouTube URL formats are as follows, where you replace `<video identifier>` with the YouTube video identifier and the preceding `www.` is optional:

```
http://www.youtube.com/watch?v=<video identifier>
http://www.youtube.com/v/<video identifier>
```

A warning message appears if the YouTube video cannot be viewed on the device.

iTunes Links

You can link to the iTunes music store from your web content or launch iTunes from a native application using a URL. The iTunes URL is complicated to construct, so you create it using an online tool called iTunes Link Maker. The tool allows you to select a country destination and media type, and then search by song, album, or artist. After you select the item you want to link to, it generates the corresponding URL. For example, this HTML fragment links to a song:

```
<a
href="http://phobos.apple.com/WebObjects/MZStore.woa/wa/viewAlbum?i=156093464&id=156093462&s=143441">
<img height="15" width="61" alt="Randy Newman - Toy Story
- You&#39;ve Got a Friend In Me"
src="http://ax.phobos.apple.com.edgesuite.net/images/
badgeitunes61x15dark.gif"></img>
</a>
```

Go to iTunes Link Maker FAQ for more information on creating iTunes links. This webpage contains a link to the iTunes Link Maker tool.

Document Revision History

This table describes the changes to *iPhone OS Programming Guide*.

Date	Notes
2008-07-08	New document that describes iPhone OS and the development process for iPhone applications.

Document Revision History