Event Handling Guide for iOS

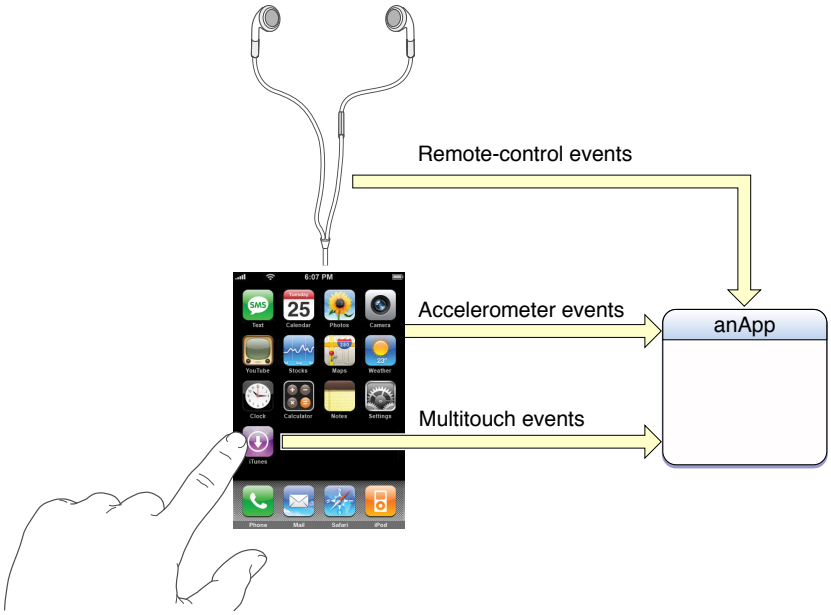
Developer

Contents

Figures, Tables, and Listings

About Events in iOS

Events are objects sent to an application to inform it of user actions. In iOS, events can take many forms: multitouch events, motion events—for example, from device accelerometers—and events for controlling multimedia. (This last type of event is known as a remote-control event because it originates from a headset or other external accessory.)



The UIKit and Core Motion frameworks are responsible for event propagation and delivery in iOS.

At a Glance

An Application Receives Multitouch Events When Users Touch its Views

The Multi-Touch interface of iPhones, iPads, and iPod touches generates low-level events when users touch views of an application. The application sends these events (as `UIEvent` objects) to the view on which the touches occurred. That view typically analyzes the touches represented by each event object and responds in an appropriate manner.

Applications are frequently interested in interpreting the touches a user makes as a common gesture, such as a tap or swipe gesture. These applications can make use of UIKit classes called gesture recognizers, each of which is designed to recognize a specific gesture.

Relevant Chapters: "Event Types and Delivery" (page 9), "Multitouch Events" (page 16), "Gesture Recognizers" (page 39)

An Application Receives Motion Events When Users Move the Device

Motion events come in different forms, and you can handle them using different frameworks. When users shake the device, the UIKit delivers a `UIEvent` object to an application; these shaking-motion events are gestures often used to trigger undo and redo actions. If you want your application to receive high-rate, continuous accelerometer and gyroscope data, use the Core Motion framework. (Only certain devices have a gyroscope.) You may also use the `UIAccelerometer` class to receive and handle accelerometer data.

Relevant Chapters: "Event Types and Delivery" (page 9), "Motion Events" (page 56)

Remote-Control Events Are Sent When Users Manipulate Multimedia Controls

By conforming to an Apple-provide specification, headsets and other external accessories can send (via the UIKit framework) remote-control events to an application capable of playing audio or video. The view hosting the multimedia can receive the events and thereby control the audio video according to the user's command (for example, pausing or fast-forwarding).

Relevant Chapters: "Event Types and Delivery" (page 9), "Remote Control of Multimedia" (page 76)

How to Use this Book

Regardless of the type of event you're interested in, you should first read "Event Types and Delivery" (page 9). This chapter provides essential background information.

See Also

Some iPhones and other devices have GPS and compass hardware that also generate low-level data delivered to an application for processing. *Location Awareness Programming Guide* discusses how to receive and handle this data.

Many sample code projects in the iOS Reference Library have code that illustrates the handling of multitouch events and the use of gesture recognizers. Among these are the following projects: *Touches*, *CopyPasteTile*, and *SimpleGestureRecognizers*.

Event Types and Delivery

An iPhone, iPad, or iPod touch device has multiple items of hardware that generate streams of input data an application can access. The Multi-Touch technology enables the direct manipulation of views, including the virtual keyboard. Three accelerometers measure acceleration along the three spatial axes. A gyroscope (only on some device models) measures the rate of rotation around the three axes. The Global Positioning System (GPS) and compass provide measurements of location and orientation. Each of these hardware systems, as they detect touches, device movements, and location changes, produce raw data that is passed to system frameworks. The frameworks package the data and deliver them as events to an application for processing.

The following sections identify these frameworks and describe how events are packaged and delivered to applications for handling.

> **Note** This document describes touch events, motion events, and remote control events only. For information on handling GPS and magnetometer (compass) data, see *Location Awareness Programming Guide*.

UIKit Event Objects and Types

An *event* is an object that represents a user action detected by hardware on the device and conveyed to iOS—for example, a finger touching the screen or hand shaking the device. Many events are instances of the `UIEvent` class of the UIKit framework. A `UIEvent` object may encapsulate state related to the user event, such as the associated touches. It also records the moment the event was generated. As a user action takes place—for example, as fingers touch the screen and move across its surface—the operating system continually sends event objects to an application for handling.

UIKit currently recognizes three types of events: touch events, "shaking" motion events, and remote-control events. The `UIEvent` class declares the `enum` constants shown in Listing 1-1.

Listing 1-1 Event-type and event-subtype constants

```
typedef enum {
    UIEventTypeTouches,
    UIEventTypeMotion,
```

```
    UIEventTypeRemoteControl,
} UIEventType;

typedef enum {
    UIEventSubtypeNone                              = 0,

    UIEventSubtypeMotionShake                       = 1,

    UIEventSubtypeRemoteControlPlay                 = 100,
    UIEventSubtypeRemoteControlPause                = 101,
    UIEventSubtypeRemoteControlStop                 = 102,
    UIEventSubtypeRemoteControlTogglePlayPause      = 103,
    UIEventSubtypeRemoteControlNextTrack            = 104,
    UIEventSubtypeRemoteControlPreviousTrack        = 105,
    UIEventSubtypeRemoteControlBeginSeekingBackward = 106,
    UIEventSubtypeRemoteControlEndSeekingBackward   = 107,
    UIEventSubtypeRemoteControlBeginSeekingForward  = 108,
    UIEventSubtypeRemoteControlEndSeekingForward    = 109,
} UIEventSubtype;
```

Each event has one of these event type and subtype constants associated with it, which you can access through the `type` and `subtype` properties of `UIEvent`. The event type includes touch events, motion events, and remote control events. In iOS 3.0, there is a shake-motion subtype (`UIEventSubtypeMotionShake`) and many remote-control subtypes; touch events always have a subtype of `UIEventSubtypeNone`.

A remote-control event originates as commands from the system transport controls or an external accessory conforming to an Apple-provided specification, such as a headset. They are intended to allow users to control multimedia content using those controls and external accessories. Remote-control events are new with iOS 4.0 and are described in detail in "Remote Control of Multimedia" (page 76).

You should never retain a `UIEvent` object in your code. If you need to preserve the current state of an event object for later evaluation, you should copy and store those bits of state in an appropriate manner (using an instance variable or a dictionary object, for example).

A device running iOS can send other types of events, broadly considered, to an application for handling. These events are not `UIEvent` objects, but still encapsulate a measurement of some hardware-generated values. "Motion Event Types" (page 14) discusses these other events.

Event Delivery

The delivery of an event to an object for handling occurs along a specific path. As described in "Preparing Your Application for Remote-Control Events" (page 76), when users touch the screen of a device, iOS recognizes the set of touches and packages them in a `UIEvent` object that it places in the active application's event queue. If the system interprets the shaking of the device as a motion event, an event object representing that event is also placed in the application's event queue. The singleton `UIApplication` object 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 window object representing that window sends the event to an initial object for handling. That object is different for touch events and motion events.

- **Touch events**. The window object uses hit-testing and the responder chain to find the view to receive the touch event. In hit-testing, a window calls `hitTest:withEvent:` on the top-most view of the view hierarchy; this method proceeds by recursively calling `pointInside:withEvent:` on each view in the view hierarchy that returns `YES`, proceeding down the hierarchy until it finds the subview within whose bounds the touch took place. That view becomes the hit-test view.

  If the hit-test view cannot handle the event, the event travels up the responder chain as described in "Responder Objects and the Responder Chain" (page 11) until the system finds a view that can handle it. A touch object (described in "Events and Touches" (page 17)) is associated with its hit-test view for its lifetime, even if the touch represented by the object subsequently moves outside the view. "Hit-Testing" (page 33) discusses some of the programmatic implications of hit-testing.
- **Motion and remote-control events**. The window object sends each shaking-motion or remote-control event to the first responder for handling. (The first responder is described in "Responder Objects and the Responder Chain."

Although the hit-test view and the first responder are often the same view object, they do not have to be the same.

The `UIApplication` object and each `UIWindow` object dispatches events in the `sendEvent:` method. (These classes declare a method with the same signature). 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 (which is something few applications would need to do). If you override these methods, be sure to call the superclass implementation (that is, `[super sendEvent:theEvent]`); never tamper with the distribution of events.

Responder Objects and the Responder Chain

The preceding discussion mentions the concept of responders. What is a responder object and how does it fit into the architecture for event delivery?

A *responder object* is an object that can respond to events and handle them. `UIResponder` is the base class for all responder objects, also known as, simply, responders. 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`, and thus their instances are responder objects.

The *first responder* is the responder object in an application (usually a `UIView` object) that is designated to be the first recipient of events other than touch events. A `UIWindow` object sends the first responder these events in messages, giving it the first shot at handling them. To receive these messages, the responder object must implement `canBecomeFirstResponder` to return `YES`; it must also receive a `becomeFirstResponder` message (which it can invoke on itself). The first responder is the first view in a window to receive the following type of events and messages:

- Motion events—via calls to the `UIResponder` motion-handling methods described in "Shaking-Motion Events" (page 56)
- Remote-control events—via calls to the `UIResponder` method `remoteControlReceivedWithEvent:`
- Action messages—sent when the user manipulates a control (such as a button or slider) and no target is specified for the action message
- Editing-menu messages—sent when users tap the commands of the editing menu (described in "Displaying and Managing the Edit Menu")

The first responder also plays a role in text editing. A text view or text field that is the focus of editing is made the first responder, which causes the virtual keyboard to appear.

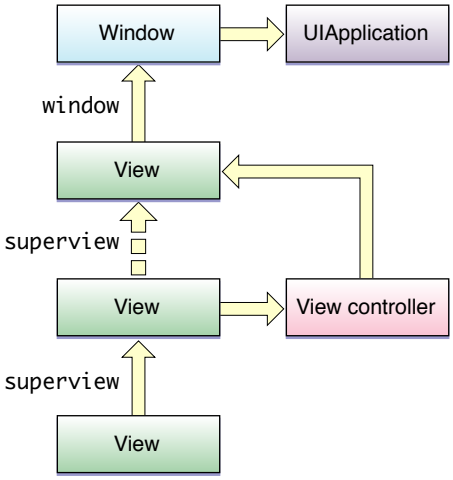
> **Note** Applications must explicitly set a first responder to handle motion events, action messages, and editing-menu messages; UIKit automatically sets the text field or text view a user taps to be the first responder.

If the first responder or the hit-test view doesn't handle an event, UIKit may pass 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 along which an event, action message, or editing-menu message is passed. It allows responder objects to transfer 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. Because the hit-test view is also a responder object, an application may also take advantage of the responder chain when handing touch events. The responder chain consists of a series of next responders (each returned by the `nextResponder` method) in the sequence depicted in Figure 1-1.

Figure 1-1 The responder chain in iOS



When the system delivers a touch event, it first sends it to a specific view. For touch events, that view is the one returned by `hitTest:withEvent:`; for "shaking"-motion events, remote-control events, action messages, and editing-menu messages, that view is the first responder. If the initial view doesn't handle the event, it travels up the responder chain along a particular path:

1. The hit-test view or first responder passes the event or message to its view controller if it has one; if the view doesn't have a view controller, it passes the event or message to its superview.
2. If a view or its view controller cannot handle the event or message, it passes it to the superview of the view.
3. Each subsequent superview in the hierarchy follows the pattern described in the first two steps if it cannot handle the event or message.
4. The topmost view in the view hierarchy, if it doesn't handle the event or message, passes it to the window object for handling.
5. The `UIWindow` object, if it doesn't handle the event or message, passes it to the singleton application object.

If the application object cannot handle the event or message, it discards it.

If you implement a custom view to handle "shaking"-motion events, remote-control events, action messages, or editing-menu messages, you should not forward the event or message to `nextResponder` directly to send it up the responder chain. Instead invoke the superclass implementation of the current event-handling method—let UIKit handle the traversal of the responder chain.

Motion Event Types

Motion events come from two hardware sources on a device: the three accelerometers and the gyroscope, which is available only some devices. An accelerometer measures changes in velocity over time along a given linear path. The 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. A gyroscope measures the rate of rotation around each of the three axes. (Although there are three accelerometers, one for each axis, the remainder of this document refers to them as a single entity.)

The Core Motion framework is primarily responsible for accessing raw accelerometer and gyroscope data and feeding that data to an application for handling. In addition, Core Motion processes combined accelerometer and gyroscope data using special algorithms and presents that refined motion data to applications. Motion events from Core Motion are represented by three data objects, each encapsulating one or more measurements:

- A `CMAccelerometerData` object encapsulates a structure that captures the acceleration along each of the spatial axes.
- A `CMGyroData` object encapsulates a structure that captures the rate of rotation around each of the three spatial axes.
- A `CMDeviceMotion` object encapsulates several different measurements, including attitude and more useful measurements of rotation rate and acceleration.

Core Motion is apart from UIKit architectures and conventions. There is no connection with the `UIEvent` model and there is no notion of first responder or responder chain. It delivers motion events directly to applications that request them.

The `CMMotionManager` class is the central access point for Core Motion. You create an instance of the class, specify an update interval (either explicitly or implicitly), request that updates start, and handle the motion events as they are delivered. "Core Motion" (page 63) describes this procedure in full detail.

An alternative to Core Motion, at least for accessing accelerometer data, is the `UIAccelerometer` class of the UIKit framework. When you use this class, accelerometer events are delivered as `UIAcceleration` objects. Although `UIAccelerometer` is part of UIKit, it is also separate from the `UIEvent` and responder-chain architectures. See "Accessing Accelerometer Events Using UIAccelerometer" (page 59) for information on using the UIKit facilities.

Notes The `UIAccelerometer` and `UIAcceleration` classes will be deprecated in a future release, so if your application handles accelerometer events, it should transition to the Core Motion API.

In iOS 3.0 and later, if you are trying to detect specific types of motion as gestures—specifically shaking motions—you should consider handling motion events (`UIEventTypeMotion`) instead of using the accelerometer interfaces. If you want to receive and handle high-rate, continuous motion data, you should instead use the Core Motion accelerometer API. Motion events are described in "Shaking-Motion Events" (page 56).

Multitouch Events

Note This chapter contains information that used to be in *iOS App Programming Guide*. The information in this chapter has not been updated specifically for iOS 4.0.

Touch events in iOS 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. iOS recognizes one or more fingers touching the screen as part of a *multitouch sequence*. This sequence begins when the first finger touches down on the screen and ends when the last finger is lifted from the screen. iOS tracks fingers touching the screen throughout a multitouch 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.

Notes 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.

iOS 4.0 still reports touches on iPhone 4 (and on future high-resolution devices) in a 320x480 coordinate space to maintain source compatibility, but the resolution is twice as high in each dimension for applications built for iOS 4.0 and later releases. In concrete terms, that means that touches for applications built for iOS 4 running on iPhone 4 can land on half-point boundaries where on older devices they land only on full point boundaries. If you have any round-to-integer code in your touch-handling path you may lose this precision.

Many classes in UIKit handle multitouch 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.

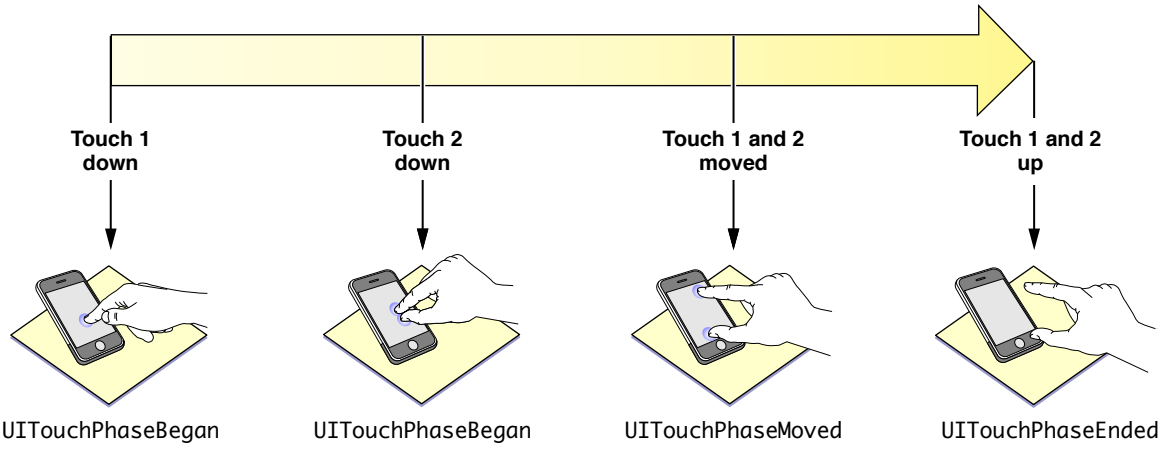
Some applications may not need to handle events directly; instead, they can rely on the classes of UIKit for that behavior. However, if you create a custom subclass of `UIView`—a common pattern in iOS development—and if you want that view to respond to certain touch events, you need to implement the 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 iOS, a *touch* is the presence or movement of a finger on the screen that is part of a unique multitouch 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, a drag, 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.

A `UIEvent` object of type `UIEventTypeTouches` represents a touch event. The system continually sends these touch-event objects (or simply, touch events) to an application as fingers touch the screen and move across its surface. The event provides a snapshot of all touches during a multitouch sequence, most importantly the touches that are new or have changed for a particular view. As depicted in Figure 2-1, a multitouch 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.

Figure 2-1 A multitouch sequence and touch phases

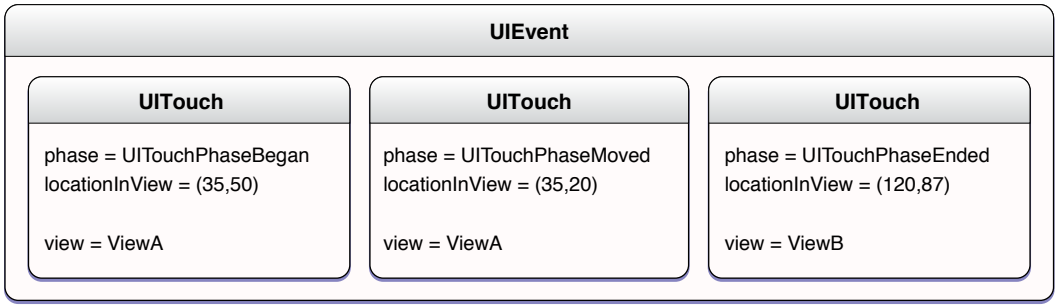


Touches, which are represented by `UITouch` objects, have both temporal and spatial aspects. The temporal aspect, called a phase, indicates when a touch has just begun, whether it is moving or stationary, and when it ends—that is, when the finger is lifted from the screen.

The spatial aspect of touches concerns their association with the object in which they occur as well as their location in it. When a finger touches the screen, the touch is associated with the underlying window and 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. A touch object stores the current location and previous location (if any) of the touch in its view or window.

An event object contains all touch objects for the current multitouch sequence and can provide touch objects specific to a view or window (see Figure 2-2). 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 may evaluate these attributes to determine how to respond to a touch event.

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



Because the system can cancel a multitouch sequence at any time, 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.

Approaches for Handling Touch Events

Most applications that are interested in users' touches on their custom views are interested in detecting and handling well-established gestures. These gestures include tapping (one or multiple times), pinching (to zoom a view in or out), swiping , panning or dragging a view, and using two fingers to rotate a view.

You could implement the touch-event handling code to recognize and handle these gestures, but that code would be complex, possibly buggy, and take some time to write. Alternatively, you could simplify the interpretation and handling of common gestures by using one of the gesture recognizer classes introduced in iOS 3.2. To use a gesture recognizer, you instantiate it, attach it to the view receiving touches, configure it, and assign it an action selector and a target object. When the gesture recognizer recognizes its gesture, it sends an action message to the target, allowing the target to respond to the gesture.

You can implement a custom gesture recognizer by subclassing `UIGestureRecognizer`. A custom gesture recognizer requires you to analyze the stream of events in a multitouch sequence to recognize your distinct gesture; to do this, you should be familiar with the information in this chapter.

For information about gesture recognizers, see "Gesture Recognizers" (page 39).

Regulating Touch Event Delivery

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

- **Turning off delivery of touch events**. By default, a view receives touch events, but you can set its `userInteractionEnabled` property to `NO` to turn off delivery of touch events. A view also does not receive these events if it's hidden or if it's transparent.
- **Turning off delivery of touch 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 events entirely; the second method is called to resume the receipt of such events. You sometimes want to turn off event delivery while your code is performing animations.
- **Turning on delivery of multiple touches**. By default, a view ignores all but the first touch during a multitouch sequence. If you want the view to handle multiple touches you must enable this capability for the view. You do this programmatically by setting the `multipleTouchEnabled` property of your view to `YES`, or in Interface Builder by setting the related attribute in the inspector for the related view.
- **Restricting event delivery to a single view**. By default, a view's `exclusiveTouch` property is set to `NO`, which means that this view does not block other views in a window from receiving touches. 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 multitouch events to its subviews. See "Hit-Testing" (page 33) for a discussion of this technique.

Handling Multitouch Events

To handle multitouch events, you must first create a subclass of a responder class. This subclass could be any one of the following:

- A custom view (subclass of `UIView`)
- A subclass of `UIViewController` or one of its UIKit subclasses
- A subclass of a UIKit view or control class, such as `UIImageView` or `UISlider`
- A subclass of `UIApplication` or `UIWindow` (although this would be rare)

A view controller typically receives, via the responder chain, touch events initially sent to its view if that view does not override the touch-handling methods.

For instances of your subclass to receive multitouch events, your subclass must implement one or more of the `UIResponder` methods for touch-event handling, described below. In addition, the view must be visible (neither transparent or hidden) and must have its `userInteractionEnabled` property set to `YES`, which is the default.

The following sections describe the touch-event handling methods, describe approaches for handling common gestures, show an example of a responder object that handles a complex sequence of multitouch events, discuss event forwarding, and suggest some techniques for event handling.

The Event-Handling Methods

During a multitouch sequence, the application dispatches a series of event messages to the target responder. To receive and handle these messages, the class of a responder object must implement at least one of the following methods declared by `UIResponder`, and, in some cases, all of these methods:

```
- (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, `touchesBegan:withEvent:` is associated with `UITouchPhaseBegan`. You can get the phase of any `UITouch` object 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 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.

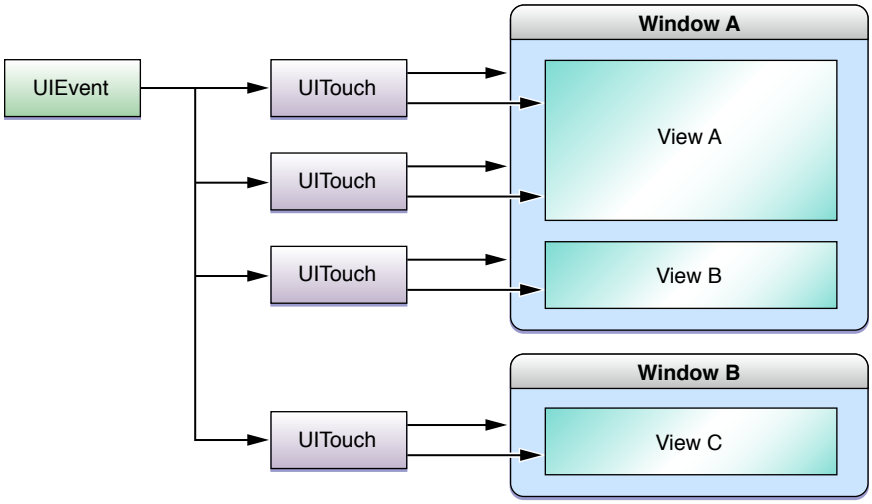
Basics of Touch-Event Handling

You frequently handle an event for a given phase by getting one or more of the `UITouch` objects in the passed-in set, evaluating their properties or getting their locations, and proceeding accordingly. The objects in the set represent those touches that are new or have changed for the phase represented by the implemented event-handling method. If any of the touch objects will do, you can send the `NSSet` object an `anyObject` message; this is the case when the view receives only the first touch in a multitouch sequence (that is, the `multipleTouchEnabled` property is set to `NO`).

An important `UITouch` method is `locationInView:`, which, if passed a parameter of `self`, yields the location of the touch in the coordinate system of the receiving view. 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`).

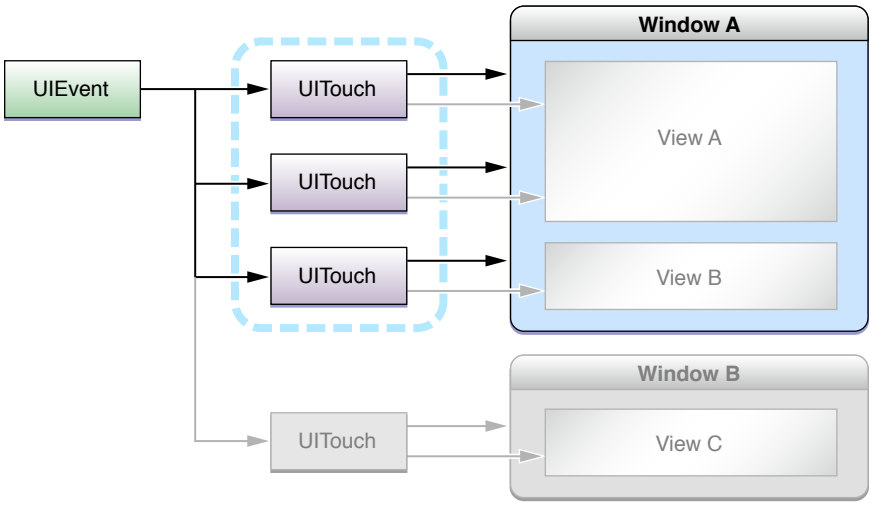
If for some reason you are interested in touches in the current multitouch sequence that have not changed since the last phase or that are in a phase other than the ones in the passed-in set, you can request them from the passed-in `UIEvent` object. The diagram in Figure 2-3 depicts a `UIEvent` object that contains four touch objects. To get all these touch objects, you would invoke the `allTouches` on the event object.

Figure 2-3 All touches for a given touch event



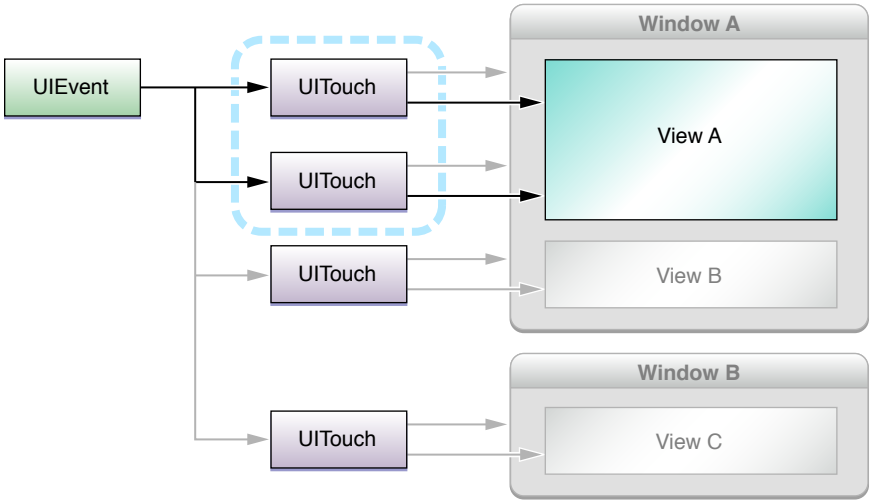
If on the other hand you are interested in only those touches associated with a specific window (Window A in Figure 2-4), you would send the `UIEvent` object a `touchesForWindow:` message.

Figure 2-4 All touches belonging to a specific window



If you want to get the touches associated with a specific view, you would call `touchesForView:` on the event object, passing in the view object (View A in Figure 2-5).

Figure 2-5 All touches belonging to a specific view



If a responder creates persistent objects while handling events during a multitouch 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 multitouch sequence. (See "Forwarding Touch Events" (page 35) to find out how to determine the last `UITouchPhaseEnded` touch object in a multitouch sequence.)

Important If your custom responder class is a subclass of `UIView` or `UIViewController`, you should implement all of the methods described in "The Event-Handling Methods" (page 20). If your class is a subclass of any other UIKit responder class, you do not need to override all of the event-handling methods; however, in those methods that you do override, be sure to call the superclass implementation of the method (for example, `super touchesBegan:touches withEvent:theEvent];`). The reason for this guideline is simple: All views that process touches, including your own, expect (or should expect) to receive a full touch-event stream. If you prevent a UIKit responder object from receiving touches for a certain phase of an event, the resulting behavior may be undefined and probably undesirable.

Handling Tap Gestures

A very common gesture in iOS applications is the tap: the user taps an object on the screen with his or her finger. A responder object can handle 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.

Listing 2-1 shows how to determine whether a double-tap occurred in one of your views.

Listing 2-1 Detecting a double-tap gesture

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

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

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {
        if (touch.tapCount >= 2) {
            [self.superview bringSubviewToFront:self];
        }
    }
```

```
}

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

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? Listing 2-2 (page 25) illustrates an implementation of the event-handling methods that increases the size of the receiving view upon a double-tap gesture and decreases it upon a single-tap gesture.

The following is a commentary on this code:

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 second parameter is an `NSValue` or `NSDictionary` object that holds some state of the `UITouch` object; the delay is some reasonable interval between a single- and a double-tap gesture.

   Note Because a touch object is mutated as it proceeds through a multitouch sequence, you cannot retain a touch and assume that its state remains the same. (And you cannot copy a touch object because `UITouch` does not adopt the `NSCopying` protocol.) Thus if you want to preserve the state of a touch object, you should store those bits of state in a `NSValue` object, a dictionary, or a similar object. (The code in Listing 2-2 stores the location of the touch in a dictionary but does not use it; this code is included for purposes of illustration.)

2. In `touchesBegan:withEvent:`, if the tap count is two, the responder object cancels the pending delayed-perform invocation by calling the `cancelPreviousPerformRequestsWithTarget:` method of `NSObject`, passing itself as the argument. 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.

Listing 2-2 Handling a single-tap gesture and a double-tap gesture

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    if (aTouch.tapCount == 2) {
```

```
        [NSObject cancelPreviousPerformRequestsWithTarget:self];
    }
}

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

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    if (theTouch.tapCount == 1) {
        NSDictionary *touchLoc = [NSDictionary dictionaryWithObject:
            [NSValue valueWithCGPoint:[theTouch locationInView:self]]
forKey:@"location"];
        [self performSelector:@selector(handleSingleTap:) withObject:touchLoc
afterDelay:0.3];
    } else if (theTouch.tapCount == 2) {
        // Double-tap: increase image size by 10%"
        CGRect myFrame = self.frame;
        myFrame.size.width += self.frame.size.width * 0.1;
        myFrame.size.height += self.frame.size.height * 0.1;
        myFrame.origin.x -= (self.frame.origin.x * 0.1) / 2.0;
        myFrame.origin.y -= (self.frame.origin.y * 0.1) / 2.0;
        [UIView beginAnimations:nil context:NULL];
        [self setFrame:myFrame];
        [UIView commitAnimations];
    }
}

- (void)handleSingleTap:(NSDictionary *)touches {
    // Single-tap: decrease image size by 10%"
    CGRect myFrame = self.frame;
    myFrame.size.width -= self.frame.size.width * 0.1;
    myFrame.size.height -= self.frame.size.height * 0.1;
    myFrame.origin.x += (self.frame.origin.x * 0.1) / 2.0;
    myFrame.origin.y += (self.frame.origin.y * 0.1) / 2.0;
```

```
    [UIView beginAnimations:nil context:NULL];
    [self setFrame:myFrame];
    [UIView commitAnimations];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    /* no state to clean up, so null implementation */
}
```

Handling Swipe and Drag 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 whether 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 2-3 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` instance variable. As the user's finger moves, the code compares the current touch location to the starting location to determine whether 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 2-3 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 is an instance variable
    startTouchPosition = [touch locationInView:self];
}
```

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint currentTouchPosition = [touch locationInView:self];

    // To be a swipe, direction of touch must be horizontal and long enough.
   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];
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = CGPointZero;
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = CGPointZero;
}
```

Listing 2-4 shows an even simpler implementation of tracking a single touch, but this time for the purposes of dragging the receiving view around the screen. In this instance, the responder class fully implements only the `touchesMoved:withEvent:` method, and in this method computes a delta value between the touch's current location in the view and its previous location in the view. It then uses this delta value to reset the origin of the view's frame.

Listing 2-4 Dragging a view using a single touch

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

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    CGPoint loc = [aTouch locationInView:self];
    CGPoint prevloc = [aTouch previousLocationInView:self];

    CGRect myFrame = self.frame;
    float deltaX = loc.x - prevloc.x;
    float deltaY = loc.y - prevloc.y;
    myFrame.origin.x += deltaX;
    myFrame.origin.y += deltaY;
    [self setFrame:myFrame];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
}

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

Handling a Complex Multitouch Sequence

Taps, drags, and swipes are simple gestures, typically involving only a single touch. Handling a touch event consisting of two or more touches is a more complicated affair. You may have to track all touches through all phases, recording the touch attributes that have changed and altering internal state appropriately. There are a couple of things you should do when tracking and handling multiple touches:

- Set the `multipleTouchEnabled` property of the view to `YES`.
- Use a Core Foundation dictionary object (`CFDictionaryRef`) to track the mutations of touches through their phases during the event.

When handling an event with multiple touches, you often store initial bits of each touch's 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 `CFDictionaryRef` 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 `CFDictionaryRef` type 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.)

Listing 2-5 illustrates how you might store beginning locations of `UITouch` objects in a Core Foundation dictionary.

Listing 2-5 Storing the beginning locations of multiple touches

```
- (void)cacheBeginPointForTouches:(NSSet *)touches
{
    if ([touches count] > 0) {
        for (UITouch *touch in touches) {
            CGPoint *point = (CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch);
            if (point == NULL) {
                point = (CGPoint *)malloc(sizeof(CGPoint));
                CFDictionarySetValue(touchBeginPoints, touch, point);
            }
            *point = [touch locationInView:view.superview];
        }
    }
}
```

Listing 2-6 illustrates how to retrieve those initial locations stored in the dictionary. It also gets the current locations of the same touches. It uses these values in computing an affine transformation (not shown).

Listing 2-6 Retrieving the initial locations of touch objects

```
- (CGAffineTransform)incrementalTransformWithTouches:(NSSet *)touches {
   NSArray *sortedTouches = [[touches allObjects]
sortedArrayUsingSelector:@selector(compareAddress:)];

   // other code here ...

    UITouch *touch1 = [sortedTouches objectAtIndex:0];
    UITouch *touch2 = [sortedTouches objectAtIndex:1];
```

```
    CGPoint beginPoint1 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch1);
    CGPoint currentPoint1 = [touch1 locationInView:view.superview];
    CGPoint beginPoint2 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch2);
    CGPoint currentPoint2 = [touch2 locationInView:view.superview];

    // compute the affine transform...
}
```

Although the code example in Listing 2-7 doesn't use a dictionary to track touch mutations, it also handles multiple touches during an event. It shows a custom `UIView` object responding 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 2-7 Handling a complex multitouch 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 *View Programming Guide for iOS*.

To find out when the last finger in a multitouch 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. If they are the same, then the multitouch sequence has concluded. Listing 2-8 illustrates how to do this in code.

Listing 2-8 Determining when the last touch in a multitouch sequence has ended

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    if ([touches count] == [[event touchesForView:self] count]) {
        // last finger has lifted....
    }
}
```

Remember that the passed-in set contains all touch objects associated with the receiving view that are new or changed for the given phase whereas the touch objects returned from `touchesForView:` includes *all* objects associated with the specified view.

Hit-Testing

Your custom responder can use hit-testing to find the subview or sublayer of itself that is "under" a touch, and then handle the event appropriately. It does this by either calling the `hitTest:withEvent:` method of `UIView` or the `hitTest:` method of `CALayer`; or it can override one of these methods. Responders sometimes perform hit-testing prior to event forwarding (see "Forwarding Touch Events" (page 35)).

> **Note** The `hitTest:withEvent:` and `hitTest:` methods have some slightly different behaviors.

If the point passed into `hitTest:withEvent:` or `hitTest:` is outside the bounds of the view, it is ignored. This means that subviews that are outside their superview do not receive touch events.

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.

The example in Listing 2-9 detects when an "Info" image in a layer of the custom view is tapped.

Listing 2-9 Calling `hitTest:` on a view's CALayer object

```
- (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];
    }
}
```

In Listing 2-10, a responder subclass (in this case, a subclass of `UIWindow`) overrides `hitTest:withEvent:`. It first gets the hit-test view returned by the superclass. Then, if that view is itself, it substitutes the view that is furthest down the view hierarchy.

Listing 2-10 Overriding `hitTest:withEvent:`

```
- (UIView*)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
    UIView *hitView = [super hitTest:point withEvent:event];

    if (hitView == self)
        return [[self subviews] lastObject];
    else
        return hitView;

}
```

Forwarding Touch Events

Event forwarding is a technique used by some applications. You forward touch events by invoking the event-handling methods of another responder object. Although this can be an effective technique, you should use it with caution. The classes of the UIKit framework are not designed to receive touches that are not bound to them; in programmatic terms, this means that the `view` property of the `UITouch` object must hold a reference to the framework object in order for the touch to be handled. If you want to conditionally forward touches to other responders in your application, all of these responders should be instances of your own subclasses of `UIView`.

For example, let's say an application has three custom views: A, B, and C. When the user touches view A, the application's window determines that it is the hit-test view and sends the initial touch event to it. Depending on certain conditions, view A forwards the event to either view B or view C. In this case, views A, B, and C must be aware that this forwarding is going on, and views B and C must be able to deal with touches that are not bound to them.

Event forwarding often requires analysis of touch objects to determine where they should be forwarded. There are several approaches you can take for this analysis:

- With an "overlay" view (such as a common superview), use hit-testing to intercept events for analysis prior to forwarding them to subviews (see "Hit-Testing" (page 33)).
- Override `sendEvent:` in a custom subclass of `UIWindow`, analyze touches, and forward them to the appropriate responders. In your implementation you should always invoke the superclass implementation of `sendEvent:`.
- Design your application so that touch analysis isn't necessary

Listing 2-11 illustrates the second technique, that of overriding `sendEvent:` in a subclass of `UIWindow`. In this example, the object to which touch events are forwarded is a custom "helper" responder that performs affine transformations on the view that is associated with.

Listing 2-11 Forwarding touch events to "helper" responder objects

```
- (void)sendEvent:(UIEvent *)event
{
    for (TransformGesture *gesture in transformGestures) {
        // collect all the touches we care about from the event
        NSSet *touches = [gesture observedTouchesForEvent:event];
        NSMutableSet *began = nil;
        NSMutableSet *moved = nil;
        NSMutableSet *ended = nil;
```

```
        NSMutableSet *cancelled = nil;

        // sort the touches by phase so we can handle them similarly to normal
event dispatch
        for(UITouch *touch in touches) {
            switch ([touch phase]) {
                case UITouchPhaseBegan:
                    if (!began) began = [NSMutableSet set];
                    [began addObject:touch];
                    break;
                case UITouchPhaseMoved:
                    if (!moved) moved = [NSMutableSet set];
                    [moved addObject:touch];
                    break;
                case UITouchPhaseEnded:
                    if (!ended) ended = [NSMutableSet set];
                    [ended addObject:touch];
                    break;
                case UITouchPhaseCancelled:
                    if (!cancelled) cancelled = [NSMutableSet set];
                    [cancelled addObject:touch];
                    break;
                default:
                    break;
            }
        }
        // call our methods to handle the touches
        if (began)     [gesture touchesBegan:began withEvent:event];
        if (moved)     [gesture touchesMoved:moved withEvent:event];
        if (ended)     [gesture touchesEnded:ended withEvent:event];
        if (cancelled) [gesture touchesCancelled:cancelled withEvent:event];
    }
    [super sendEvent:event];
}
```

Notice that in this example that the overriding subclass does something important to the integrity of the touch-event stream: It invokes the superclass implementation of `sendEvent:`.

Handling Events in Subclasses of UIKit Views and Controls

If you subclass a view or control class of the UIKit framework (for example, `UIImageView` or `UISwitch`) for the purpose of altering or extending event-handling behavior, you should keep the following points in mind:

- Unlike in a custom view, it is not necessary to override each event-handling method.
- Always invoke the superclass implementation of each event-handling method that you do override.
- Do not forward events to UIKit framework objects.

Best Practices for Handling Multitouch Events

When handling events, both touch events and motion events, there are a few recommended techniques and patterns you should follow.

- Always implement the event-cancellation methods.

  In your implementation, you should restore the state of the view to what it was before the current multitouch sequence, freeing any transient resources set up for handling the event. If you don't implement the cancellation method your view could be left in an inconsistent state. In some cases, another view might receive the cancellation message.
- If you handle events in a subclass of `UIView`, `UIViewController`, or (in rare cases) `UIResponder`,
  - You should implement all of the event-handling methods (even if it is a null implementation).
  - Do not call the superclass implementation of the methods.
- If you handle events in a subclass of any other UIKit responder class,
  - You do not have to implement all of the event-handling methods.
  - But in the methods you do implement, be sure to call the superclass implementation. For example,

    ```
    [super touchesBegan:theTouches withEvent:theEvent];
    ```
- Do not forward events to other responder objects of the UIKit framework.

  The responders that you forward events to should be instances of your own subclasses of `UIView`, and all of these objects must be aware that event-forwarding is taking place and that, in the case of touch events, they may receive touches that are not bound to them.
- Custom views that redraw themselves in response to events should only set drawing state in the event-handling methods and perform all of the drawing in the `drawRect:` method.

- Do not explicitly send events up the responder (via `nextResponder`); instead, invoke the superclass implementation and let the UIKit handle responder-chain traversal.

Gesture Recognizers

Note This chapter contains information that used to be in *iPad Programming Guide* . The information in this chapter has not been updated specifically for iOS 4.0.

Applications for iOS are driven largely through events generated when users touch buttons, toolbars, table-view rows and other objects in an application's user interface. The classes of the UIKit framework provide default event-handling behavior for most of these objects. However, some applications, primarily those with custom views, have to do their own event handling. They have to analyze the stream of touch objects in a multitouch sequence and determine the intention of the user.

Most event-handling views seek to detect common gestures that users make on their surface—things such as triple-tap, touch-and-hold (also called long press), pinching, and rotating gestures, The code for examining a raw stream of multitouch events and detecting one or more gestures is often complex. Prior to iOS 3.2, you cannot reuse the code except by copying it to another project and modifying it appropriately.

To help applications detect gestures, iOS 3.2 introduces gesture recognizers, objects that inherit directly from the `UIGestureRecognizer` class. The following sections tell you about how these objects work, how to use them, and how to create custom gesture recognizers that you can reuse among your applications.

Gesture Recognizers Simplify Event Handling

`UIGestureRecognizer` is the abstract base class for concrete gesture-recognizer subclasses (or, simply, gesture recognizers). The `UIGestureRecognizer` class defines a programmatic interface and implements the behavioral underpinnings for gesture recognition. The UIKit framework provides six gesture recognizers for the most common gestures. For other gestures, you can design and implement your own gesture recognizer (see "Creating Custom Gesture Recognizers" (page 50) for details).

Recognized Gestures

The UIKit framework supports the recognition of the gestures listed in Table 3-1. Each of the listed classes is a direct subclass of `UIGestureRecognizer`.

Table 3-1 Gestures recognized by the gesture-recognizer classes of the UIKit framework

Gesture	UIKit class
Tapping (any number of taps)	`UITapGestureRecognizer`
Pinching in and out (for zooming a view)	`UIPinchGestureRecognizer`
Panning or dragging	`UIPanGestureRecognizer`
Swiping (in any direction)	`UISwipeGestureRecognizer`
Rotating (fingers moving in opposite directions)	`UIRotationGestureRecognizer`
Long press (also known as "touch and hold")	`UILongPressGestureRecognizer`

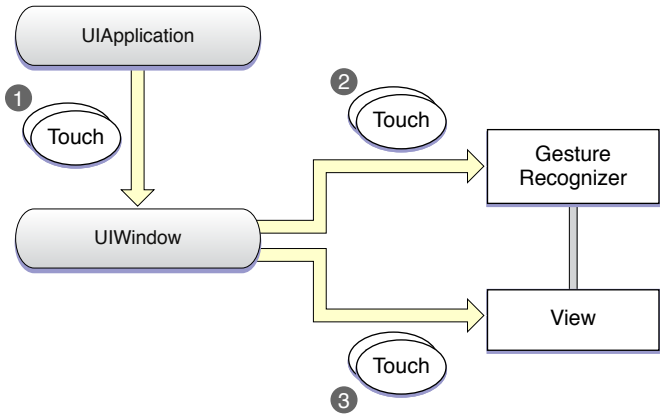
Before you decide to use a gesture recognizer, consider how you are going to use it. Respond to gestures only in ways that users expect. For example, a pinching gesture should scale a view, zooming it in and out; it should not be interpreted as, say, a selection request, for which a tap is more appropriate. For guidelines about the proper use of gestures, see *iOS Human Interface Guidelines*.

Gestures Recognizers Are Attached to a View

To detect its gestures, a gesture recognizer must be attached to the view that a user is touching. This view is known as the *hit-tested view*. Recall that events in iOS are represented by `UIEvent` objects, and each event object encapsulates the `UITouch` objects of the current multitouch sequence. A set of those `UITouch` objects is specific to a given phase of a multitouch sequence. Delivery of events initially follows the usual path: from operating system to the application object to the window object representing the window in which the touches

are occurring. But before sending an event to the hit-tested view, the window object sends it to the gesture recognizer attached to that view or to any of that view's subviews. Figure 3-1 illustrates this general path, with the numbers indicating the order in which touches are received.

Figure 3-1 Path of touch objects when gesture recognizer is attached to a view



Thus gesture recognizers act as observers of touch objects sent to their attached view or view hierarchy. However, they are not part of that view hierarchy and do not participate in the responder chain. Gesture recognizers may delay the delivery of touch objects to the view while they are recognizing gestures, and by default they cancel delivery of remaining touch objects to the view once they recognize their gesture. For more on the possible scenarios of event delivery from a gesture recognizer to its view, see "Regulating the Delivery of Touches to Views" (page 48).

For some gestures, the `locationInView:` and the `locationOfTouch:inView:` methods of `UIGestureRecognizer` enable clients to find the location of gestures or specific touches in the attached view or its subviews. See "Responding to Gestures" (page 44) for more information.

Gestures Trigger Action Messages

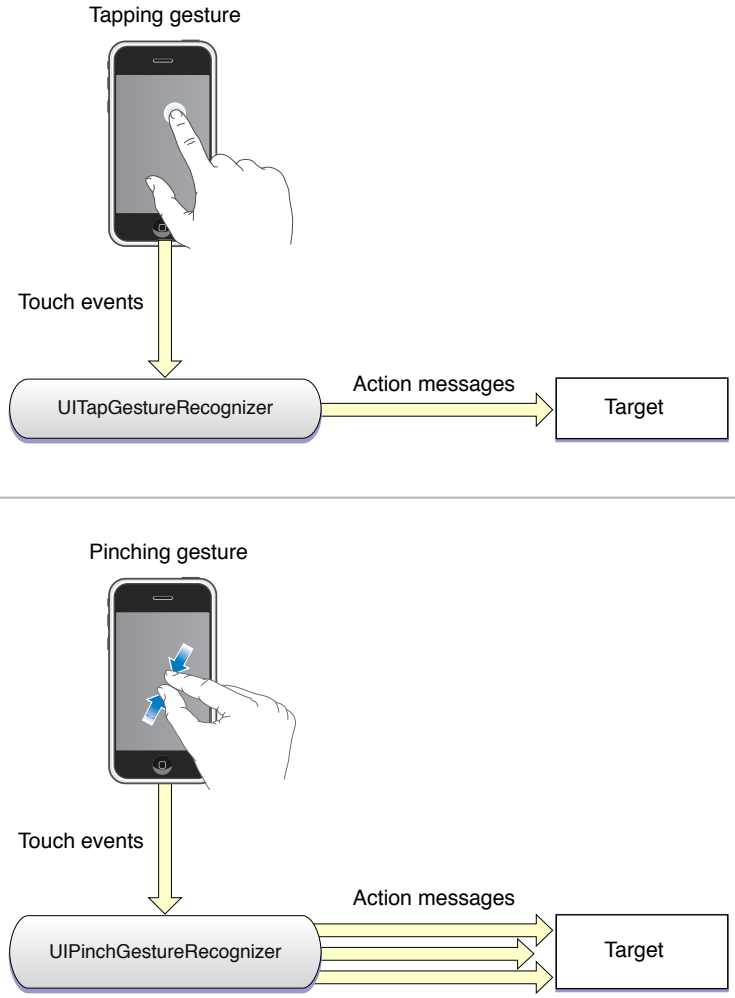
When a gesture recognizer recognizes its gesture, it sends one or more action messages to one or more targets. When you create a gesture recognizer, you initialize it with an action and a target. You may add more target-action pairs to it thereafter. The target-action pairs are not additive; in other words, an action is only sent to the target it was originally linked with, and not to other targets (unless they're specified in another target-action pair).

Discrete Gestures and Continuous Gestures

When a gesture recognizer recognizes a gesture, it sends either a single action message to its target or multiple action messages until the gesture ends. This behavior is determined by whether the gesture is discrete or continuous. A discrete gesture, such as a double-tap, happens just once; when a gesture recognizer recognizes

a discrete gesture, it sends its target a single action message. A continuous gesture, such as pinching, takes place over a period and ends when the user lifts the final finger in the multitouch sequence. The gesture recognizer sends action messages to its target at short intervals until the multitouch sequence ends.

Figure 3-2 Discrete versus continuous gestures



The reference documents for the gesture-recognizer classes note whether the instances of the class detect discrete or continuous gestures.

Implementing Gesture Recognition

To implement gesture recognition, you create a gesture-recognizer instance to which you assign a target, action, and, in some cases, gesture-specific attributes. You attach this object to a view and then implement the action method in your target object that handles the gesture.

Preparing a Gesture Recognizer

To create a gesture recognizer, you must allocate and initialize an instance of a concrete `UIGestureRecognizer` subclass. When you initialize it, specify a target object and an action selector, as in the following code:

```
UITapGestureRecognizer *doubleFingerDTap = [[UITapGestureRecognizer alloc]
   initWithTarget:self action:@selector(handleDoubleDoubleTap:)];
```

The action methods for handling gestures—and the selector for identifying them—are expected to conform to one of two signatures:

- `(void)`*handleGesture*
- `(void)`*handleGesture*`:(UIGestureRecognizer *)`*sender*

where *handleGesture* and *sender* can be any name you choose. Methods having the second signature allow the target to query the gesture recognizer for addition information. For example, the target of a `UIPinchGestureRecognizer` object can ask that object for the current scale factor related to the pinching gesture.

After you create a gesture recognizer, you must attach it to the view receiving touches—that is, the hit-test view—using the `UIView` method `addGestureRecognizer:`. You can find out what gesture recognizers a view currently has attached through the `gestureRecognizers` property, and you can detach a gesture recognizer from a view by calling `removeGestureRecognizer:`.

The sample method in Listing 3-1 creates and initializes three gesture recognizers: a single-finger double-tap, a panning gesture, and a rotation gesture. It then attaches each gesture-recognizer object to the same view. For the `singleFingerDTap` object, the code specifies that two taps are required for the gesture to be recognized. Each method adds the created gesture recognizer to a view and then releases it (because the view now retains it).

Listing 3-1 Creating and initializing discrete and continuous gesture recognizers

```
- (void)createGestureRecognizers {
    UITapGestureRecognizer *singleFingerDTap = [[UITapGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleSingleDoubleTap:)];
    singleFingerDTap.numberOfTapsRequired = 2;
    [self.theView addGestureRecognizer:singleFingerDTap];
    [singleFingerDTap release];

    UIPanGestureRecognizer *panGesture = [[UIPanGestureRecognizer alloc]
```

```
        initWithTarget:self action:@selector(handlePanGesture:)];
    [self.theView addGestureRecognizer:panGesture];
    [panGesture release];

    UIPinchGestureRecognizer *pinchGesture = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinchGesture:)];
    [self.theView addGestureRecognizer:pinchGesture];
    [pinchGesture release];
}
```

You may also add additional targets and actions to a gesture recognizer using the `addTarget:action:` method of `UIGestureRecognizer`. Remember that action messages for each target and action pair are restricted to that pair; if you have multiple targets and actions, they are not additive.

Responding to Gestures

To handle a gesture, the target for the gesture recognizer must implement a method corresponding to the action selector specified when you initialized the gesture recognizer. For discrete gestures, such as a tapping gesture, the gesture recognizer invokes the method once per recognition; for continuous gestures, the gesture recognizer invokes the method at repeated intervals until the gesture ends (that is, the last finger is lifted from the gesture recognizer's view).

In gesture-handling methods, the target object often gets additional information about the gesture from the gesture recognizer; it does this by obtaining the value of a property defined by the gesture recognizer, such as `scale` (for scale factor) or `velocity`. It can also query the gesture recognizer (in appropriate cases) for the location of the gesture.

Listing 3-2 shows handlers for two continuous gestures: a pinching gesture (`handlePinchGesture:`) and a panning gesture (`handlePanGesture:`). It also gives an example of a handler for a discrete gesture; in this example, when the user double-taps the view with a single finger, the handler (`handleSingleDoubleTap:`) centers the view at the location of the double-tap.

Listing 3-2 Handling pinch, pan, and double-tap gestures

```
- (IBAction)handlePinchGesture:(UIGestureRecognizer *)sender {
    CGFloat factor = [(UIPinchGestureRecognizer *)sender scale];
    self.view.transform = CGAffineTransformMakeScale(factor, factor);
}
```

```
- (IBAction)handlePanGesture:(UIPanGestureRecognizer *)sender {
    CGPoint translate = [sender translationInView:self.view];

    CGRect newFrame = currentImageFrame;
    newFrame.origin.x += translate.x;
    newFrame.origin.y += translate.y;
    sender.view.frame = newFrame;

    if (sender.state == UIGestureRecognizerStateEnded)
        currentImageFrame = newFrame;
}

- (IBAction)handleSingleDoubleTap:(UIGestureRecognizer *)sender {
    CGPoint tapPoint = [sender locationInView:sender.view.superview];
    [UIView beginAnimations:nil context:NULL];
    sender.view.center = tapPoint;
    [UIView commitAnimations];
}
```

These action methods handle the gestures in distinctive ways:

- In the `handlePinchGesture:` method, the target communicates with its gesture recognizer (`sender`) to get the scale factor (`scale`). The method uses the scale value in a Core Graphics function that scales the view and assigns the computed value to the view's affine `transform` property.
- The `handlePanGesture:` method applies the `translationInView:` values obtained from its gesture recognizer to a cached frame value for the attached view. When the gesture concludes, it caches the newest frame value.
- In the `handleSingleDoubleTap:` method, the target gets the location of the double-tap gesture from its gesture recognizer by calling the `locationInView:` method. It then uses this point, converted to superview coordinates, to animate the center of the view to the location of the double-tap.

The scale factor obtained in the `handlePinchGesture:` method, as with the rotation angle and the translation value related to other recognizers of continuous gestures, is to be applied to the state of the view when the gesture is first recognized. It is not a delta value to be concatenated over each handler invocation for a given gesture.

A hit-test with an attached gesture recognizer does not have to be passive when there are incoming touch events. Instead, it can determine which gesture recognizers, if any, are involved with a particular `UITouch` object by querying the `gestureRecognizers` property. Similarly, it can find out which touches a given gesture recognizer is analyzing for a given event by calling the `UIEvent` method `touchesForGestureRecognizer:`.

Interacting with Other Gesture Recognizers

More than one gesture recognizer may be attached to a view. In the default behavior, touch events in a multitouch sequence go from one gesture recognizer to another in a nondeterministic order until the events are finally delivered to the view (if at all). Often this default behavior is what you want. But sometimes you might want one or more of the following behaviors:

- Have one gesture recognizer fail before another can start analyzing touch events.
- Prevent other gesture recognizers from analyzing a specific multitouch sequence or a touch object in that sequence.
- Permit two gesture recognizers to operate simultaneously.

The `UIGestureRecognizer` class provides client methods, delegate methods, and methods overridden by subclasses to enable you to effect these behaviors.

Requiring a Gesture Recognizer to Fail

You might want a relationship between two gesture recognizers so that one can operate only if the other one fails. For example, recognizer A doesn't begin analyzing a multitouch sequence until recognizer B fails and, conversely, if recognizer B does recognize its gesture, recognizer A never looks at the multitouch sequence. An example where you might specify this relationship is when you have a gesture recognizer for a single tap and another gesture recognizer for a double tap; the single-tap recognizer requires the double-tap recognizer to fail before it begins operating on a multitouch sequence.

The method you call to specify this relationship is `requireGestureRecognizerToFail:`. After sending the message, the receiving gesture recognizer must stay in the `UIGestureRecognizerStatePossible` state until the specified gesture recognizer transitions to `UIGestureRecognizerStateFailed`. If the specified gesture recognizer transitions to `UIGestureRecognizerStateRecognized` or `UIGestureRecognizerStateBegan` instead, then the receiving recognizer can proceed, but no action message is sent if it recognizes its gesture.

Note In the case of the single-tap versus double-tap gestures, if a single-tap gesture recognizer doesn't require the double-tap recognizer to fail, you should expect to receive your single-tap actions before your double-tap actions, even in the case of a double tap. This is expected and desirable behavior because the best user experience generally involves stackable actions. If you want double-tap and single-tap gesture recognizers to have mutually exclusive actions, you can require the double-tap recognizer to fail. You won't get any single-tap actions on a double tap, but any single-tap actions you do receive will necessarily lag behind the user's touch input. In other words, there is no way to know if the user double tapped until after the double-tap delay, so the single-tap gesture recognizer cannot send its action until that delay has passed.

For a discussion of gesture-recognition states and possible transition between these states, see "State Transitions" (page 50).

Preventing Gesture Recognizers from Analyzing Touches

You can prevent gesture recognizers from looking at specific touches or from even recognizing a gesture. You can specify these "prevention" relationships using either delegation methods or overriding methods declared by the `UIGestureRecognizer` class.

The `UIGestureRecognizerDelegate` protocol declares two optional methods that prevent specific gesture recognizers from recognizing gestures on a case-by-case basis:

- `gestureRecognizerShouldBegin:` — This method is called when a gesture recognizer attempts to transition out of `UIGestureRecognizerStatePossible`. Return `NO` to make it transition to `UIGestureRecognizerStateFailed` instead. (The default value is `YES`.)
- `gestureRecognizer:shouldReceiveTouch:` —This method is called before the window object calls `touchesBegan:withEvent:` on the gesture recognizer when there are one or more new touches. Return `NO` to prevent the gesture recognizer from seeing the objects representing these touches. (The default value is `YES`.)

In addition, there are two `UIGestureRecognizer` methods (declared in `UIGestureRecognizerSubclass.h`) that effect the same behavior as these delegation methods. A subclass can override these methods to define class-wide prevention rules:

```
- (BOOL)canPreventGestureRecognizer:(UIGestureRecognizer
*)preventedGestureRecognizer;

- (BOOL)canBePreventedByGestureRecognizer:(UIGestureRecognizer
*)preventingGestureRecognizer;
```

Permitting Simultaneous Gesture Recognition

By default, no two gesture recognizers can attempt to recognize their gestures simultaneously. But you can change this behavior by implementing `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:`, an optional method of the `UIGestureRecognizerDelegate` protocol. This method is called when the recognition of the receiving gesture recognizer would block the operation of the specified gesture recognizer, or vice versa. Return `YES` to allow both gesture recognizers to recognize their gestures simultaneously.

> **Note** Returning YES is guaranteed to allow simultaneous recognition, but returning NO is not guaranteed to prevent simultaneous recognition because the other gesture's delegate may return YES.

Regulating the Delivery of Touches to Views

Generally, a window delivers `UITouch` objects (packaged in `UIEvent` objects) to a gesture recognizer before it delivers them to the attached hit-test view. But there are some subtle detours and dead-ends in this general delivery path that depend on whether a gesture is recognized. You can alter this delivery path to suit the requirements of your application.

Default Touch-Event Delivery

By default a window in a multitouch sequence delays the delivery of touch objects in Ended phases to the hit-test view and, if the gesture is recognized, both prevents the delivery of current touch objects to the view and cancels touch objects previously received by the view. The exact behavior depends on the phase of touch objects and on whether a gesture recognizer recognizes its gesture or fails to recognize it in a multitouch sequence.

To clarify this behavior, consider a hypothetical gesture recognizer for a discrete gesture involving two touches (that is, two fingers). Touch objects enter a system and are passed from the `UIApplication` object to the `UIWindow` object for the hit-test view. The following sequence occurs when the gesture is recognized:

1. The window sends two touch objects in the Began phase (`UITouchPhaseBegan`) to the gesture recognizer, which doesn't recognize the gesture. The window sends these same touches to the view attached to the gesture recognizer.
2. The window sends two touch objects in the Moved phase (`UITouchPhaseMoved`) to the gesture recognizer, and the recognizer still doesn't detect its gesture. The window then sends these touches to the attached view.

3. The window sends *one* touch object in the Ended phase (`UITouchPhaseEnded`) to the gesture recognizer. This touch object doesn't yield enough information for the gesture, but the window withholds the object from the attached view.
4. The window sends the other touch object in the Ended phase. The gesture recognizer now recognizes its gesture and so it sets its state to `UIGestureRecognizerStateRecognized`. Just before the first (or only) action message is sent, the view receives a `touchesCancelled:withEvent:` message to invalidate the touch objects previously sent (in the Began and Moved phases). The touches in the Ended phase are canceled.

Now assume that the gesture recognizer in the last step instead decides that this multitouch sequence it's been analyzing is *not* its gesture. It sets its state to `UIGestureRecognizerStateFailed`. The window then sends the two touch objects in the Ended phase to the attached view in a `touchesEnded:withEvent:` message.

A gesture recognizer for a continuous gesture goes through a similar sequence, except that it is more likely to recognize its gesture before touch objects reach the Ended phase. Upon recognizing its gesture, it sets its state to `UIGestureRecognizerStateBegan`. The window sends all subsequent touch objects in the multitouch sequence to the gesture recognizer but not to the attached view.

> **Note** For a discussion of gesture-recognition states and possible transition between these states, see "State Transitions" (page 50).

Affecting the Delivery of Touches to Views

You can change the values of three `UIGestureRecognizer` properties to alter the default delivery path of touch objects to views in certain ways. These properties and their default values are:

`cancelsTouchesInView` (default of `YES`)

`delaysTouchesBegan` (default of `NO`)

`delaysTouchesEnded` (default of `YES`)

If you change the default values of these properties, you get the following differences in behavior:

- `cancelsTouchesInView` set to `NO` — Causes `touchesCancelled:withEvent:` to *not* be sent to the view for any touches belonging to the recognized gesture. As a result, any touch objects in Began or Moved phases previously received by the attached view are not invalidated.
- `delaysTouchesBegan` set to `YES` — Ensures that when a gesture recognizer recognizes a gesture, no touch objects that were part of that gesture are delivered to the attached view. This setting provides a behavior similar to that offered by the `delaysContentTouches` property on `UIScrollView`; in this

case, when scrolling begins soon after the touch begins, subviews of the scroll-view object never receive the touch, so there is no flash of visual feedback. You should be careful about this setting because it can easily make your interface feel unresponsive.

- `delaysTouchesEnded` set to `NO` — Prevents a gesture recognizer that's recognized its gesture after a touch has ended from canceling that touch on the view. For example, say a view has a `UITapGestureRecognizer` object attached with its `numberOfTapsRequired` set to 2, and the user double-taps the view. If this property is set to `NO`, the view gets the following sequence of messages: `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, `touchesBegan:withEvent:`, and `touchesCancelled:withEvent:`. With the property set to `YES`, the view gets `touchesBegan:withEvent:`, `touchesBegan:withEvent:`, `touchesCancelled:withEvent:`, and `touchesCancelled:withEvent:`. The purpose of this property is to ensure that a view won't complete an action as a result of a touch that the gesture will want to cancel later.

Creating Custom Gesture Recognizers

If you are going to create a custom gesture recognizer, you need to have a clear understanding of how gesture recognizers work. The following section gives you the architectural background of gesture recognition, and the subsequent section goes into details of actually creating a gesture recognizer.

State Transitions

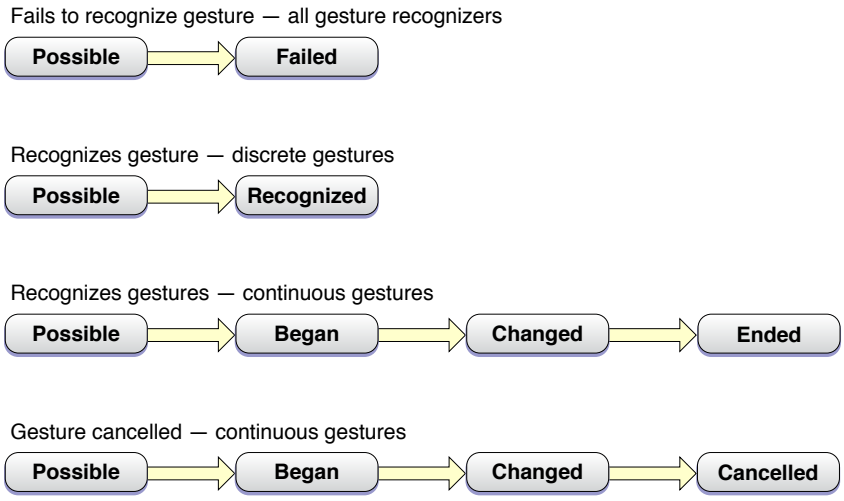
Gesture recognizers operate in a predefined state machine. They transition from one state to another depending on whether certain conditions apply. The following `enum` constants from `UIGestureRecognizer.h` define the states for gesture recognizers:

```
typedef enum {
    UIGestureRecognizerStatePossible,
    UIGestureRecognizerStateBegan,
    UIGestureRecognizerStateChanged,
    UIGestureRecognizerStateEnded,
    UIGestureRecognizerStateCancelled,
    UIGestureRecognizerStateFailed,
    UIGestureRecognizerStateRecognized = UIGestureRecognizerStateEnded
} UIGestureRecognizerState;
```

The sequence of states that a gesture recognizer may transition through varies, depending on whether a discrete or continuous gesture is being recognized. All gesture recognizers start in the Possible state (`UIGestureRecognizerStatePossible`). They then analyze the multitouch sequence targeted at their attached hit-test view, and they either recognize their gesture or fail to recognize it. If a gesture recognizer does not recognize its gesture, it transitions to the Failed state(`UIGestureRecognizerStateFailed`); this is true of all gesture recognizers, regardless of whether the gesture is discrete or continuous.

When a gesture is recognized, however, the state transitions differ for discrete and continuous gestures. A recognizer for a discrete gesture transitions from Possible to Recognized (`UIGestureRecognizerStateRecognized`). A recognizer for a continuous gesture, on the other hand, transitions from Possible to Began (`UIGestureRecognizerStateBegan`) when it first recognizes the gesture. Then it transitions from Began to Changed (`UIGestureRecognizerStateChanged`), and subsequently from Changed to Changed every time there is a change in the gesture. Finally, when the last finger in the multitouch sequence is lifted from the hit-test view, the gesture recognizer transitions to the Ended state (`UIGestureRecognizerStateEnded`), which is an alias for the `UIGestureRecognizerStateRecognized` state. A recognizer for a continuous gesture can also transition from the Changed state to a Cancelled state (`UIGestureRecognizerStateCancelled`) if it determines that the recognized gesture no longer fits the expected pattern for its gesture. Figure 3-3 illustrates these transitions.

Figure 3-3 Possible state transitions for gesture recognizers



> **Note** The Began, Changed, Ended, and Cancelled states are not necessarily associated with `UITouch` objects in corresponding touch phases. They strictly denote the phase of the gesture itself, not the touch objects that are being recognized.

When a gesture is recognized, every subsequent state transition causes an action message to be sent to the target. When a gesture recognizer reaches the Recognized or Ended state, it is asked to reset its internal state in preparation for a new attempt at recognizing the gesture. The `UIGestureRecognizer` class then sets the gesture recognizer's state back to Possible.

Implementing a Custom Gesture Recognizer

To implement a custom gesture recognizer, first create a subclass of `UIGestureRecognizer` in Xcode. Then, add the following import directive in your subclass's header file:

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

Next copy the following method declarations from `UIGestureRecognizerSubclass.h` to your header file; these are the methods you override in your subclass:

```
- (void)reset;
- (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;
```

You must be sure to call the superclass implementation (`super`) in all of the methods you override.

Examine the declaration of the `state` property in `UIGestureRecognizerSubclass.h`. Notice that it is now given a `readwrite` option instead of `readonly` (in `UIGestureRecognizer.h`). Your subclass can now change its state by assigning `UIGestureRecognizerState` constants to the property.

The `UIGestureRecognizer` class sends action messages for you and controls the delivery of touch objects to the hit-test view. You do not need to implement these tasks yourself.

Implementing the Multitouch Event-Handling Methods

The heart of the implementation for a gesture recognizer are the four methods `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:`. You implement these methods much as you would implement them for a custom view.

> **Note** See "Handling Multi-Touch Events" in *iOS App Programming Guide* in "Document Revision History" for information about handling events delivered during a multitouch sequence.

The main difference in the implementation of these methods for a gesture recognizer is that you transition between states at the appropriate moment. To do this, you must set the value of the state property to the appropriate `UIGestureRecognizerState` constant. When a gesture recognizer recognizes a discrete gesture, it sets the state property to `UIGestureRecognizerStateRecognized`. If the gesture is continuous, it sets the state property first to `UIGestureRecognizerStateBegan`; then, for each change in position of the gesture, it sets (or resets) the property to `UIGestureRecognizerStateChanged`. When the gesture ends, it sets `state` to `UIGestureRecognizerStateEnded`. If at any point a gesture recognizer realizes that this multitouch sequence is not its gesture, it sets its state to `UIGestureRecognizerStateFailed`.

Listing 3-3 is an implementation of a gesture recognizer for a discrete single-touch "checkmark" gesture (actually any V-shaped gesture). It records the midpoint of the gesture—the point at which the upstroke begins—so that clients can obtain this value.

Listing 3-3 Implementation of a "checkmark" gesture recognizer.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesBegan:touches withEvent:event];
    if ([touches count] != 1) {
        self.state = UIGestureRecognizerStateFailed;
        return;
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesMoved:touches withEvent:event];
    if (self.state == UIGestureRecognizerStateFailed) return;
    CGPoint nowPoint = [[touches anyObject] locationInView:self.view];
    CGPoint prevPoint = [[touches anyObject] previousLocationInView:self.view];
    if (!strokeUp) {
```

```
        // on downstroke, both x and y increase in positive direction
        if (nowPoint.x >= prevPoint.x && nowPoint.y >= prevPoint.y) {
            self.midPoint = nowPoint;
            // upstroke has increasing x value but decreasing y value
        } else if (nowPoint.x >= prevPoint.x && nowPoint.y <= prevPoint.y) {
            strokeUp = YES;
        } else {
            self.state = UIGestureRecognizerStateFailed;
        }
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesEnded:touches withEvent:event];
    if ((self.state == UIGestureRecognizerStatePossible) && strokeUp) {
        self.state = UIGestureRecognizerStateRecognized;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesCancelled:touches withEvent:event];
    self.midPoint = CGPointZero;
    strokeUp = NO;
    self.state = UIGestureRecognizerStateFailed;
}
```

If a gesture recognizer detects a touch (as represented by a `UITouch` object) that it determines is not part of its gesture, it can pass it on directly to its view. To do this, it calls `ignoreTouch:forEvent:` on itself, passing in the touch object. Ignored touches are not withheld from the attached view even if the value of the `cancelsTouchesInView` property is `YES`.

Resetting State

When your gesture recognizer transitions to either the `UIGestureRecognizerStateRecognized` state or the`UIGestureRecognizerStateEnded` state, the `UIGestureRecognizer` class calls the `reset` method of the gesture recognizer just before it winds back the gesture recognizer's state to

`UIGestureRecognizerStatePossible`. A gesture recognizer class should implement this method to reset any internal state so that it is ready for a new attempt at recognizing the gesture. After a gesture recognizer returns from this method, it receives no further updates for touches that have already begun but haven't ended.

Listing 3-4 Resetting a gesture recognizer

```
- (void)reset {
    [super reset];
    self.midPoint = CGPointZero;
    strokeUp = NO;
}
```

Motion Events

An iPhone, iPad, or iPod touch device generates motion events when users move the device in a certain way, such as shaking it or tilting it. All motion events have their origin in the device accelerometer or gyroscope.

If you want to detect motions as gestures—specifically shaking motions—you should handle motion events as described in "Shaking-Motion Events" (page 56). If you want to receive and handle high-rate, continuous motion data, you should instead follow the approaches described in "Core Motion" (page 63) or "Accessing Accelerometer Events Using UIAccelerometer" (page 59).

> **Notes** This chapter contains information that used to be in *iOS App Programming Guide*. The information in this chapter has not been updated specifically for iOS 4.0.

Shaking-Motion Events

When users shake a device, the system evaluates the accelerometer data and, if that data meets certain criteria, interprets it as a shaking gesture. The system creates a `UIEvent` object representing this gesture and sends the event object to the currently active application for processing.

> **Note** Motion events as a type of `UIEvent` were introduced in iOS 3.0. Currently, only shaking motions are interpreted as gestures and become motion events.

Motion events are much simpler than touch events. The system tells an application when a motion starts and when it stops, and not when each individual motion occurs. And, whereas a touch event includes a set of touches and their related state, a motion event carries with it no state other than the event type, event subtype, and timestamp. The system interprets motion gestures in a way that does not conflict with orientation changes.

To receive motion events, the responder object that is to handle them must be the first responder. Listing 4-1 shows how a responder can make itself the first responder.

Listing 4-1 Becoming first responder

```
- (BOOL)canBecomeFirstResponder {
    return YES;
```

```
}

- (void)viewDidAppear:(BOOL)animated {
    [self becomeFirstResponder];
}
```

To handle motion events, a class inheriting from `UIResponder` must implement either the `motionBegan:withEvent:` method or `motionEnded:withEvent:` method, or possibly both of these methods (see "Best Practices for Handling Multitouch Events" (page 37)). For example, if an application wants to give horizontal shakes and vertical shakes different meanings, it can cache the current acceleration axis values in `motionBegan:withEvent:`, compare those cached values to the same axis values in `motionEnded:withEvent:`, and act on the results accordingly. A responder should also implement the `motionCancelled:withEvent:` method to respond to events that the system sends to cancel a motion event; these events sometimes reflect the system's determination that the motion is not a valid gesture after all.

Listing 4-2 shows code that handles a shaking-motion event by resetting views that have have been altered (by translation, rotation, and scaling) to their original positions, orientations, and sizes.

Listing 4-2 Handling a motion event

```
- (void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:0.5];
    self.view.transform = CGAffineTransformIdentity;

    for (UIView *subview in self.view.subviews) {
        subview.transform = CGAffineTransformIdentity;
    }
    [UIView commitAnimations];

    for (TransformGesture *gesture in [window allTransformGestures]) {
```

```
        [gesture resetTransform];
    }
}

- (void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}
```

An application and its key window deliver a motion event to a window's first responder for handling. If the first responder doesn't handle it, the event progresses up the responder chain in the same way touch events do until it is either handled or ignored. (See "Event Delivery" (page 11) for details.) However, there is one important difference between touch events and shaking-motion events. When the user starts shaking the device, the system sends a motion event to the first responder in a `motionBegan:withEvent:` message; if the first responder doesn't handle the event, it travels up the responder chain. If the shaking lasts less than a second or so, the system sends a `motionEnded:withEvent:` message to the first responder. But if the shaking lasts longer or if the system determines the motion is not a shake, the first responder receives a `motionCancelled:withEvent:` message.

If a shaking-motion event travels up the responder chain to the window without being handled and the `applicationSupportsShakeToEdit` property of `UIApplication` is set to `YES`, iOS displays a sheet with Undo and Redo commands. By default, this property is set to `YES`.

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 simple and does not require that you 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.

Setting Required Hardware Capabilities for Accelerometer and Gyroscope Events

If your application requires device-related features in order to run—such as the ability to receive accelerometer data—you must add a list of required capabilities to your application. At runtime, iOS launches your application only if those capabilities are present on the device. Furthermore, the App Store uses the information in this key to generate a list of requirements for user devices and prevent users from downloading applications that they cannot run.

You declare your application's required capabilities by adding the `UIRequiredDeviceCapabilities` key to your application's `Info.plist` file. This key, supported in iOS 3.0 and later, has a value that is either an array or a dictionary. If you use an array, the presence of a given key indicates the corresponding feature is required. If you use a dictionary, you must specify a Boolean value for each key indicating whether the feature is required. In both cases, having no key for a feature indicates that the feature is not required.

The following `UIRequiredDeviceCapabilities` keys are for motion events, based on hardware source:

- `accelerometer` (for accelerometer events)

  You do not need to include this key if your application detects only device orientation changes or if your application handles shaking-motion events delivered via `UIEvent` objects.

- `gyroscope` (for gyroscope events)

Accessing Accelerometer Events Using UIAccelerometer

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 (ms), 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 4-3 shows the basic steps for configuring an accelerometer. In this example, the update frequency is 50 Hz, which corresponds to an update interval of 20 ms. 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 4-3 Configuring the accelerometer

```
#define kAccelerometerFrequency        50.0 //Hz
-(void)configureAccelerometer
{
    UIAccelerometer*  theAccelerometer = [UIAccelerometer sharedAccelerometer];
    theAccelerometer.updateInterval = 1 / kAccelerometerFrequency;

    theAccelerometer.delegate = self;
    // Delegate events begin immediately.
}
```

At regular intervals, the shared accelerometer object delivers event data to your delegate's `accelerometer:didAccelerate:` method, shown in Listing 4-4. 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 4-4 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 delegate object to `nil` lets the system know that it can turn off the accelerometer hardware as needed and thus save battery life.

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 a 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 and still meets 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 4-1 lists some typical update frequencies and explains what you can do with the acceleration data generated at that frequency.

Table 4-1 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 caused by gravity from the portion of the data 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 then reflect the more constant effects of gravity.

Listing 4-5 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 4-5 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 4-6 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 4-6 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.
}
```

Core Motion

Core Motion is a system framework that obtains motion data from sensors on a device and presents that data to applications for processing. The handling of the sensor data and the application of related algorithms occurs on Core Motion's own thread. The items of hardware that detect and originate these motion events are the accelerometer and the gyroscope. (The gyroscope is currently available only on iPhone 4.) Core Motion publishes an Objective-C programmatic interface that enables applications to receive device-motion data of various types, which they can then process in appropriate ways.

As illustrated by Figure 4-1 (page 64), Core Motion defines a manager class, `CMMotionManager`, and three classes whose instances encapsulate measurements of motion data of various types:

- A `CMAccelerometerData` object encapsulates a data structure that records a measurement of device acceleration along the three spatial axes. This data derives from the accelerometer.

  For more on `CMAccelerometerData`, see "Handling Accelerometer Events Using Core Motion" (page 65).

- A `CMGyroData` object encapsulates a data structure that records a biased estimate of a device's rate of rotation along the three spatial axes. This "raw" data derives from the gyroscope. ("Biased" in this context refers to an offset from the true rotation rate. Thus, if the device is not rotating, this estimate from the gyroscope will still give a non-zero value.)

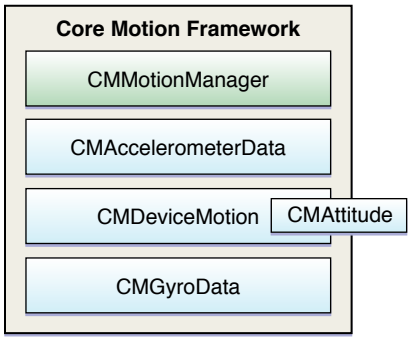
  For more on `CMGyroData`, see "Handling Rotation-Rate Data" (page 68).

- A `CMDeviceMotion` object encapsulates processed device-motion data that derives from both the accelerometer and the gyroscope. Core Motion's sensor fusion algorithms process both accelerometer and gyroscope data and provide an application with highly accurate measurements of device attitude, the (unbiased) rotation rate of a device, the direction of gravity on a device, and the acceleration that the

user is giving to a device. A `CMAttitude` object, which is contained in an `CMDeviceMotion` instance, contains properties giving different measurements of attitude, including the Euler angles indicated by roll, pitch, and yaw.

Attitude refers to the orientation of a device in three dimensions relative to a reference frame that is external to the device. For more on attitude and `CMDeviceMotion`, see "Handling Processed Device-Motion Data" (page 72).

Figure 4-1 Core Motion classes



All of the data-encapsulating classes of Core Motion are subclasses of `CMLogItem`, which defines a timestamp so that motion data can be tagged with the event time and logged to a file. An application can also compare the timestamp of motion events with earlier motion events to determine the true update interval between events.

For each of the data-motion types described above, the `CMMotionManager` class offers two approaches for obtaining motion data, a push approach and a pull approach:

- **Push.** An application requests an update interval and implements a block (of a specific type) for handling the motion data; it then starts updates for that type of motion data, passing into Core Motion an operation queue as well as the block. Core Motion delivers each update to the block, which executes as a task in the operation queue.
- **Pull.** An application starts updates of a type of motion data and periodically samples the most recent measurement of motion data.

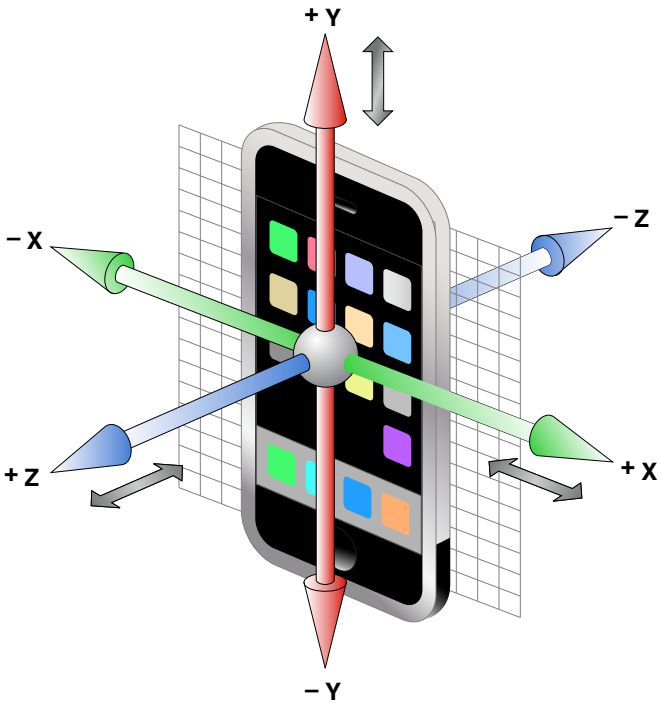
The pull approach is the recommended approach for most applications, especially games; it is generally more efficient and requires less code. The push approach is appropriate for data-collection applications and similar applications that cannot miss a sample measurement. Both approaches have benign thread-safety effects; with the push approach, your block executes on the operation-queue's thread whereas, with the pull approach, Core Motion never interrupts your threads.

Important An application should create only a single instance of the `CMMotionManager` class. Multiple instances of this class can affect the rate at which an application receives data from the accelerometer and gyroscope.

Be aware that there is no simulator support for application features related to Core Motion. You have to test and debug applications on a device.

Handling Accelerometer Events Using Core Motion

Core Motion provides an alternative programmatic interface to `UIAccelerometer` for accessing accelerometer events. Each of these events is an instance of `CMAccelerometerData` that encapsulates a measurement of accelerometer activity in a structure of type `CMAcceleration`.



To start receiving and handling accelerometer data, create an instance of the `CMMotionManager` class and call one of the following two methods on it:

- `startAccelerometerUpdates`

  After this method is called, Core Motion continuously updates the `accelerometerData` property of `CMMotionManager` with the latest measurement of accelerometer activity. The application periodically samples this property, usually in a render loop typical of games. If you adopt this polling approach, you should set the update-interval property (`accelerometerUpdateInterval`) to the maximum interval at which Core Motion performs updates. (Core Motion might perform updates at a faster rate, however.)

The code examples in this section illustrate this first approach.

- `startAccelerometerUpdatesToQueue:withHandler:`

  Before calling this method, the application assigns an update interval to the `accelerometerUpdateInterval` property. It also creates an instance of `NSOperationQueue` and implements a block of the `CMAccelerometerHandler` type that handles the accelerometer updates. Then it calls `startAccelerometerUpdatesToQueue:withHandler:` on the motion-manager object, passing in the operation queue and the block. At the specified update interval, Core Motion passes the latest sample of accelerometer activity to the block, which executes as a task in the queue.

You should stop updates of motion data as soon as your application is finished processing the data. Doing so allows Core Motion to turn off motion sensors, thereby saving battery power.

For Core Motion accelerometer events, you configure the update interval exactly as you do when using `UIAccelerometer`. You must identify an interval suitable for your application and then assign that value (expressed as seconds) to the `accelerometerUpdateInterval` property. If you prefer to think of the update interval in terms of cycles per second (Hertz), divide 1 by the desired Hertz value to get the update-interval value. Listing 4-3 (page 60) gives an example. ("Choosing an Appropriate Update Interval" (page 61) offers guidance for choosing a suitable update interval.)

The following code examples are based on the OpenGL ES project template in Xcode. An OpenGL ES application periodically samples device-motion updates using the render loop it sets up for drawing its view. The application first declares an instance variable—a three-member C array—to hold the acceleration values:

```
double filteredAcceleration[3];
```

As shown in Listing 4-7, the application creates an instance of `CMMotionManager` in the same template method used for configuring and scheduling the timing mechanism of the render loop (`startAnimation`). The application then assigns an appropriate accelerometer-update interval to the motion manager, allocates memory for the C array, and starts accelerometer updates. Note that the application stops accelerometer updates in the same template method (`stopAnimation`) used for invalidating the timing mechanism of the render loop.

Listing 4-7 Configuring the motion manager and starting updates

```
- (void)startAnimation {
    if (!animating) {
        // code that configures and schedules CADisplayLink or timer here ...
    }
```

```
    motionManager = [[CMMotionManager alloc] init]; // motionManager is an instance
 variable
    motionManager.accelerometerUpdateInterval = 0.01; // 100Hz
    memset(filteredAcceleration, 0, sizeof(filteredAcceleration));
    [motionManager startAccelerometerUpdates];
}

- (void)stopAnimation {
    if (animating) {
        // code that invalidates CADisplayLink or timer here...
    }
    [motionManager stopAccelerometerUpdates];
}
```

In the OpenGL ES application template, the `drawView` method is invoked at each cycle of the render loop. Listing 4-8 shows how the application, in this same method, gets the latest accelerometer data and runs it through a low-pass filter. It then updates the drawing model with the filtered acceleration values and renders its view.

Listing 4-8 Sampling and filtering accelerometer data

```
- (void)drawView {
    // alpha is the filter value (instance variable)
    CMAccelerometerData *newestAccel = motionManager.accelerometerData;
    filteredAcceleration[0] = filteredAcceleration[0] * (1.0-alpha) +
newestAccel.acceleration.x * alpha;
    filteredAcceleration[1] = filteredAcceleration[1] * (1.0-alpha) +
newestAccel.acceleration.y * alpha;
    filteredAcceleration[2] = filteredAcceleration[2] * (1.0-alpha) +
newestAccel.acceleration.z * alpha;
    [self updateModelsWithAcceleration:filteredAcceleration];
    [renderer render];
}
```

Note You can apply a low-pass or high-pass filter to acceleration values and thereby isolate the gravity and user-acceleration components:

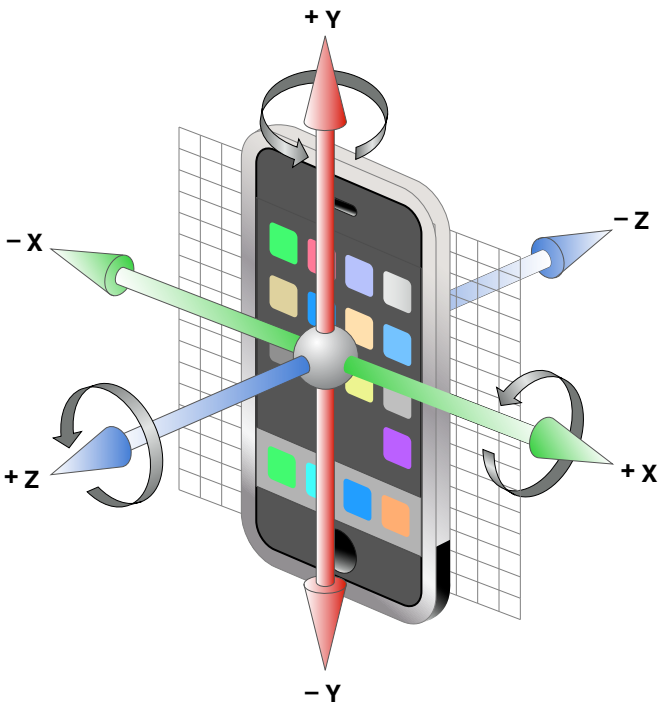
- To apply a low-pass filter, thereby isolating the gravity component, see "Isolating the Gravity Component from Acceleration Data" (page 61).
- To apply a high-pass filter, thereby isolating the user-acceleration component, see "Isolating Instantaneous Motion from Acceleration Data" (page 62) (which refers to the user-acceleration component as "instantaneous motion.")

Your application can also receive the gravity-caused and user-caused components of acceleration directly from Core Motion by receiving and handling device-motion updates instead of accelerometer updates. See "Handling Processed Device-Motion Data" (page 72) for information.

Handling Rotation-Rate Data

A gyroscope measures the rate at which a device rotates around each of the three spatial axes. (Compare this with the accelerometer, which measures the *acceleration* of the device along each of the three spatial axes.) For each requested gyroscope update, Core Motion takes a biased estimate of the rate of rotation and returns this information to an application in the form of a `CMGyroData` object. The object has a `rotationRate` property through which you can access a `CMRotationRate` structure that captures the rotation rate (in radians per second) for each of the three axes.

Note The measurement of rotation rate encapsulated by a `CMGyroData` object is biased. You can obtain a much more accurate (unbiased) measurement by accessing the `rotationRate` property of `CMDeviceMotion`.



To start receiving and handling rotation-rate data, create an instance of the `CMMotionManager` class and call one of the following two methods on it:

- `startGyroUpdates`

  After this method is called, Core Motion continuously updates the `gyroData` property of `CMMotionManager` with the latest measurement of gyroscope activity. The application periodically samples this property, usually in a render loop that is typical of games. If you adopt this polling approach, you should set the update-interval property (`gyroUpdateInterval`) to the maximum interval at which Core Motion performs updates. (Core Motion might perform updates at a faster rate, however.)

- `startGyroUpdatesToQueue:withHandler:`

  Before calling this method, the application assigns an update interval to the `gyroUpdateInterval` property. It also creates an instance of `NSOperationQueue` and implements a block of the `CMGyroHandler` type that handles the gyroscope updates. Then it calls `startGyroUpdatesToQueue:withHandler:` on the motion-manager object, passing in the operation queue and the block. At the specified update interval, Core Motion passes the latest sample of gyroscope activity to the block, which executes as a task in the queue.

The code examples in this section illustrate this approach.

You should stop updates of motion data as soon as your application is finished processing the data. Doing so allows Core Motion to turn off motion sensors, thereby saving battery power.

When configuring the update interval for rotation-rate (gyroscope) events, identify an interval suitable for your application and then assign that value (expressed as seconds) to the `gyroUpdateInterval` property. If you prefer to think of the update interval in terms of cycles per second (Hertz), divide 1 by the desired Hertz value to get the update-interval value. Listing 4-3 (page 60) in "Choosing an Appropriate Update Interval" (page 61) gives an example in the context of accelerometer updates.

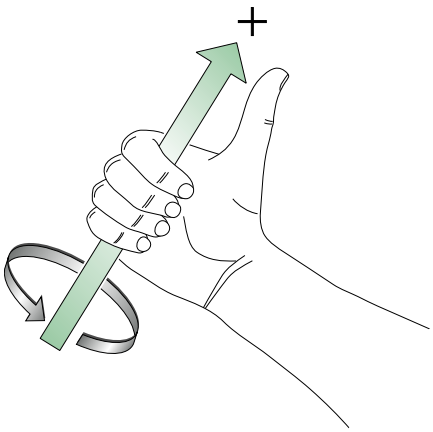
The following code segments illustrate how to start gyroscope updates by calling the `startGyroUpdatesToQueue:withHandler:` method. In Listing 4-9, a view controller in its `viewDidLoad` instantiates a `CMMotionManager` object and assigns an update interval for gyroscope data. If the device has a gyroscope, the view controller creates an `NSOperationQueue` object and defines a block handler for gyroscope updates.

Listing 4-9 Creating the `CMMotionManager` object and setting up for gyroscope updates

```
- (void)viewDidLoad {
    [super viewDidLoad];
    motionManager = [[CMMotionManager alloc] init];
    motionManager.gyroUpdateInterval = 1.0/60.0;
    if (motionManager.gyroAvailable) {
        opQ = [[NSOperationQueue currentQueue] retain];
        gyroHandler = ^ (CMGyroData *gyroData, NSError *error) {
            CMRotationRate rotate = gyroData.rotationRate;
            // handle rotation-rate data here......
        };
    } else {
        NSLog(@"No gyroscope on device.");
        toggleButton.enabled = NO;
        [motionManager release];
    }
}
```

When analyzing rotation-rate data—that is, the fields of the `CMRotationMatrix` structure—follow the "right-hand rule" to determine the direction of rotation (see Figure 4-2 (page 71)). For example, if you wrap your right hand around the X axis such that the tip of the thumb points toward positive X, a positive rotation is one toward the tips of the other 4 fingers. A negative rotation goes away from the tips of those fingers.

Figure 4-2 Right-hand rule



When the user taps a button, an action message is sent to the view controller. The view controller implements the action method to toggle between starting updates and stopping updates. Listing 4-10 shows how it does this.

Listing 4-10 Starting and stopping gyroscope updates

```
- (IBAction)toggleGyroUpdates:(id)sender {
    if ([[(UIButton *)sender currentTitle] isEqualToString:@"Start"]) {
        [motionManager startGyroUpdatesToQueue:opQ withHandler:gyroHandler];
    } else {
        [motionManager stopGyroUpdates];
    }

}
```

Handling Processed Device-Motion Data

If a device has an accelerometer and a gyroscope, Core Motion offers a device-motion service that reads raw motion data from both sensors. The service uses sensor fusion algorithms to refine the raw data and generate information on a device's attitude, its unbiased rotation rate, the direction of gravity on a device, and the acceleration that the user imparts to a device. An instance of the `CMDeviceMotion` class encapsulates this data.

You can access attitude data through the `attitude` property of a `CMDeviceMotion` object. An instance of the `CMAttitude` class encapsulates a measurement of attitude. This class defines three mathematical representations of attitude:

- a quaternion
- a rotation matrix
- the three Euler angles (roll, pitch, and yaw)

Because the device-motion service returns gravity and user acceleration as separate items of data, there is no need to filter the acceleration data.

To start receiving and handling device-motion updates, create an instance of the `CMMotionManager` class and call one of the following two methods on it:

- `startDeviceMotionUpdates`

  After this method is called, Core Motion continuously updates the `deviceMotion` property of `CMMotionManager` with the latest refined measurements of accelerometer and gyroscope activity (as encapsulated in a `CMDeviceMotion` object). The application periodically samples this property, usually in a render loop that is typical of games. If you adopt this polling approach, you should set the update-interval property (`deviceMotionUpdateInterval`) to the maximum interval at which Core Motion performs updates. (Core Motion might perform updates at a faster rate, however.)

  The code examples in this section illustrate this approach.

- `startDeviceMotionUpdatesToQueue:withHandler:`

  Before calling this method, the application assigns an update interval to the `deviceMotionUpdateInterval` property. It also creates an instance of `NSOperationQueue` and implements a block of the `CMDeviceMotionHandler` type that handles the accelerometer updates. Then it calls `startDeviceMotionUpdatesToQueue:withHandler:` on the motion-manager object, passing in the operation queue and the block. At the specified update interval, Core Motion passes the latest sample of combined accelerometer and gyroscope activity (as represented by a `CMDeviceMotion` object) to the block, which executes as a task in the queue.

You should stop updates of motion data as soon as your application is finished processing the data. Doing so allows Core Motion to turn off motion sensors, thereby saving battery power.

When configuring the update interval for device-motion events, identify an interval suitable for your application and then assign that value (expressed as seconds) to the `deviceMotionUpdateInterval` property. If you prefer to think of the update interval in terms of cycles per second (Hertz), divide 1 by the desired Hertz value to get the update-interval value. Listing 4-3 (page 60) in "Choosing an Appropriate Update Interval" (page 61) gives an example of this in the context of accelerometer updates.

An Example of Handling Device-Motion Data

The following code examples are based on the OpenGL ES project template in Xcode. An OpenGL ES application periodically samples device-motion updates using the render loop it sets up for drawing the view. In Listing 4-11, the application creates an instance of `CMMotionManager` in `initWithCoder:` and assigns this object to an instance variable. It also specifies a minimum update interval for device-motion data. The application then starts device-motion updates when the OpenGL view schedules the render loop; it stops device-motion updates when the view invalidates that loop.

Listing 4-11 Starting and stopping device-motion updates

```
- (id)initWithCoder:(NSCoder*)coder {
    if ((self = [super initWithCoder:coder])) {
        motionManager = [[CMMotionManager alloc] init];
        motionManager.deviceMotionUpdateInterval = 0.02; // 50 Hz
        // other initialization code here...
    }
}

- (void)startAnimation {
    if (!animating) {
        // code that configures and schedules CADisplayLink or timer here ...
    }
   if ([motionManager.isDeviceMotionAvailable])
       [motionManager startDeviceMotionUpdates];
}


- (void)stopAnimation {
    if (animating) {
```

```
        // code that invalidates CADisplayLink or timer here...
    }
    if ([motionManager.isDeviceMotionActive])
        [motionManager stopDeviceMotionUpdates];
}
```

Note that if device-motion services are not available—most likely because the device lacks a gyroscope—you might want to implement an alternative approach that responds to device motion by handling accelerometer data.

Device Attitude and the Reference Frame

A particularly useful bit of information yielded by a `CMDeviceMotion` object is device attitude. From a practical standpoint, an even more useful bit of information is the *change* in device attitude. The attitude, or spatial orientation of a device is always measured in relation to a reference frame. Core Motion establishes the reference frame when your application starts device-motion updates. An instance of `CMAttitude` gives the rotation from this initial reference frame to the device's current reference frame. Core Motion's reference frame is always chosen so that the z-axis is always vertical, and the x-axis and y-axis are always orthogonal to gravity. When expressed in Core Motion's reference frame, gravity is always the vector [0, 0, -1]; this is called the gravity reference. If you multiply the rotation matrix obtained from a `CMAttitude` object by the gravity reference, you get gravity in the device's frame. Or, mathematically:

$$\texttt{deviceMotion.gravity} = \mathsf{R} \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

You can change the reference frame used by a `CMAttitude` instance. To do that, cache the attitude object that contains that reference frame and pass that as the argument to `multiplyByInverseOfAttitude:`. The attitude argument receiving the message is changed to represent the change in attitude from that reference frame.

To see how this might be useful, consider a baseball game where the user rotates the device to swing. Normally, at the beginning of a pitch, the bat would be at some resting orientation. After that, the bat would be rendered at an orientation determined by how the device's attitude had changed from where it was at the start of a pitch. Listing 4-12 illustrates how you might do this.

Listing 4-12 Getting the change in attitude prior to rendering

```
-(void) startPitch {
```

```
    // referenceAttitude is an instance variable
    referenceAttitude = [motionManager.deviceMotion.attitude retain];
}

- (void)drawView {
    CMAttitude *currentAttitude = motionManager.deviceMotion.attitude;
    [currentAttitude multiplyByInverseOfAttitude: referenceAttitude];
    // render bat using currentAttitude .....
    [self updateModelsWithAttitude:currentAttitude];
    [renderer render];
}
```

After `multiplyByInverseOfAttitude:` returns, `currentAttitude` in this example represents the change in attitude (that is, the rotation) from `referenceAttitude` to the most recently sampled `CMAttitude` instance.

Remote Control of Multimedia

Remote-control events let users control application multimedia through the system transport controls or through an external accessory. If your application plays audio or video content as a feature, you might want to write the code that enables it to respond to remote-control events. These events originate either from the transport controls or as commands issued by external accessories (such as a headset) that conform to an Apple-provided specification. iOS converts these commands into `UIEvent` objects that it delivers to an application. The application sends them to the first responder and, if the first responder doesn't handle them, they go up the responder chain.

The following sections describe how to prepare your application for receiving remote-control events and how to handle them. The code examples are taken from the *Audio Mixer (MixerHost)* sample code project.

Preparing Your Application for Remote-Control Events

To receive remote-control events, the view or view controller managing the presentation of multimedia content must be the first responder. It (or some other object in the application) must also tell the application object that it is ready to receive remote-control events.

To make itself capable of becoming first responder, the view or view controller should override the `UIResponder` method `canBecomeFirstResponder` to return `YES`. It should also send itself the `becomeFirstResponder` at an appropriate time, such as (for view controllers) in an override of the `viewDidAppear:` method. Listing 5-1 shows this call and also shows something else: The view controller calls the `beginReceivingRemoteControlEvents` method of `UIApplication` to "turn on" the delivery of remote-control events.

Listing 5-1 Preparing to receive remote-control events

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
    [self becomeFirstResponder];
}
```

When the view or view controller is no longer managing audio or video, it should turn off the delivery of remote-control events and resign first-responder status by implementing the `viewWillDisappear:` as shown in Listing 5-2.

Listing 5-2 Ending the receipt of remote-control events

```
- (void)viewWillDisappear:(BOOL)animated {
    [[UIApplication sharedApplication] endReceivingRemoteControlEvents];
    [self resignFirstResponder];
    [super viewWillDisappear:animated];
}
```

Handling Remote-Control Events

To handle remote-control events, the first responder must implement the `remoteControlReceivedWithEvent:` method declared by `UIResponder`. The method implementation should evaluate the `subtype` of each `UIEvent` object passed in and send the appropriate message to the object presenting the audio or video content. The example in Listing 5-3 sends play, pause, and stop messages to an audio object.

Listing 5-3 Handling remote-control events

```
- (void) remoteControlReceivedWithEvent: (UIEvent *) receivedEvent {

    if (receivedEvent.type == UIEventTypeRemoteControl) {

        switch (receivedEvent.subtype) {

            case UIEventSubtypeRemoteControlTogglePlayPause:
                [self playOrStop: nil];
                break;

            case UIEventSubtypeRemoteControlPreviousTrack:
                [self previousTrack: nil];
                break;

            case UIEventSubtypeRemoteControlNextTrack:
```

```
                [self nextTrack: nil];
                break;

            default:
                break;
        }
    }
}
```

Other remote-control `UIEvent` subtypes are possible. See *UIEvent Class Reference* for details.

You can test your application's receipt and handling of remote-control events by using the Now Playing Controls. These controls are available on recent models of device (for iPhone, iPhone 3GS and later) that are running iOS 4.0 or later. To access these controls, double-press the Home button, then flick left or right along the bottom of the screen until you find the audio playback controls. These controls send remote-control events to the application that is currently or was most recently playing audio; the icon to the right of the playback controls represents the application.

For testing purposes, you can programmatically make your application begin audio playback and then test the remote-control events sent to your application by tapping the Now Playing Controls. Note that a deployed application should not programmatically begin playback; that should always be done by the user.

Document Revision History

This table describes the changes to *Event Handling Guide for iOS* .

Date	Notes
2011-03-10	Made some minor corrections.
2010-09-29	Made some minor corrections and clarifications.
2010-08-12	Corrected code snippet in "Remote Control of Multimedia"
2010-08-03	Corrected code examples and related text in "Remote Control of Multimedia" chapter. Made other minor corrections.
2010-07-09	Changed the title from "Event Handling Guide for iPhone OS" and changed "iPhone OS" to "iOS" throughout. Updated the section on the Core Motion framework.
2010-05-18	First version of a document that describes how applications can handle multitouch, motion, and other events.

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