Event Handling Guide for iOS

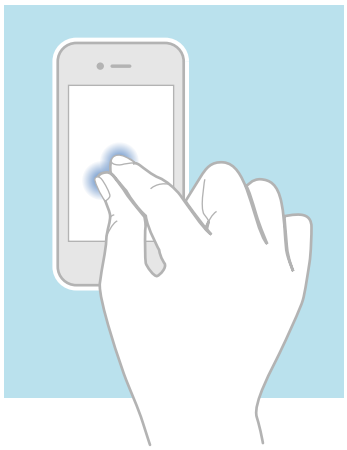
Developer

Contents

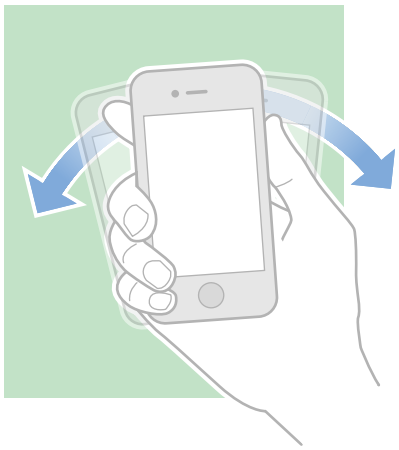
Figures, Tables, and Listings

About Events in iOS

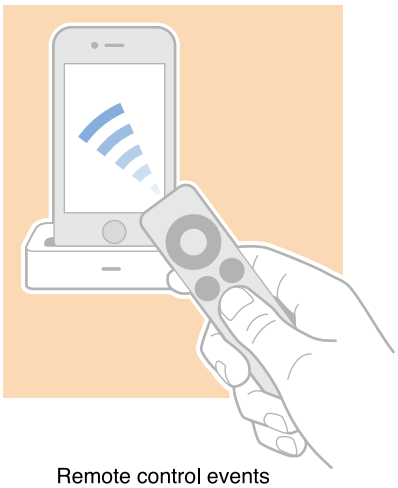
Users manipulate their iOS devices in a number of ways, such as touching the screen or shaking the device. iOS interprets when and how a user is manipulating the hardware and passes this information to your app. The more your app responds to actions in natural and intuitive ways, the more compelling the experience is for the user.



Multitouch events



Accelerometer events



Remote control events

At a Glance

Events are objects sent to an app to inform it of user actions. In iOS, events can take many forms: Multi-Touch events, motion events, and events for controlling multimedia. This last type of event is known as a *remote control event* because it can originate from an external accessory.

UIKit Makes It Easy for Your App to Detect Gestures

iOS apps recognize combinations of touches 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. In fact, some gestures are so common that they are built in to UIKit. For example, `UIControl` subclasses, such as `UIButton` and `UISlider`, respond to specific gestures—a tap for a button and a drag for a slider. When you configure these controls, they send an action message to a target object when that touch occurs. You can also employ the target-action mechanism on views by using gesture recognizers. When you attach a gesture recognizer to a view, the entire view acts like a control—responding to whatever gesture you specify.

Gesture recognizers provide a higher-level abstraction for complex event handling logic. Gesture recognizers are the preferred way to implement touch-event handling in your app because gesture recognizers are powerful, reusable, and adaptable. You can use one of the built-in gesture recognizers and customize its behavior. Or you can create your own gesture recognizer to recognize a new gesture.

Relevant Chapter: "Gesture Recognizers" (page 10)

An Event Travels Along a Specific Path Looking for an Object to Handle It

When iOS recognizes an event, it passes the event to the initial object that seems most relevant for handling that event, such as the view where a touch occurred. If the initial object cannot handle the event, iOS continues to pass the event to objects with greater scope until it finds an object with enough context to handle the event. This sequence of objects is known as a *responder chain* , and as iOS passes events along the chain, it also transfers the responsibility of responding to the event. This design pattern makes event handling cooperative and dynamic.

Relevant Chapter: "Event Delivery: The Responder Chain" (page 31)

A UIEvent Encapsulates a Touch, Shake-Motion, or Remote-Control Event

Many events are instances of the UIKit `UIEvent` class. A `UIEvent` object contains information about the event that your app uses to decide how to respond to the event. As a user action occurs—for example, as fingers touch the screen and move across its surface—iOS continually sends event objects to an app for handling. Each event object has a type—touch, "shaking" motion, or remote control—and a subtype.

Relevant Chapters: "Multitouch Events" (page 37), "Motion Events" (page 55), and "Remote Control Events" (page 69)

An App Receives Multitouch Events When Users Touch Its Views

Depending on your app, UIKit controls and gesture recognizers might be sufficient for all of your app's touch event handling. Even if your app has custom views, you can use gesture recognizers. As a rule of thumb, you write your own custom touch-event handling when your app's response to touch is tightly coupled with the view itself, such as drawing under a touch. In these cases, you are responsible for the low-level event handling. You implement the touch methods, and within these methods, you analyze raw touch events and respond appropriately.

Relevant Chapter: "Multitouch Events" (page 37)

An App Receives Motion Events When Users Move Their Devices

Motion events provide information about the device's location, orientation, and movement. By reacting to motion events, you can add subtle, yet powerful features to your app. Accelerometer and gyroscope data allow you to detect tilting, rotating, and shaking.

Motion events come in different forms, and you can handle them using different frameworks. When users shake the device, UIKit delivers a `UIEvent` object to an app. If you want your app to receive high-rate, continuous accelerometer and gyroscope data, use the Core Motion framework.

Relevant Chapter: "Motion Events" (page 55)

An App Receives Remote Control Events When Users Manipulate Multimedia Controls

iOS controls and external accessories send remote control events to an app. These events allow users to control audio and video, such as adjusting the volume through a headset. Handle multimedia remote control events to make your app responsive to these types of commands.

Relevant Chapter: "Remote Control Events" (page 69)

Prerequisites

This document assumes that you are familiar with:

- The basic concepts of iOS app development
- The different aspects of creating your app's user interface
- How views and view controllers work, and how to customize them

If you are not familiar with those concepts, start by reading *Start Developing iOS Apps Today*. Then, be sure to read either *View Programming Guide for iOS* or *View Controller Programming Guide for iOS*, or both.

See Also

In the same way that iOS devices provide touch and device motion data, most iOS devices have GPS and compass hardware that generates low-level data that your app might be interested in. *Location Awareness Programming Guide* discusses how to receive and handle location data.

For advanced gesture recognizer techniques such as curve smoothing and applying a low-pass filter, see *WWDC 2012: Building Advanced Gesture Recognizers* .

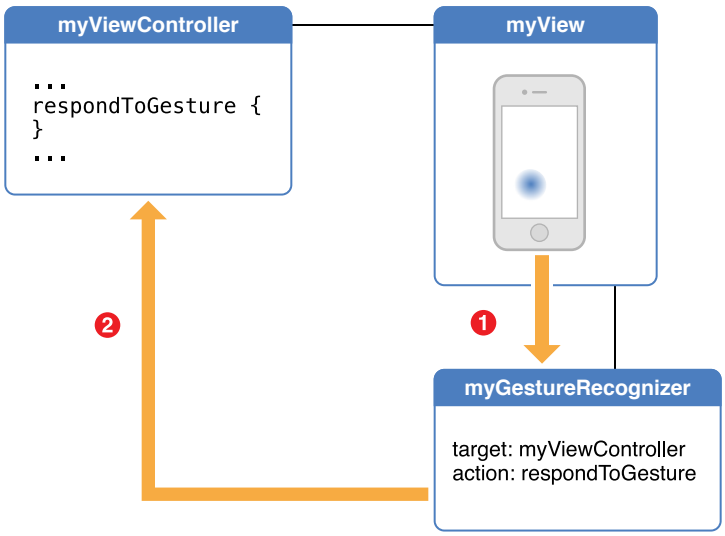
Many sample code projects in the iOS Reference Library have code that uses gesture recognizers and handles events. Among these are the following projects:

- *Simple Gesture Recognizers* is a perfect starting point for understanding gesture recognition. This app demonstrates how to recognize tap, swipe, and rotate gestures. The app responds to each gesture by displaying and animating an image at the touch location.
- *Touches* includes two projects that demonstrate how to handle multiple touches to drag squares around onscreen. One version uses gesture recognizers, and the other uses custom touch-event handling methods. The latter version is especially useful for understanding touch phases because it displays the current touch phase onscreen as the touches occur.
- *MoveMe* shows how to animate a view in response to touch events. Examine this sample project to further your understanding of custom touch-event handling.

Gesture Recognizers

Gesture recognizers convert low-level event handling code into higher-level actions. They are objects that you attach to a view, which allows the view to respond to actions the way a control does. Gesture recognizers interpret touches to determine whether they correspond to a specific gesture, such as a swipe, pinch, or rotation. If they recognize their assigned gesture, they send an action message to a target object. The target object is typically the view's view controller, which responds to the gesture as shown in Figure 1-1. This design pattern is both powerful and simple; you can dynamically determine what actions a view responds to, and you can add gesture recognizers to a view without having to subclass the view.

Figure 1-1 A gesture recognizer attached to a view



Use Gesture Recognizers to Simplify Event Handling

The UIKit framework provides predefined gesture recognizers that detect common gestures. It's best to use a predefined gesture recognizer when possible because their simplicity reduces the amount of code you have to write. In addition, using a standard gesture recognizer instead of writing your own ensures that your app behaves the way users expect.

If you want your app to recognize a unique gesture, such as a checkmark or a swirly motion, you can create your own custom gesture recognizer. To learn how to design and implement your own gesture recognizer, see "Creating a Custom Gesture Recognizer" (page 27).

Built-in Gesture Recognizers Recognize Common Gestures

When designing your app, consider what gestures you want to enable. Then, for each gesture, determine whether one of the predefined gesture recognizers listed in Table 1-1 is sufficient.

Table 1-1 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`

Your app should respond to gestures only in ways that users expect. For example, a pinch should zoom in and out whereas a tap should select something. For guidelines about how to properly use gestures, see "Apps Respond to Gestures, Not Clicks" in *iOS Human Interface Guidelines*.

Gesture Recognizers Are Attached to a View

Every gesture recognizer is associated with one view. By contrast, a view can have multiple gesture recognizers, because a single view might respond to many different gestures. For a gesture recognizer to recognize touches that occur in a particular view, you must attach the gesture recognizer to that view. When a user touches that view, the gesture recognizer receives a message that a touch occurred before the view object does. As a result, the gesture recognizer can respond to touches on behalf of the view.

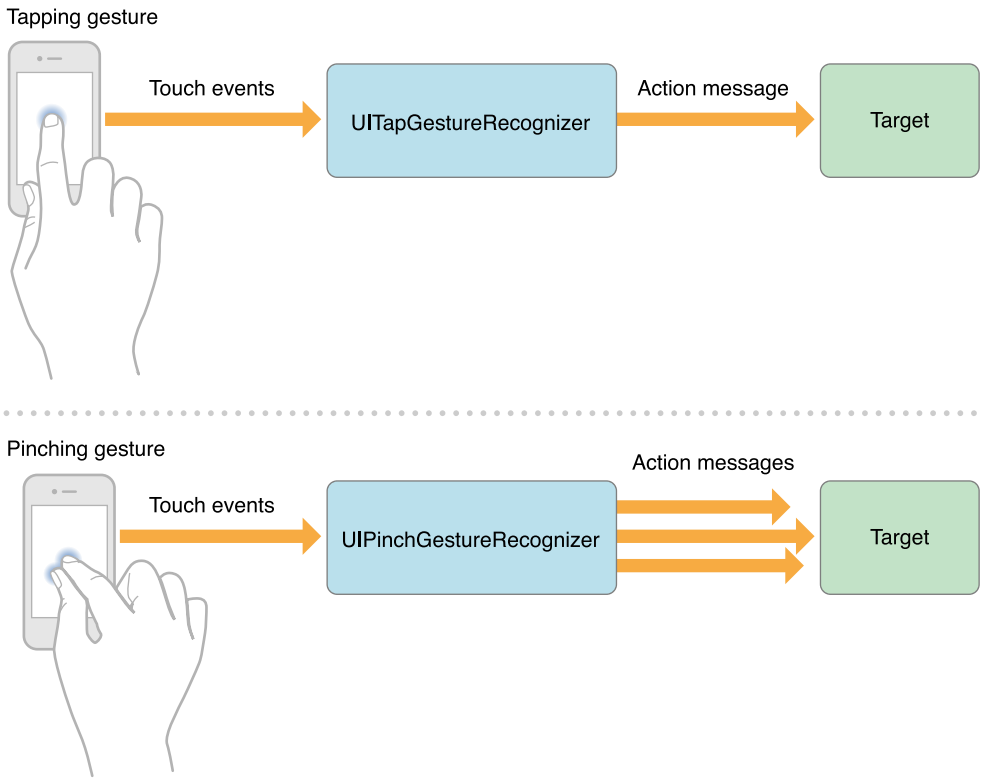
Gestures Trigger Action Messages

When a gesture recognizer recognizes its specified gesture, it sends an action message to its target. To create a gesture recognizer, you initialize it with a target and an action.

Discrete and Continuous Gestures

Gestures are either discrete or continuous. A *discrete gesture*, such as a tap, occurs once. A *continuous gesture*, such as pinching, takes place over a period of time. For discrete gestures, a gesture recognizer sends its target a single action message. A gesture recognizer for continuous gestures keeps sending action messages to its target until the multitouch sequence ends, as shown in Figure 1-2.

Figure 1-2 Discrete and continuous gestures



Responding to Events with Gesture Recognizers

There are three things you do to add a built-in gesture recognizer to your app:

1. Create and configure a gesture recognizer instance.

   This step includes assigning a target, action, and sometimes assigning gesture-specific attributes (such as the `numberOfTapsRequired`).

2. Attach the gesture recognizer to a view.

3. Implement the action method that handles the gesture.

Using Interface Builder to Add a Gesture Recognizer to Your App

Within Interface Builder in Xcode, add a gesture recognizer to your app the same way you add any object to your user interface—drag the gesture recognizer from the object library to a view. When you do this, the gesture recognizer automatically becomes attached to that view. You can check which view your gesture recognizer is attached to, and if necessary, change the connection in the nib file.

After you create the gesture recognizer object, you need to create and connect an action method. This method is called whenever the connected gesture recognizer recognizes its gesture. If you need to reference the gesture recognizer outside of this action method, you should also create and connect an outlet for the gesture recognizer. Your code should look similar to Listing 1-1.

Listing 1-1 Adding a gesture recognizer to your app with Interface Builder

```
@interface APLGestureRecognizerViewController ()
@property (nonatomic, strong) IBOutlet UITapGestureRecognizer *tapRecognizer;
@end

@implementation
- (IBAction)displayGestureForTapRecognizer:(UITapGestureRecognizer *)recognizer
    // Will implement method later...
}
@end
```

Adding a Gesture Recognizer Programmatically

You can create a gesture recognizer programmatically by allocating and initializing an instance of a concrete `UIGestureRecognizer` subclass, such as `UIPinchGestureRecognizer`. When you initialize the gesture recognizer, specify a target object and an action selector, as in Listing 1-2. Often, the target object is the view's view controller.

If you create a gesture recognizer programmatically, you need to attach it to a view using the `addGestureRecognizer:` method. Listing 1-2 creates a single tap gesture recognizer, specifies that one tap is required for the gesture to be recognized, and then attaches the gesture recognizer object to a view. Typically, you create a gesture recognizer in your view controller's `viewDidLoad` method, as shown in Listing 1-2.

Listing 1-2 Creating a single tap gesture recognizer programmatically

```
- (void)viewDidLoad {
    [super viewDidLoad];
```

```
    // Create and initialize a tap gesture
    UITapGestureRecognizer *tapRecognizer = [[UITapGestureRecognizer alloc]
         initWithTarget:self action:@selector(respondToTapGesture:)];

    // Specify that the gesture must be a single tap
    tapRecognizer.numberOfTapsRequired = 1;

    // Add the tap gesture recognizer to the view
    [self.view addGestureRecognizer:tapRecognizer];

    // Do any additional setup after loading the view, typically from a nib
}
```

Responding to Discrete Gestures

When you create a gesture recognizer, you connect the recognizer to an action method. Use this action method to respond to your gesture recognizer's gesture. Listing 1-3 provides an example of responding to a discrete gesture. When the user taps the view that the gesture recognizer is attached to, the view controller displays an image view that says "Tap." The `showGestureForTapRecognizer:` method determines the location of the gesture in the view from the recognizer's `locationInView:` property and then displays the image at that location.

> **Note:** The next three code examples are from the *Simple Gesture Recognizers* sample code project, which you can examine for more context.

Listing 1-3 Handling a double tap gesture

```
- (IBAction)showGestureForTapRecognizer:(UITapGestureRecognizer *)recognizer {
       // Get the location of the gesture
      CGPoint location = [recognizer locationInView:self.view];

       // Display an image view at that location
      [self drawImageForGestureRecognizer:recognizer atPoint:location];

       // Animate the image view so that it fades out
```

```
      [UIView animateWithDuration:0.5 animations:^{
           self.imageView.alpha = 0.0;
      }];
}
```

Each gesture recognizer has its own set of properties. For example, in Listing 1-4, the `showGestureForSwipeRecognizer:` method uses the swipe gesture recognizer's `direction` property to determine if the user swiped to the left or to the right. Then, it uses that value to make an image fade out in the same direction as the swipe.

Listing 1-4 Responding to a left or right swipe gesture

```
// Respond to a swipe gesture
- (IBAction)showGestureForSwipeRecognizer:(UISwipeGestureRecognizer *)recognizer
{
       // Get the location of the gesture
       CGPoint location = [recognizer locationInView:self.view];

       // Display an image view at that location
       [self drawImageForGestureRecognizer:recognizer atPoint:location];

       // If gesture is a left swipe, specify an end location
       // to the left of the current location
       if (recognizer.direction == UISwipeGestureRecognizerDirectionLeft) {
            location.x -= 220.0;
       } else {
            location.x += 220.0;
       }

       // Animate the image view in the direction of the swipe as it fades out
       [UIView animateWithDuration:0.5 animations:^{
            self.imageView.alpha = 0.0;
            self.imageView.center = location;
       }];
}
```

Responding to Continuous Gestures

Continuous gestures allow your app to respond to a gesture as it is happening. For example, your app can zoom while a user is pinching or allow a user to drag an object around the screen.

Listing 1-5 displays a "Rotate" image at the same rotation angle as the gesture, and when the user stops rotating, animates the image so it fades out in place while rotating back to horizontal. As the user rotates his fingers, the `showGestureForRotationRecognizer:` method is called continually until both fingers are lifted.

Listing 1-5 Responding to a rotation gesture

```
// Respond to a rotation gesture
- (IBAction)showGestureForRotationRecognizer:(UIRotationGestureRecognizer
*)recognizer {
       // Get the location of the gesture
       CGPoint location = [recognizer locationInView:self.view];

       // Set the rotation angle of the image view to
       // match the rotation of the gesture
       CGAffineTransform transform = CGAffineTransformMakeRotation([recognizer
rotation]);
       self.imageView.transform = transform;

       // Display an image view at that location
       [self drawImageForGestureRecognizer:recognizer atPoint:location];

      // If the gesture has ended or is canceled, begin the animation
      // back to horizontal and fade out
      if (([recognizer state] == UIGestureRecognizerStateEnded) || ([recognizer
state] == UIGestureRecognizerStateCancelled)) {
           [UIView animateWithDuration:0.5 animations:^{
                self.imageView.alpha = 0.0;
                self.imageView.transform = CGAffineTransformIdentity;
           }];
      }

}
```

Each time the method is called, the image is set to be opaque in the `drawImageForGestureRecognizer:` method. When the gesture is complete, the image is set to be transparent in the `animateWithDuration:` method. The `showGestureForRotationRecognizer:` method determines whether a gesture is complete by checking the gesture recognizer's state. These states are explained in more detail in "Gesture Recognizers Operate in a Finite State Machine" (page 17).

Defining How Gesture Recognizers Interact

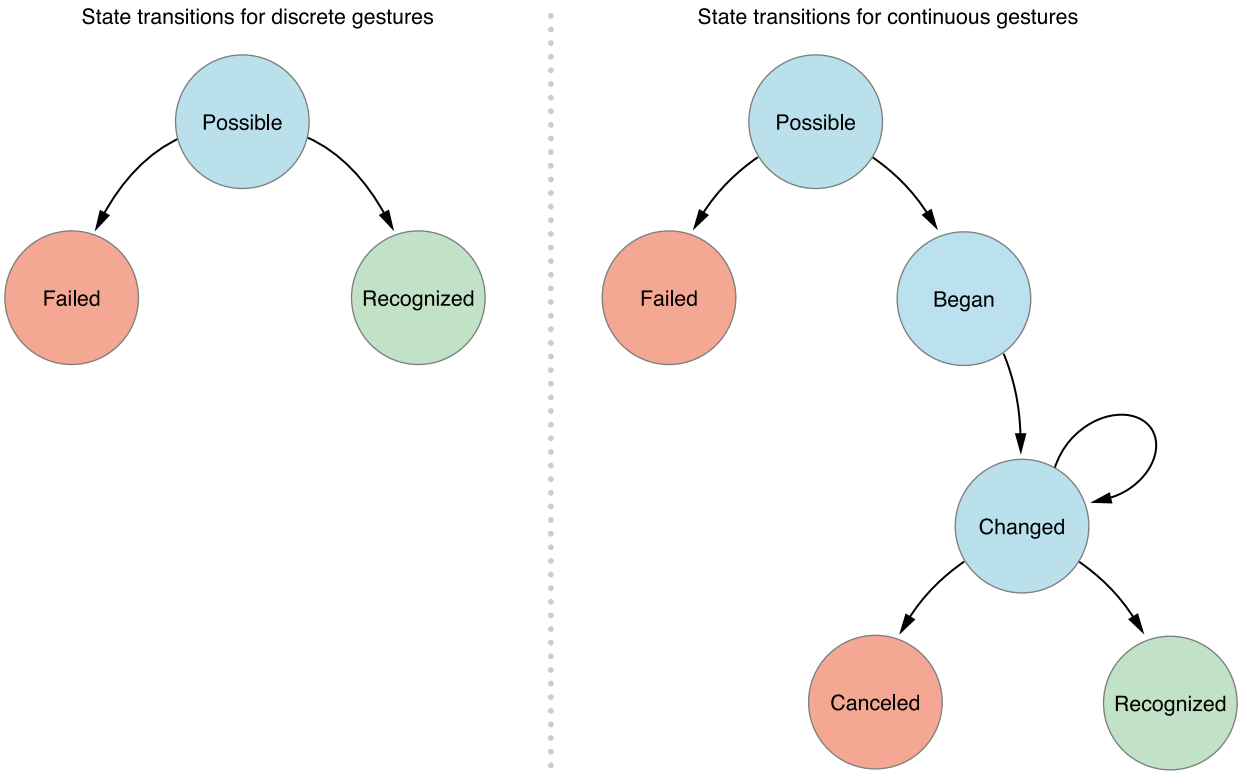
Oftentimes, as you add gesture recognizers to your app, you need to be specific about how you want the recognizers to interact with each other or any other touch-event handling code in your app. To do this, you first need to understand a little more about how gesture recognizers work.

Gesture Recognizers Operate in a Finite State Machine

Gesture recognizers transition from one state to another in a predefined way. From each state, they can move to one of several possible next states based on whether they meet certain conditions. The exact state machine varies depending on whether the gesture recognizer is discrete or continuous, as illustrated in Figure 1-3. All gesture recognizers start in the Possible state (`UIGestureRecognizerStatePossible`). They analyze any

multitouch sequences that they receive, and during analysis they either recognize or fail to recognize a gesture. Failing to recognize a gesture means the gesture recognizer transitions to the Failed state (`UIGestureRecognizerStateFailed`).

Figure 1-3 State machines for gesture recognizers



When a discrete gesture recognizer recognizes its gesture, the gesture recognizer transitions from Possible to Recognized (`UIGestureRecognizerStateRecognized`) and the recognition is complete.

For continuous gestures, the gesture recognizer transitions from Possible to Began (`UIGestureRecognizerStateBegan`) when the gesture is first recognized. Then, it transitions from Began to Changed (`UIGestureRecognizerStateChanged`), and continues to move from Changed to Changed as the gesture occurs. When the user's last finger is lifted from the view, the gesture recognizer transitions to the Ended state (`UIGestureRecognizerStateEnded`) and the recognition is complete. Note that the Ended state is an alias for the Recognized state.

A recognizer for a continuous gesture can also transition from Changed to Canceled (`UIGestureRecognizerStateCancelled`) if it decides that the gesture no longer fits the expected pattern.

Every time a gesture recognizer changes state, the gesture recognizer sends an action message to its target, unless it transitions to Failed or Canceled. Thus, a discrete gesture recognizer sends only a single action message when it transitions from Possible to Recognized. A continuous gesture recognizer sends many action messages as it changes states.

When a gesture recognizer reaches the Recognized (or Ended) state, it resets its state back to Possible. The transition back to Possible does not trigger an action message.

Interacting with Other Gesture Recognizers

A view can have more than one gesture recognizer attached to it. Use the view's `gestureRecognizers` property to determine what gesture recognizers are attached to a view. You can also dynamically change how a view handles gestures by adding or removing a gesture recognizer from a view with the `addGestureRecognizer:` and `removeGestureRecognizer:` methods, respectively.

When a view has multiple gesture recognizers attached to it, you may want to alter how the competing gesture recognizers receive and analyze touch events. By default, there is no set order for which gesture recognizers receive a touch first, and for this reason touches can be passed to gesture recognizers in a different order each time. You can override this default behavior to:

- Specify that one gesture recognizer should analyze a touch before another gesture recognizer.
- Allow two gesture recognizers to operate simultaneously.
- Prevent a gesture recognizer from analyzing a touch.

Use the `UIGestureRecognizer` class methods, delegate methods, and methods overridden by subclasses to effect these behaviors.

Declaring a Specific Order for Two Gesture Recognizers

Imagine that you want to recognize a swipe and a pan gesture, and you want these two gestures to trigger distinct actions. By default, when the user attempts to swipe, the gesture is interpreted as a pan. This is because a swiping gesture meets the necessary conditions to be interpreted as a pan (a continuous gesture) before it meets the necessary conditions to be interpreted as a swipe (a discrete gesture).

For your view to recognize both swipes and pans, you want the swipe gesture recognizer to analyze the touch event before the pan gesture recognizer does. If the swipe gesture recognizer determines that a touch is a swipe, the pan gesture recognizer never needs to analyze the touch. If the swipe gesture recognizer determines that the touch is not a swipe, it moves to the Failed state and the pan gesture recognizer should begin analyzing the touch event.

You indicate this type of relationship between two gesture recognizers by calling the `requireGestureRecognizerToFail:` method on the gesture recognizer that you want to delay, as in Listing 1-6. In this listing, both gesture recognizers are attached to the same view.

Listing 1-6 Pan gesture recognizer requires a swipe gesture recognizer to fail

```
- (void)viewDidLoad {
        [super viewDidLoad];
        // Do any additional setup after loading the view, typically from a nib
        [self.panRecognizer requireGestureRecognizerToFail:self.swipeRecognizer];
}
```

The `requireGestureRecognizerToFail:` method sends a message to the receiver and specifies some *otherGestureRecognizer* that must fail before the receiving recognizer can begin. While it's waiting for the other gesture recognizer to transition to the Failed state, the receiving recognizer stays in the Possible state. If the other gesture recognizer fails, the receiving recognizer analyzes the touch event and moves to its next state. On the other hand, if the other gesture recognizer transitions to Recognized or Began, the receiving recognizer moves to the Failed state. For information about state transitions, see "Gesture Recognizers Operate in a Finite State Machine" (page 17).

> **Note:** If your app recognizes both single and double taps and your single tap gesture recognizer does not require the double tap recognizer to fail, then you should expect to receive single tap actions before double tap actions, even when the user double taps. This behavior is intentional because the best user experience generally enables multiple types of actions.
>
> If you want these two actions to be mutually exclusive, your single tap recognizer must require the double tap recognizer to fail. However, your single tap actions will lag a little behind the user's input because the single tap recognizer is delayed until the double tap recognizer fails.

Preventing Gesture Recognizers from Analyzing Touches

You can alter the behavior of a gesture recognizer by adding a delegate object to your gesture recognizer. The `UIGestureRecognizerDelegate` protocol provides a couple of ways that you can prevent a gesture recognizer from analyzing touches. You use either the `gestureRecognizer:shouldReceiveTouch:` method or the `gestureRecognizerShouldBegin:` method—both are optional methods of the `UIGestureRecognizerDelegate` protocol.

When a touch begins, if you can immediately determine whether or not your gesture recognizer should consider that touch, use the `gestureRecognizer:shouldReceiveTouch:` method. This method is called every time there is a new touch. Returning `NO` prevents the gesture recognizer from being notified that a touch occurred. The default value is `YES`. This method does not alter the state of the gesture recognizer.

Listing 1-7 uses the `gestureRecognizer:shouldReceiveTouch:` delegate method to prevent a tap gesture recognizer from receiving touches that are within a custom subview. When a touch occurs, the `gestureRecognizer:shouldReceiveTouch:` method is called. It determines whether the user touched the custom view, and if so, prevents the tap gesture recognizer from receiving the touch event.

Listing 1-7 Preventing a gesture recognizer from receiving a touch

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Add the delegate to the tap gesture recognizer
    self.tapGestureRecognizer.delegate = self;
}

// Implement the UIGestureRecognizerDelegate method
-(BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
shouldReceiveTouch:(UITouch *)touch {
    // Determine if the touch is inside the custom subview
    if ([touch view] == self.customSubview)){
        // If it is, prevent all of the delegate's gesture recognizers
        // from receiving the touch
        return NO;
    }
    return YES;
}
```

If you need to wait as long as possible before deciding whether or not a gesture recognizer should analyze a touch, use the `gestureRecognizerShouldBegin:` delegate method. Generally, you use this method if you have a `UIView` or `UIControl` subclass with custom touch-event handling that competes with a gesture recognizer. Returning `NO` causes the gesture recognizer to immediately fail, which allows the other touch handling to proceed. This method is called when a gesture recognizer attempts to transition out of the Possible state, if the gesture recognition would prevent a view or control from receiving a touch.

You can use the `gestureRecognizerShouldBegin:UIView` method if your view or view controller cannot be the gesture recognizer's delegate. The method signature and implementation is the same.

Permitting Simultaneous Gesture Recognition

By default, two gesture recognizers cannot recognize their respective gestures at the same time. But suppose, for example, that you want the user to be able to pinch and rotate a view at the same time. You need to change the default behavior by implementing the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` method, an optional method of the `UIGestureRecognizerDelegate`protocol. This method is called when one gesture recognizer's analysis of a gesture would block another gesture recognizer from recognizing its gesture, or vice versa. This method returns `NO` by default. Return `YES` when you want two gesture recognizers to analyze their gestures simultaneously.

> **Note:** You need to implement a delegate and return YES on only one of your gesture recognizers to allow simultaneous recognition. However, that also means that returning NO doesn't necessarily prevent simultaneous recognition because the other gesture recognizer's delegate could return YES.

Specifying a One-Way Relationship Between Two Gesture Recognizers

If you want to control how two recognizers interact with each other but you need to specify a one-way relationship, you can override either the `canPreventGestureRecognizer:` or `canBePreventedByGestureRecognizer:` subclass methods to return `NO` (default is `YES`). For example, if you want a rotation gesture to prevent a pinch gesture but you don't want a pinch gesture to prevent a rotation gesture, you would specify:

```
[rotationGestureRecognizer canPreventGestureRecognizer:pinchGestureRecognizer];
```

and override the rotation gesture recognizer's subclass method to return `NO`. For more information about how to subclass `UIGestureRecognizer`, see "Creating a Custom Gesture Recognizer" (page 27).

If neither gesture should prevent the other, use the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` method, as described in "Permitting Simultaneous Gesture Recognition" (page 22). By default, a pinch gesture prevents a rotation and vice versa because two gestures cannot be recognized at the same time.

Interacting with Other User Interface Controls

In iOS 6.0 and later, default control actions prevent overlapping gesture recognizer behavior. For example, the default action for a button is a single tap. If you have a single tap gesture recognizer attached to a button's parent view, and the user taps the button, then the button's action method receives the touch event instead of the gesture recognizer. This applies only to gesture recognition that overlaps the default action for a control, which includes:

- A single finger single tap on a `UIButton`, `UISwitch`, `UIStepper`, `UISegmentedControl`, and `UIPageControl`.
- A single finger swipe on the knob of a `UISlider`, in a direction parallel to the slider.
- A single finger pan gesture on the knob of a `UISwitch`, in a direction parallel to the switch.

If you have a custom subclass of one of these controls and you want to change the default action, attach a gesture recognizer directly to the control instead of to the parent view. Then, the gesture recognizer receives the touch event first. As always, be sure to read the *iOS Human Interface Guidelines* to ensure that your app offers an intuitive user experience, especially when overriding the default behavior of a standard control.

Gesture Recognizers Interpret Raw Touch Events

So far, you've learned about gestures and how your app can recognize and respond to them. However, to create a custom gesture recognizer or to control how gesture recognizers interact with a view's touch-event handling, you need to think more specifically in terms of touches and events.

An Event Contains All the Touches for the Current Multitouch Sequence

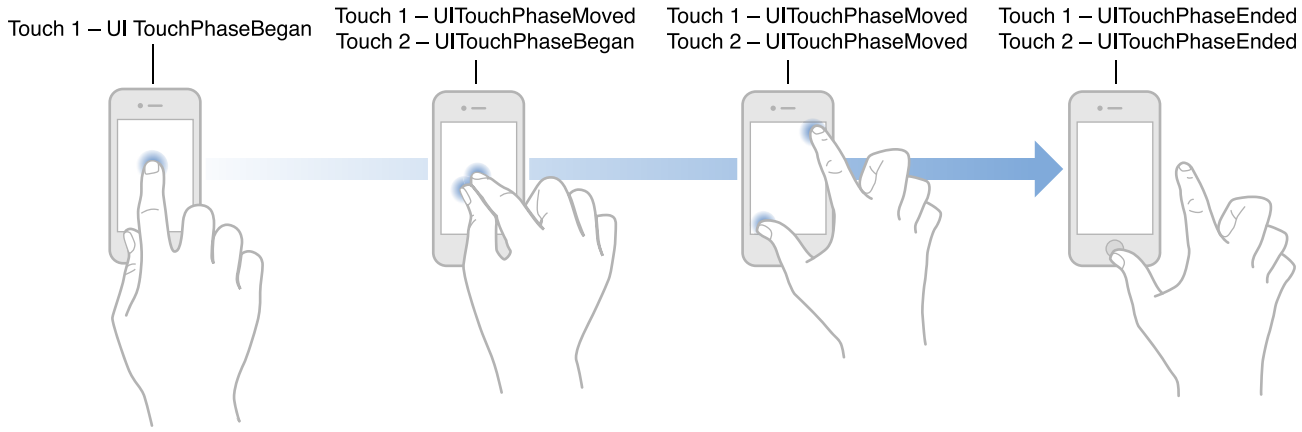
In iOS, a **touch** is the presence or movement of a finger on the screen. A gesture has one or more touches, which are represented by `UITouch` objects. For example, a pinch-close gesture has two touches—two fingers on the screen moving toward each other from opposite directions.

An **event** encompasses all touches that occur during a multitouch sequence. A multitouch sequence begins when a finger touches the screen and ends when the last finger is lifted. As a finger moves, iOS sends touch objects to the event. An multitouch event is represented by a `UIEvent` object of type `UIEventTypeTouches`.

Each touch object tracks only one finger and lasts only as long as the multitouch sequence. During the sequence, UIKit tracks the finger and updates the attributes of the touch object. These attributes include the phase of the touch, its location in a view, its previous location, and its timestamp.

The **touch phase** indicates when a touch begins, whether it is moving or stationary, and when it ends—that is, when the finger is no longer touching the screen. As depicted in Figure 1-4, an app receives event objects during each phase of any touch.

Figure 1-4 A multitouch sequence and touch phases



Note: A finger is less precise than a mouse pointer. When a user touches the screen, the area of contact is actually elliptical and tends to be slightly lower than the user expects. This "contact patch" varies based on the size and orientation of the finger, the amount of pressure, which finger is used, and other factors. The underlying multitouch system analyzes this information for you and computes a single touch point, so you don't need to write your own code to do this.

An App Receives Touches in the Touch-Handling Methods

During a multitouch sequence, an app sends these messages when there are new or changed touches for a given touch phase; it calls the

- `touchesBegan:withEvent:` method when one or more fingers touch down on the screen.
- `touchesMoved:withEvent:` method when one or more fingers move.
- `touchesEnded:withEvent:` method when one or more fingers lift up from the screen.
- `touchesCancelled:withEvent:` method when the touch sequence is canceled by a system event, such as an incoming phone call.

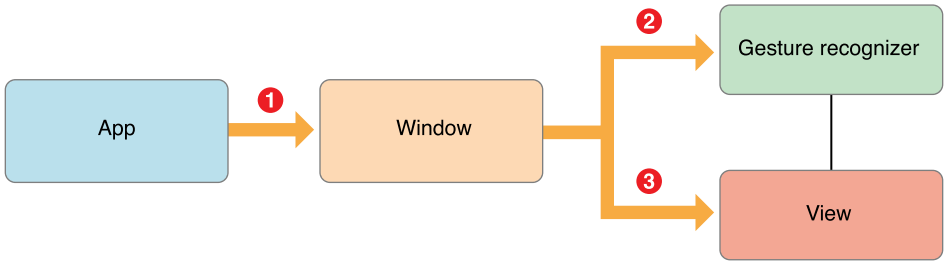
Each of these methods is associated with a touch phase; for example, the `touchesBegan:withEvent:` method is associated with `UITouchPhaseBegan`. The phase of a touch object is stored in its `phase` property.

Note: These methods are not associated with gesture recognizer states, such as `UIGestureRecognizerStateBegan` and `UIGestureRecognizerStateEnded`. Gesture recognizer states strictly denote the phase of the gesture recognizer itself, not the phase of the touch objects that are being recognized.

Regulating the Delivery of Touches to Views

There may be times when you want a view to receive a touch before a gesture recognizer. But, before you can alter the delivery path of touches to views, you need to understand the default behavior. In the simple case, when a touch occurs, the touch object is passed from the `UIApplication` object to the `UIWindow` object. Then, the window first sends touches to any gesture recognizers attached the view where the touches occurred (or to that view's superviews), before it passes the touch to the view object itself.

Figure 1-5 Default delivery path for touch events



Gesture Recognizers Get the First Opportunity to Recognize a Touch

A window delays the delivery of touch objects to the view so that the gesture recognizer can analyze the touch first. During the delay, if the gesture recognizer recognizes a touch gesture, then the window never delivers the touch object to the view, and also cancels any touch objects it previously sent to the view that were part of that recognized sequence.

For example, if you have a gesture recognizer for a discrete gesture that requires a two-fingered touch, this translates to two separate touch objects. As the touches occur, the touch objects are passed from the app object to the window object for the view where the touches occurred, and the following sequence occurs, as depicted in Figure 1-6.

Figure 1-6 Sequence of messages for touches

	❶	❷	❸	❹
Gesture Recognizer	touchesBegan:	touchesMoved:	touchesEnded:	touchesEnded:
tapGestureRecognizer.state	Possible	Possible	Possible	Recognized
View	touchesBegan:	touchesMoved:		touchesCancelled:

1. The window sends two touch objects in the Began phase—through the `touchesBegan:withEvent:` method—to the gesture recognizer. The gesture recognizer doesn't recognize the gesture yet, so its state is Possible. The window sends these same touches to the view that the gesture recognizer is attached to.
2. The window sends two touch objects in the Moved phase—through the `touchesMoved:withEvent:` method—to the gesture recognizer. The recognizer still doesn't detect the gesture, and is still in state Possible. The window then sends these touches to the attached view.
3. The window sends *one* touch object in the Ended phase—through the `touchesEnded:withEvent:` method—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, so it sets its state to Recognized. Just before the first action message is sent, the view calls the `touchesCancelled:withEvent:` method 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 decides that this multitouch sequence it's been analyzing is *not* its gesture. It sets its state to `UIGestureRecognizerStateFailed`. Then the window 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 follows 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` (not Recognized). The window sends all subsequent touch objects in the multitouch sequence to the gesture recognizer but not to the attached view.

Affecting the Delivery of Touches to Views

You can change the values of several `UIGestureRecognizer` properties to alter the default delivery path in certain ways. If you change the default values of these properties, you get the following differences in behavior:

- `delaysTouchesBegan` (default of `NO`)—Normally, the window sends touch objects in the Began and Moved phases to the view and the gesture recognizer. Setting `delaysTouchesBegan` to `YES` prevents the window from delivering touch objects in the Began phase to the view. This ensures that when a gesture recognizer recognizes its gesture, no part of the touch event was delivered to the attached view. Be cautious when setting this property because it can make your interface feel unresponsive.

  This setting provides a similar behavior to 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.

- `delaysTouchesEnded` (default of `YES`)—When this property is set to`YES`, it ensures that a view does not complete an action that the gesture might want to cancel later. When a gesture recognizer is analyzing a touch event, the window does not deliver touch objects in the Ended phase to the attached view. If a gesture recognizer recognizes its gesture, the touch objects are canceled. If the gesture recognizer does not recognize its gesture, the window delivers these objects to the view through a `touchesEnded:withEvent:` message. Setting this property to `NO` allows the view to analyze touch objects in the Ended phase at the same time as the gesture recognizer.

  Consider, for example, that a view has a tap gesture recognizer that requires two taps, and the user double taps the view. With the property set to `YES`, the view gets `touchesBegan:withEvent:`, `touchesBegan:withEvent:`,`touchesCancelled:withEvent:`,and `touchesCancelled:withEvent:`. If this property is set to `NO`, the view gets the following sequence of messages: `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, `touchesBegan:withEvent:`, and `touchesCancelled:withEvent:`, which means that in `touchesBegan:withEvent:`, the view can recognize a double tap.

If a gesture recognizer detects a touch that it determines is not part of its gesture, it can pass the touch directly to its view. To do this, the gesture recognizer calls `ignoreTouch:forEvent:` on itself, passing in the touch object.

Creating 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;
```

These methods have the same exact signature and behavior as the corresponding touch-event handling methods described earlier in "An App Receives Touches in the Touch-Handling Methods" (page 24). In all of the methods you override, you must call the superclass implementation, even if the method has a null implementation.

Notice that the `state` property in `UIGestureRecognizerSubclass.h` is now `readwrite` instead of `readonly`, as it is in `UIGestureRecognizer.h`. Your subclass changes its state by assigning `UIGestureRecognizerState` constants to that property.

Implementing the Touch-Event Handling Methods for a Custom Gesture Recognizer

The heart of the implementation for a custom gesture recognizer are the four methods: `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:`. Within these methods, you translate low-level touch events into gesture recognition by setting a gesture recognizer's state. Listing 1-8 creates a gesture recognizer for a discrete single-touch checkmark gesture. It records the midpoint of the gesture—the point at which the upstroke begins—so that clients can obtain this value.

This example has only a single view, but most apps have many views. In general, you should convert touch locations to the screen's coordinate system so that you can correctly recognize gestures that span multiple views.

Listing 1-8 Implementation of a checkmark gesture recognizer

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

// Implemented in your custom subclass
- (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;
    UIWindow *win = [self.view window];
    CGPoint nowPoint = [touches.anyObject locationInView:win];
    CGPoint nowPoint = [touches.anyObject locationInView:self.view];
    CGPoint prevPoint = [touches.anyObject previousLocationInView:self.view];

    // strokeUp is a property
    if (!self.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) {
            self.strokeUp = YES;
        } else {
            self.state = UIGestureRecognizerStateFailed;
        }
    }
}

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

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesCancelled:touches withEvent:event];
```

```
    self.midPoint = CGPointZero;
    self.strokeUp = NO;
    self.state = UIGestureRecognizerStateFailed;
}
```

State transitions for discrete and continuous gestures are different, as described in "Gesture Recognizers Operate in a Finite State Machine" (page 17). When you create a custom gesture recognizer, you indicate whether it is discrete or continuous by assigning it the relevant states. As an example, the checkmark gesture recognizer in Listing 1-8 never sets the state to Began or Changed, because it's discrete.

The most important thing you need to do when subclassing a gesture recognizer is to set the gesture recognizer's `state` accurately. iOS needs to know the state of a gesture recognizer in order for gesture recognizers to interact as expected. For example, if you want to permit simultaneous recognition or require a gesture recognizer to fail, iOS needs to understand the current state of your recognizer.

For more about creating custom gesture recognizers, see *WWDC 2012: Building Advanced Gesture Recognizers* .

Resetting a Gesture Recognizer's State

If your gesture recognizer transitions to Recognized/Ended, Canceled, or Failed, the `UIGestureRecognizer` class calls the `reset` method just before the gesture recognizer transitions back to Possible.

Implement the `reset` method to reset any internal state so that your recognizer is ready for a new attempt at recognizing a gesture, as in Listing 1-9. After a gesture recognizer returns from this method, it receives no further updates for touches that are in progress.

Listing 1-9 Resetting a gesture recognizer

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

Event Delivery: The Responder Chain

When you design your app, it's likely that you want to respond to events dynamically. For example, a touch can occur in many different objects onscreen, and you have to decide which object you want to respond to a given event and understand how that object receives the event.

When a user-generated event occurs, UIKit creates an event object containing the information needed to process the event. Then it places the event object in the active app's event queue. For touch events, that object is a set of touches packaged in a `UIEvent` object. For motion events, the event object varies depending on which framework you use and what type of motion event you are interested in.

An event travels along a specific path until it is delivered to an object that can handle it. First, the singleton `UIApplication` object takes an event from the top of the queue and dispatches it for handling. Typically, it sends the event to the app's key window object, which passes the event to an initial object for handling. The initial object depends on the type of event.

- **Touch events.** For touch events, the window object first tries to deliver the event to the view where the touch occurred. That view is known as the hit-test view. The process of finding the hit-test view is called *hit-testing*, which is described in "Hit-Testing Returns the View Where a Touch Occurred" (page 31).
- **Motion and remote control events.** With these events, the window object sends the shaking-motion or remote control event to the first responder for handling. The first responder is described in "The Responder Chain Is Made Up of Responder Objects" (page 33).

The ultimate goal of these event paths is to find an object that can handle and respond to an event. Therefore, UIKit first sends the event to the object that is best suited to handle the event. For touch events, that object is the hit-test view, and for other events, that object is the first responder. The following sections explain in more detail how the hit-test view and first responder objects are determined.

Hit-Testing Returns the View Where a Touch Occurred

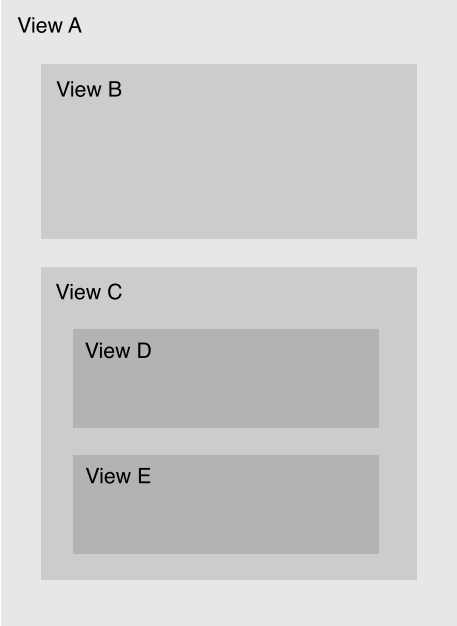
iOS uses hit-testing to find the view that is under a touch. Hit-testing involves checking whether a touch is within the bounds of any relevant view objects. If it is, it recursively checks all of that view's subviews. The lowest view in the view hierarchy that contains the touch point becomes the **hit-test view**. After iOS determines the hit-test view, it passes the touch event to that view for handling.

To illustrate, suppose that the user touches view E in Figure 2-1. iOS finds the hit-test view by checking the subviews in this order:

1. The touch is within the bounds of view A, so it checks subviews B and C.
2. The touch is not within the bounds of view B, but it's within the bounds of view C, so it checks subviews D and E.
3. The touch is not within the bounds of view D, but it's within the bounds of view E.

   View E is the lowest view in the view hierarchy that contains the touch, so it becomes the hit-test view.

Figure 2-1 Hit-testing returns the subview that was touched



The `hitTest:withEvent:` method returns the hit test view for a given `CGPoint` and `UIEvent`. The `hitTest:withEvent:` method begins by calling the `pointInside:withEvent:` method on itself. If the point passed into `hitTest:withEvent:` is inside the bounds of the view, `pointInside:withEvent:` returns `YES`. Then, the method recursively calls `hitTest:withEvent:` on every subview that returns `YES`.

If the point passed into `hitTest:withEvent:` is not inside the bounds of the view, the first call to the `pointInside:withEvent:` method returns `NO`, the point is ignored, and `hitTest:withEvent:` returns `nil`. If a subview returns `NO`, that whole branch of the view hierarchy is ignored, because if the touch did not occur in that subview, it also did not occur in any of that subview's subviews. This means that any point in a subview that is outside of its superview can't receive touch events because the touch point has to be within the bounds of the superview *and* the subview. This can occur if the subview's `clipsToBounds` property is set to `NO`.

> **Note:** A touch object is associated with its hit-test view for its lifetime, even if the touch later moves outside the view.

The hit-test view is given the first opportunity to handle a touch event. If the hit-test view cannot handle an event, the event travels up that view's chain of responders as described in "The Responder Chain Is Made Up of Responder Objects" (page 33) until the system finds an object that can handle it.

The Responder Chain Is Made Up of Responder Objects

Many types of events rely on a responder chain for event delivery. The **responder chain** is a series of linked responder objects. It starts with the first responder and ends with the application object. If the first responder cannot handle an event, it forwards the event to the next responder in the responder chain.

A **responder object** is an object that can respond to and handle events. The `UIResponder` class is the base class for all responder objects, and it defines the programmatic interface not only for event handling but also for common responder behavior. Instances of the `UIApplication`, `UIViewController`, and `UIView` classes are responders, which means that all views and most key controller objects are responders. Note that Core Animation layers are not responders.

The **first responder** is designated to receive events first. Typically, the first responder is a view object. An object becomes the first responder by doing two things:

1. Overriding the `canBecomeFirstResponder` method to return `YES`.
2. Receiving a `becomeFirstResponder` message. If necessary, an object can send itself this message.

> **Note:** Make sure that your app has established its object graph before assigning an object to be the first responder. For example, you typically call the `becomeFirstResponder` method in an override of the `viewDidAppear:` method. If you try to assign the first responder in `viewWillAppear:`, your object graph is not yet established, so the `becomeFirstResponder` method returns `NO`.

Events are not the only objects that rely on the responder chain. The responder chain is used in all of the following:

- **Touch events.** If the hit-test view cannot handle a touch event, the event is passed up a chain of responders that starts with the hit-test view.

- **Motion events.** To handle shake-motion events with UIKit, the first responder must implement either the `motionBegan:withEvent:` or `motionEnded:withEvent:` method of the `UIResponder` class, as described in "Detecting Shake-Motion Events with UIEvent" (page 57).
- **Remote control events.** To handle remote control events, the first responder must implement the `remoteControlReceivedWithEvent:` method of the `UIResponder` class.
- **Action messages.** When the user manipulates a control, such as a button or switch, and the target for the action method is `nil`, the message is sent through a chain of responders starting with the control view.
- **Editing-menu messages.** When a user taps the commands of the editing menu, iOS uses a responder chain to find an object that implements the necessary methods (such as `cut:`, `copy:`, and `paste:`). For more information, see "Displaying and Managing the Edit Menu" and the sample code project, *CopyPasteTile*.
- **Text editing.** When a user taps a text field or a text view, that view automatically becomes the first responder. By default, the virtual keyboard appears and the text field or text view becomes the focus of editing. You can display a custom input view instead of the keyboard if it's appropriate for your app. You can also add a custom input view to any responder object. For more information, see "Custom Views for Data Input".

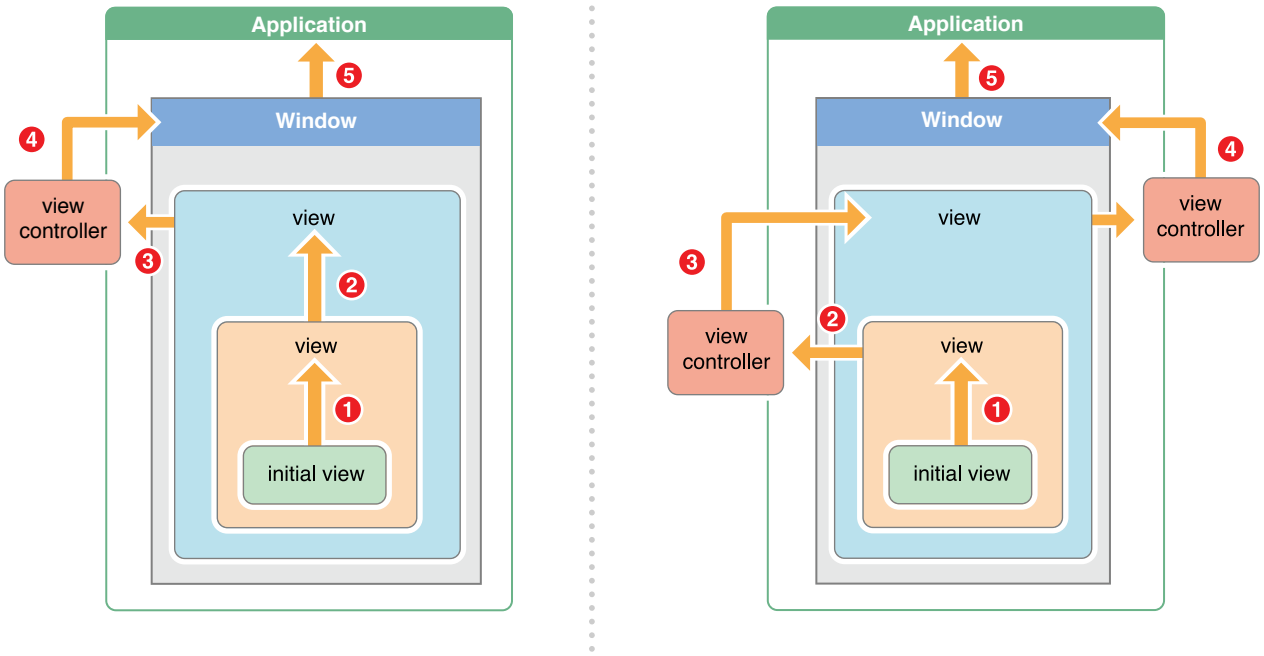
UIKit automatically sets the text field or text view that a user taps to be the first responder; Apps must explicitly set all other first responder objects with the `becomeFirstResponder` method.

The Responder Chain Follows a Specific Delivery Path

If the initial object—either the hit-test view or the first responder—doesn't handle an event, UIKit passes the event to the next responder in the chain. Each responder decides whether it wants to handle the event or pass it along to its own next responder by calling the `nextResponder` method.This process continues until a responder object either handles the event or there are no more responders.

The responder chain sequence begins when iOS detects an event and passes it to an initial object, which is typically a view. The initial view has the first opportunity to handle an event. Figure 2-2 shows two different event delivery paths for two app configurations. An app's event delivery path depends on its specific construction, but all event delivery paths adhere to the same heuristics.

Figure 2-2 The responder chain on iOS



For the app on the left, the event follows this path:

1. The **initial view** attempts to handle the event or message. If it can't handle the event, it passes the event to its superview, because the initial view is not the top most view in its view controller's view hierarchy.
2. The **superview** attempts to handle the event. If the superview can't handle the event, it passes the event to its superview, because it is still not the top most view in the view hierarchy.
3. The **topmost view** in the view controller's view hierarchy attempts to handle the event. If the topmost view can't handle the event, it passes the event to its view controller.
4. The **view controller** attempts to handle the event, and if it can't, passes the event to the window.
5. If the **window object** can't handle the event, it passes the event to the singleton app object.
6. If the **app object** can't handle the event, it discards the event.

The app on the right follows a slightly different path, but all event delivery paths follow these heuristics:

1. A view passes an event up its view controller's view hierarchy until it reaches the topmost view.

2. The topmost view passes the event to its view controller.
3. The view controller passes the event to its topmost view's superview.

   Steps 1-3 repeat until the event reaches the root view controller.
4. The root view controller passes the event to the window object.
5. The window passes the event to the app object.

Important: If you implement a custom view to handle remote control events, action messages, shake-motion events with UIKit, or editing-menu messages, don't 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 and let UIKit handle the traversal of the responder chain for you.

Multitouch Events

Generally, you can handle almost all of your touch events with the standard controls and gesture recognizers in UIKit. Gesture recognizers allow you to separate the recognition of a touch from the action that the touch produces. In some cases, you want to do something in your app—such as drawing under a touch—where there is no benefit to decoupling the touch recognition from the effect of the touch. If the view's contents are intimately related to the touch itself, you can handle touch events directly. You receive touch events when the user touches your view, interpret those events based on their properties and then respond appropriately.

Creating a Subclass of UIResponder

For your app to implement custom touch-event handling, first create a subclass of a responder class. This subclass could be any one of the following:

Subclass	Why you might choose this subclass as your first responder
`UIView`	Subclass `UIView` to implement a custom drawing view.
`UIViewController`	Subclass `UIViewController` if you are also handling other types of events, such as shake motion events.
`UIControl`	Subclass `UIControl` to implement custom controls with touch behavior.
`UIApplication` or `UIWindow`	This would be rare, because you typically do not subclass `UIApplication` or `UIWindow`.

Then, for instances of your subclass to receive multitouch events:

1. Your subclass must implement the `UIResponder` methods for touch-event handling, described in "Implementing the Touch-Event Handling Methods in Your Subclass" (page 38).
2. The view receiving touches must have its `userInteractionEnabled` property set to `YES`. If you are subclassing a view controller, the view that it manages must support user interactions.
3. The view receiving touches must be visible; it can't be transparent or hidden.

Implementing the Touch-Event Handling Methods in Your Subclass

iOS recognizes touches as part of a **multitouch sequence**. During a multitouch sequence, the app sends a series of event messages to the target responder. To receive and handle these messages, the responder object's class must implement the following `UIResponder` 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;
```

> **Note:** These methods have the same signature as the methods you override to create a custom gesture recognizer, as discussed in "Creating a Custom Gesture Recognizer" (page 27).

Each of these touch methods correspond to a touch phase: Began, Moved, Ended, and Canceled. When there are new or changed touches for a given phase, the app object calls one of these methods. Each method takes two parameters: a set of touches and an event.

The set of touches is a set (NSSet) of `UITouch` objects, representing new or changed touches for that phase. For example, when a touch transitions from the Began phase to the Moved phase, the app calls the `touchesMoved:withEvent:` method. The set of touches passed in to the `touchesMoved:withEvent:` method will now include this touch and all other touches in the Moved phase. The other parameter is an event (`UIEvent` object) that includes *all* touch objects for the event. This differs from the set of touches because some of the touch objects in the event may not have changed since the previous event message.

All views that process touches expect to receive a full touch-event stream, so when you create your subclass, keep in mind the following rules:

- If your custom responder is a subclass of `UIView` or `UIViewController`, you should implement all of the event handling methods.
- If you subclass any other responder class, you can have a null implementation for some of the event methods.
- In all methods, be sure to call the superclass implementation of the method.

If you prevent a responder object from receiving touches for a certain phase of an event, the resulting behavior may be undefined and probably undesirable.

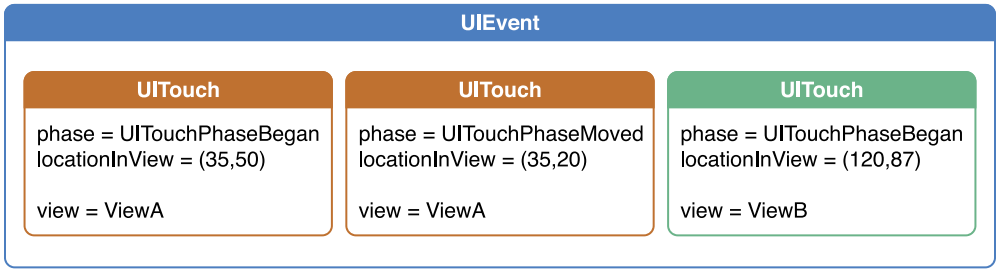
If a responder creates persistent objects while handling events, it should implement the `touchesCancelled:withEvent:` method to dispose of those objects if the system cancels the sequence. Cancellation occurs when an external event—for example, an incoming phone call—disrupts the current app's event processing. Note that a responder object should also dispose of any persistent objects when it receives the last `touchesEnded:withEvent:` message for a multitouch sequence. See "Forwarding Touch Events" (page 51) to find out how to determine the last `UITouchPhaseEnded` touch object in a multitouch sequence.

Tracking the Phase and Location of a Touch Event

iOS tracks touches in a multitouch sequence. It records attributes for each of them, including the phase of the touch, its location in a view, its previous location, and its timestamp. Use these properties to determine how to respond to a touch.

A touch object stores phase information in the `phase` property, and each phase corresponds to one of the touch event methods. A touch object stores location in three ways: the window in which the touch occurred, the view within that window, and the exact location of the touch within that view. Figure 3-1 (page 39) shows an example event with two touches in progress.

Figure 3-1 Relationship of a UIEvent object and its UITouch objects



When a finger touches the screen, that touch is associated with both the underlying window and the view for the lifetime of the event, even if the event is later passed to another view for handling. Use a touch's location information to determine how to respond to the touch. For example, if two touches occur in quick succession, they are treated as a double tap only if they both occurred in the same view. A touch object stores both its current location and its previous location, if there is one.

Retrieving and Querying Touch Objects

Within an event handling method, you get information about the event by retrieving touch objects from:

- **The set object.** The passed-in `NSSet` contains all touches that are new or have changed in the phase represented by the method, such as `UITouchPhaseBegan` for the `touchesBegan:withEvent:` method.

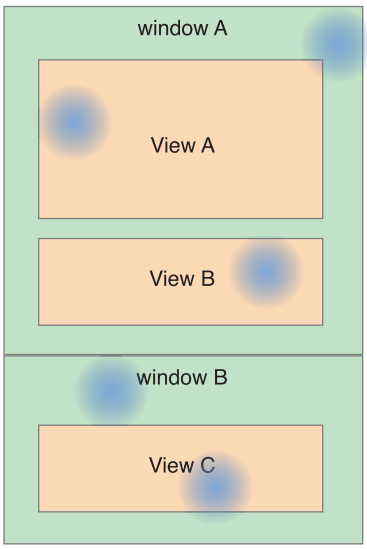
- **The event object.** The passed-in `UIEvent` object contains all of the touches for a given multitouch sequence.

The `multipleTouchEnabled` property is set to `NO` by default, which means that a view receives only the first touch in a multitouch sequence. When this property is disabled, you can retrieve a touch object by calling the `anyObject` method on the set object because there is only one object in the set.

If you want to know the location of a touch, use the `locationInView:` method. By passing the parameter `self` to this method, you get the location of the touch in the coordinate system of the receiving view. Similarly, the `previousLocationInView:` method tells you the previous location of the touch. You can also determine how many taps a touch has (`tapCount`), when the touch was created or last mutated (`timestamp`), and what phase the touch is in (`phase`).

If you are interested in touches that have not changed since the last phase or that are in a different phase than the touches in the passed-in set, you can find those in the event object. Figure 3-2 depicts an event object that contains touch objects. To get all of these touch objects, call the `allTouches` method on the event object.

Figure 3-2 All touches for a given touch event

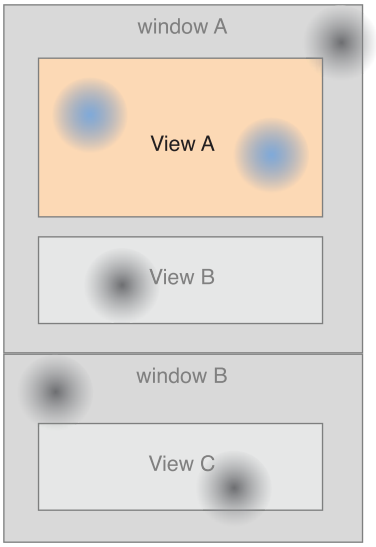


If you are interested only in touches associated with a specific window, call the `touchesForWindow:` method of the `UIEvent` object. Figure 3-3 shows all the touches for window A.

Figure 3-3 All touches belonging to a specific window

If you want to get the touches associated with a specific view, call the `touchesForView:` method of the event object. Figure 3-4 shows all the touches for view A.

Figure 3-4 All touches belonging to a specific view



Handling Tap Gestures

Besides being able to recognize a tap gesture in your app, you'll probably want to distinguish a single tap, a double tap, or even a triple tap. Use a touch's `tapCount` property to determine the number of times the user tapped a view.

The best place to find this value is the `touchesEnded:withEvent:` method, because it corresponds to when the user lifts a finger from a tap. By looking for the tap count in the touch-up phase—when the sequence has ended—you ensure that the finger is really tapping and not, for instance, touching down and dragging. Listing 3-1 shows an example of how to determine whether a double tap occurred in one of your views.

Listing 3-1 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 *aTouch in touches) {
        if (aTouch.tapCount >= 2) {
             // The view responds to the tap
             [self respondToDoubleTapGesture:aTouch];
        }
    }
}

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

Handling Swipe and Drag Gestures

Horizontal and vertical swipes are a simple type of gesture that you can track. To detect a swipe gesture, track the movement of the user's finger along the desired axis of motion. Then, decide whether the movement is a swipe by examining the following questions for each gesture:

- Did the user's finger move far enough?

- Did the finger move in a relatively straight line?
- Did the finger move quickly enough to call it a swipe?

To answer these questions, store the touch's initial location and compare its location as the touch moves.

Listing 3-2 shows some basic tracking methods you could use to detect horizontal swipes in a view. In this example, a view has a `startTouchPosition` property that it uses to store a touch's initial location. In the `touchesEnded:` method, it compares the ending touch position to the starting location to determine if it is a swipe. If the touch moves too far vertically or does not move far enough, it is not considered to be a swipe. This example does not show the implementation for the `myProcessRightSwipe:` or `myProcessLeftSwipe:` methods, but the custom view would handle the swipe gesture there.

Listing 3-2 Tracking a swipe gesture in a view

```
#define HORIZ_SWIPE_DRAG_MIN  12
#define VERT_SWIPE_DRAG_MAX    4

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    // startTouchPosition is a property
    self.startTouchPosition = [aTouch locationInView:self];
}

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

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

    //  Check if direction of touch is horizontal and long enough
    if (fabsf(self.startTouchPosition.x - currentTouchPosition.x) >=
HORIZ_SWIPE_DRAG_MIN &&
        fabsf(self.startTouchPosition.y - currentTouchPosition.y) <=
VERT_SWIPE_DRAG_MAX)
    {
        // If touch appears to be a swipe
        if (self.startTouchPosition.x < currentTouchPosition.x) {
```

```
            [self myProcessRightSwipe:touches withEvent:event];
        } else {
            [self myProcessLeftSwipe:touches withEvent:event];
    }
    self.startTouchPosition = CGPointZero;
}

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

Notice that this code does not check the location of the touch in the middle of the gesture, which means that a gesture could go all over the screen but still be considered a swipe if its start and end points are in line. A more sophisticated swipe gesture recognizer should also check middle locations in the `touchesMoved:withEvent:` method. To detect swipe gestures in the vertical direction, you would use similar code but would swap the *x* and *y* components.

Listing 3-3 shows an even simpler implementation of tracking a single touch, this time the user is dragging a view around the screen. Here, the custom view class fully implements only the `touchesMoved:withEvent:` method. This method computes a delta value between the touch's current and previous locations in the view. It then uses this delta value to reset the origin of the view's frame.

Listing 3-3 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 typically involve only one touch and are simple to track. Handling a touch event consisting of two or more touches is more challenging. You may have to track all touches through all phases, recording the touch attributes that have changed and altering internal state appropriately. To track and handle multiple touches, you need to:

- Set the view's `multipleTouchEnabled` property 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 information about a touch's state so that you can compare touches later. As an example, say you want to compare the final location of each touch with its original location. In the `touchesBegan:withEvent:` method, you get 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 touch object to get the object's original location and compare it with its current location.

> **Important:** Use a `CFDictionaryRef` data type rather than an `NSDictionary` object to track touches, because `NSDictionary` copies its keys. The `UITouch` class does not adopt the `NSCopying` protocol, which is required for object copying.

Listing 3-4 illustrates how to store the starting locations of `UITouch` objects in a Core Foundation dictionary. The `cacheBeginPointForTouches:` method stores the location of each touch in the superview's coordinates so that it has a common coordinate system to compare the location of all of the touches.

Listing 3-4 Storing the beginning locations of multiple touches

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
     [self cacheBeginPointForTouches: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 3-5 builds on the previous example. It illustrates how to retrieve the initial locations from the dictionary. Then, it gets the current locations of the same touches so that you can use these values to compute an affine transformation (not shown).

Listing 3-5 Retrieving the initial locations of touch objects

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    CGAffineTransform newTransform = [self incrementalTransformWithTouches:touches];
}

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

     // Other code here
     CGAffineTransform transform = CGAffineTransformIdentity;
```

```
    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
    return transform;
}
```

The next example, Listing 3-6, does not use a dictionary to track touch mutations; however, it handles multiple touches during an event. It shows a custom `UIView` object responding to touches by animating the movement of a "Welcome" placard as a finger moves it around the screen. It also changes the language of the placard when the user double taps. This example comes from the *MoveMe* sample code project, which you can examine to get a better understanding of the event handling context.

Listing 3-6 Handling a complex multitouch sequence

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

    // App supports only single touches, so anyObject retrieves just
    // that touch from touches
    UITouch *touch = [touches anyObject];

    // Move the placard view only if the touch was in the placard view
    if ([touch view] != placardView) {
        // In case of a double tap outside the placard view, update
        // the placard's display string
        if ([touch tapCount] == 2) {
            [placardView setupNextDisplayString];
        }
        return;
```

```
    }

    // Animate the first touch
    CGPoint touchPoint = [touch locationInView:self];
    [self animateFirstTouchAtPoint:touchPoint];
}

}

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

    UITouch *touch = [touches anyObject];

    // If the touch was in the placardView, move the placardView to its location
    if ([touch view] == placardView) {
        CGPoint location = [touch locationInView:self];
        placardView.center = location;
    }
}



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

    UITouch *touch = [touches 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];
    }
}
```

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

    // To impose as little impact on the device as possible, simply set
    // the placard view's center and transformation to the original values
    placardView.center = self.center;
    placardView.transform = CGAffineTransformIdentity;
}
```

To find out when the last finger in a multitouch sequence is lifted from a view, see how many touch objects are in the passed-in set and how many are in the passed-in `UIEvent` object. If the number is the same, then the multitouch sequence has concluded. Listing 3-7 illustrates how to do this in code.

Listing 3-7 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 a passed-in set contains all touch objects associated with the view that are new or changed for a given phase, whereas the touch objects returned from the `touchesForView:` method includes *all* objects associated with the specified view.

Specifying Custom Touch Event Behavior

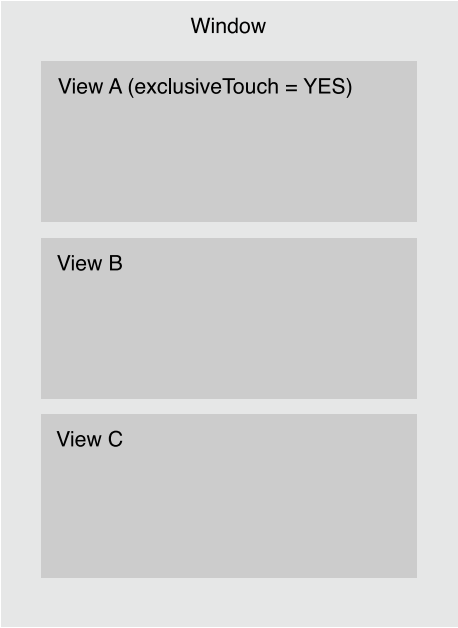
Customize the way your app handles touches by altering the behavior of a specific gesture, a specific view, or all of the touch events in your app. You can alter the stream of touch events in the following ways:

- **Turn 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 by setting the related attribute in Interface Builder or by programmatically setting the `multipleTouchEnabled` property of your view to `YES`.

- **Restrict event delivery to a single view**. By default, a view's `exclusiveTouch` property is set to `NO`, which means that one view does not block other views in a window from receiving touches. If you set the property to `YES` for a specific view, then that view receives touches if—and only if—it is the only view tracking touches.

  If your views are nonexclusive, a user can touch one finger in one view and another finger in another view, and each view can track its touch simultaneously. Now imagine that you have views set up as in Figure 3-5 and that view A is an exclusive-touch view. If the user touches inside A, it recognizes the touch. But if a user holds one finger inside view B and also touches inside view A, then view A does not receive the touch because it was not the only view tracking touches. Similarly, if a user holds one finger inside view A and also touches inside view B, then view B does not receive the touch because view A is the only view tracking touches. At any time, the user can still touch both B and C, and those views can track their touches simultaneously.

Figure 3-5 Restricting event delivery with an exclusive-touch view



- **Restrict event delivery to subviews**. A custom `UIView` class can override `hitTest:withEvent:` so that multitouch events are not delivered to a specific subview. See "Intercepting Touches by Overriding Hit-Testing" (page 51) for a discussion of this technique.

You can also turn off touch-event delivery completely, or just for a period of time:

- **Turn off delivery of touch events**. Set a view's `userInteractionEnabled` property to `NO` to turn off delivery of touch events. Note that a view also does not receive touch events if it's hidden or transparent.

- **Turn off delivery of touch events for a period**. Sometimes you want to temporarily turn off event delivery—for example, while your code is performing animations. Your app can call the `beginIgnoringInteractionEvents` method to stop receiving touch events, and then later call the `endIgnoringInteractionEvents` method to resume touch-event delivery.

Intercepting Touches by Overriding Hit-Testing

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 you chose to handle touches at the superview level—meaning that your subviews do not implement the `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, or `touchesMoved:withEvent:` methods—then the superview should override `hitTest:withEvent:` to return itself rather than any of its subviews.

Overriding hit-testing ensures that the superview receives all touches because, by setting itself as the hit-test view, the superview intercepts and receives touches that are normally passed to the subview first. If a superview does not override `hitTest:withEvent:`, touch events are associated with the subviews where they first occurred and are never sent to the superview.

Recall that there are two hit-test methods: the `hitTest:withEvent:` method of views and the `hitTest:` method of layers, as described in "Hit-Testing Returns the View Where a Touch Occurred" (page 31). You rarely need to call these methods yourself. It's more likely that you will override them to intercept touch events from subviews. However, sometimes responders perform hit-testing prior to event forwarding (see "Forwarding Touch Events" (page 51)).

Forwarding Touch Events

To forward touch events to another responder object, send the appropriate touch-event handling messages to that object. Use this technique with caution because UIKit classes are not designed to receive touches that are not bound to them. For a responder object to handle a touch, the touch's `view` property must hold a reference to the responder. If you want to conditionally forward touches to other responders in your app, all of the responders should be instances of your own `UIView` subclass.

For example, let's say an app has three custom views: A, B, and C. When the user touches view A, the app's window determines that it is the hit-test view and sends to it the initial touch event. 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 of this forwarding, and views B and C must be able to deal with touches that are not bound to them.

Event forwarding often requires analyzing touch objects to determine where they should be forwarded. There are a couple of 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 "Intercepting Touches by Overriding Hit-Testing" (page 51)).
- Override `sendEvent:` in a custom subclass of `UIWindow`, analyze touches, and forward them to the appropriate responders.

Overriding the `sendEvent:` method allows you to monitor the events your app receives. Both the `UIApplication` object and each `UIWindow` object dispatch events in the `sendEvent:` method, so this method serves as a funnel point for events coming in to an app. This is something that very few apps need to do and, if you do override `sendEvent:`, be sure to invoke the superclass implementation—`[super sendEvent:theEvent]`. Never tamper with the distribution of events.

Listing 3-8 illustrates this technique in a subclass of `UIWindow`. In this example, events are forwarded to a custom helper responder that performs affine transformations on the view that it is associated with.

Listing 3-8 Forwarding touch events to helper responder objects

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

        // Sort touches by phase to handle--similar 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 (!canceled) canceled = [NSMutableSet set];
                    [canceled addObject:touch];
                    break;
                default:
                    break;
            }
        }
        // Call 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 (canceled) [gesture touchesCancelled:canceled withEvent:event];
    }
    [super sendEvent:event];
}
```

Notice that the overriding subclass invokes the superclass implementation of the `sendEvent:` method. This is important to the integrity of the touch-event stream.

Best Practices for Handling Multitouch Events

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

- Always implement the event cancelation methods.

  In your implementation, restore the state of a view to what it was before the current multitouch sequence. If you don't, your view could be left in an inconsistent state, or in some cases, another view could receive the cancelation message.

- If you handle events in a subclass of `UIView`, `UIViewController`, or `UIResponder`:
  - Implement all of the event handling methods, even if your implementations of those methods do nothing.

    - 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.
    - In the methods you do implement, be sure to call the superclass implementation. For example, `[super touchesBegan:touches withEvent:event]`.
- Do not forward events to other responder objects of the UIKit framework.

  Instead, forward events to instances of your own subclasses of `UIView`. Additionally, make sure these responder objects are aware that event forwarding is taking place and that they can receive touches that are not bound to them.
- Do not explicitly send events up the responder chain through the `nextResponder` method; instead, invoke the superclass implementation and let UIKit handle responder-chain traversal.
- Do not use round-to-integer code in your touch handling because you lose precision.

  iOS reports touches in a 320x480 coordinate space to maintain source compatibility. However, on high-resolution devices, the resolution is actually twice as high in each dimension (640x960), which means that touches can land on half-point boundaries on high-resolution devices. On older devices, touches land only on full-point boundaries.

Motion Events

Users generate motion events when they move, shake, or tilt the device. These motion events are detected by the device hardware, specifically, the accelerometer and the gyroscope.

The **accelerometer** is actually made up of three accelerometers, one for each axis—x, y, and z. Each one measures changes in velocity over time along a linear path. Combining all three accelerometers lets you detect device movement in any direction and get the device's current orientation. Although there are three accelerometers, the remainder of this document refers to them as a single entity. The **gyroscope** measures the rate of rotation around the three axes.

All motion events originate from the same hardware. There are several different ways that you can access that hardware data, depending on your app's needs:

- If you need to detect the general orientation of a device, but you don't need to know the orientation vector, use the `UIDevice` class. See "Getting the Current Device Orientation with UIDevice" (page 55) for more information.
- If you want your app to respond when a user shakes the device, you can use the UIKit motion-event handling methods to get information from the passed-in `UIEvent` object. See "Detecting Shake-Motion Events with UIEvent" (page 57) for more information.
- If neither the `UIDevice` nor the `UIEvent` classes are sufficient, it's likely you'll want to use the Core Motion framework to access the accelerometer, gyroscope, and device motion classes. See "Capturing Device Movement with Core Motion" (page 59) for more information.

Getting the Current Device Orientation with UIDevice

Use the methods of the `UIDevice` class when you need to know only the general orientation of the device and not the exact vector of orientation. Using `UIDevice` is simple and doesn't require you to calculate the orientation vector yourself.

Before you can get the current orientation, you need to tell the `UIDevice` class to begin generating device orientation notifications by calling the `beginGeneratingDeviceOrientationNotifications` method. This turns on the accelerometer hardware, which may be off to conserve battery power. Listing 4-1 demonstrates this in the `viewDidLoad` method.

After enabling orientation notifications, get the current orientation from the `orientation` property of the `UIDevice` object. If you want to be notified when the device orientation changes, register to receive `UIDeviceOrientationDidChangeNotification` notifications. The device orientation is reported using `UIDeviceOrientation` constants, indicating whether the device is in landscape mode, portrait mode, screen-side up, screen-side down, and so on. These constants indicate the physical orientation of the device and don't necessarily correspond to the orientation of your app's user interface.

When you no longer need to know the orientation of the device, always disable orientation notifications by calling the `UIDevice` method, `endGeneratingDeviceOrientationNotifications`. This gives the system the opportunity to disable the accelerometer hardware if it's not being used elsewhere, which preserves battery power.

Listing 4-1 Responding to changes in device orientation

```
-(void) viewDidLoad {
     // Request to turn on accelerometer and begin receiving accelerometer events
     [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
     [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(orientationChanged:) name:UIDeviceOrientationDidChangeNotification
 object:nil];
}

- (void)orientationChanged:(NSNotification *)notification {
     // Respond to changes in device orientation
}

-(void) viewDidDisappear {
     // Request to stop receiving accelerometer events and turn off accelerometer
     [[NSNotificationCenter defaultCenter] removeObserver:self];
     [[UIDevice currentDevice] endGeneratingDeviceOrientationNotifications];
}
```

For another example of responding to `UIDevice` orientation changes, see the *AlternateViews* sample code project.

Detecting Shake-Motion Events with UIEvent

When users shake a device, iOS evaluates the accelerometer data. If the data meets certain criteria, iOS interprets the shaking gesture and creates a `UIEvent` object to represent it. Then, it sends the event object to the currently active app for processing. Note that your app can respond to shake-motion events and device orientation changes at the same time.

Motion events are simpler than touch events. The system tells an app when a motion starts and stops, but not when each individual motion occurs. And, motion events include only an event type (`UIEventTypeMotion`), event subtype (`UIEventSubtypeMotionShake`), and timestamp.

Designating a First Responder for Motion Events

To receive motion events, designate a responder object as the first responder. This is the responder object that you want to handle the motion events. Listing 4-2 shows how a responder can make itself the first responder.

Listing 4-2 Becoming first responder

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

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

Motion events use the responder chain to find an object that can handle the event. When the user starts shaking the device, iOS sends the first motion event to the first responder. If the first responder doesn't handle the event, it progresses up the responder chain. See "The Responder Chain Follows a Specific Delivery Path" (page 34) for more information. 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` (the default), iOS displays a sheet with Undo and Redo commands.

Implementing the Motion-Handling Methods

There are three motion-handling methods: `motionBegan:withEvent:`, `motionEnded:withEvent:`, and `motionCancelled:withEvent:`. To handle motion events, you must implement either the `motionBegan:withEvent:` method or the `motionEnded:withEvent:` method, and sometimes both. A

responder should also implement the `motionCancelled:withEvent:` method to respond when iOS cancels a motion event. An event is canceled if the shake motion is interrupted or if iOS determines that the motion is not valid after all—for example, if the shaking lasts too long.

Listing 4-3 is extracted from the sample code project, *GLPaint*. In this app, the user paints on the screen, and then shakes the device to erase the painting. This code detects whether a shake has occurred in the `motionEnded:withEvent:` method, and if it has, posts a notification to perform the shake-to-erase functionality.

Listing 4-3 Handling a motion event

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    if (motion == UIEventSubtypeMotionShake)
    {
        // User was shaking the device. Post a notification named "shake."
        [[NSNotificationCenter defaultCenter] postNotificationName:@"shake"
object:self];
     }
}
```

Note: Besides its simplicity, another reason to consider using shake-motion events, instead of Core Motion, is that you can simulate shake-motion events in iOS Simulator as you test and debug your app. For more information about iOS Simulator, see *iOS Simulator User Guide*.

Setting and Checking Required Hardware Capabilities for Motion Events

Before your app can access device-related features, such as accelerometer data, you must add a list of required capabilities to your app. Specifically, you add keys to your app's `Info.plist` file. At runtime, iOS launches your app only if the device has the required capabilities. For example, if your app relies on gyroscope data, list the gyroscope as required so that your app doesn't launch on devices without a gyroscope. The App Store also uses this list to inform users so that they can avoid downloading apps that they can't run.

Declare your app's required capabilities by adding keys to your app's property list. There are two `UIRequiredDeviceCapabilities` keys for motion events, based on hardware source:

- `accelerometer`
- `gyroscope`

> **Note:** You don't need to include the accelerometer key if your app detects only device orientation changes.

You can use either an array or a dictionary to specify the key-values. If you use an array, list each required feature as a key in the array. If you use a dictionary, specify a Boolean value for each required key in the dictionary. In both cases, not listing a key for a feature indicates that the feature is not required. For more information, see "UIRequiredDeviceCapabilities" in *Information Property List Key Reference*.

If the features of your app that use gyroscope data are not integral to the user experience, you might want to allow users with non-gyroscope devices to download your app. If you do not make gyroscope a required hardware capability, but still have code that requests gyroscope data, you need to check whether the gyroscope is available at runtime. You do this with the `gyroAvailable` property of the `CMMotionManager` class.

Capturing Device Movement with Core Motion

The Core Motion framework is primarily responsible for accessing raw accelerometer and gyroscope data and passing that data to an app for handling. Core Motion uses unique algorithms to process the raw data it collects, so that it can present more refined information. This processing occurs on the framework's own thread.

Core Motion is distinct from UIKit. It is not connected with the `UIEvent` model and does not use the responder chain. Instead, Core Motion simply delivers motion events directly to apps that request them.

Core Motion events are represented by three data objects, each encapsulating one or more measurements:

- A `CMAccelerometerData` object captures the acceleration along each of the spatial axes.
- A `CMGyroData` object 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.

The `CMMotionManager` class is the central access point for Core Motion. You create an instance of the class, specify an update interval, request that updates start, and handle motion events as they are delivered. An app should create only a single instance of the `CMMotionManager` class. Multiple instances of this class can affect the rate at which an app receives data from the accelerometer and gyroscope.

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 a time and logged to a file. An app can 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, the `CMMotionManager` class offers two approaches for obtaining motion data:

- **Pull.** An app requests that updates start and then periodically samples the most recent measurement of motion data.
- **Push.** An app specifies an update interval and implements a block for handling the data. Then, it requests that updates start, and passes Core Motion an operation queue and the block. Core Motion delivers each update to the block, which executes as a task in the operation queue.

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

> **Important:** With Core Motion, you have to test and debug your app on a device. There is no support in iOS Simulator for accelerometer or gyroscope data.

Always stop motion updates as soon as your app finishes processing the necessary data. As a result, Core Motion can turn off motion sensors, which saves battery power.

Choosing a Motion Event Update Interval

When you request motion data with Core Motion, you specify an update interval. You should choose the largest interval that meets your app's needs. The larger the interval, the fewer events are delivered to your app, which improves battery life. Table 4-1 lists some common update frequencies and explains what you can do with data generated at that frequency. Few apps need acceleration events delivered 100 times a second.

Table 4-1 Common update intervals for acceleration events

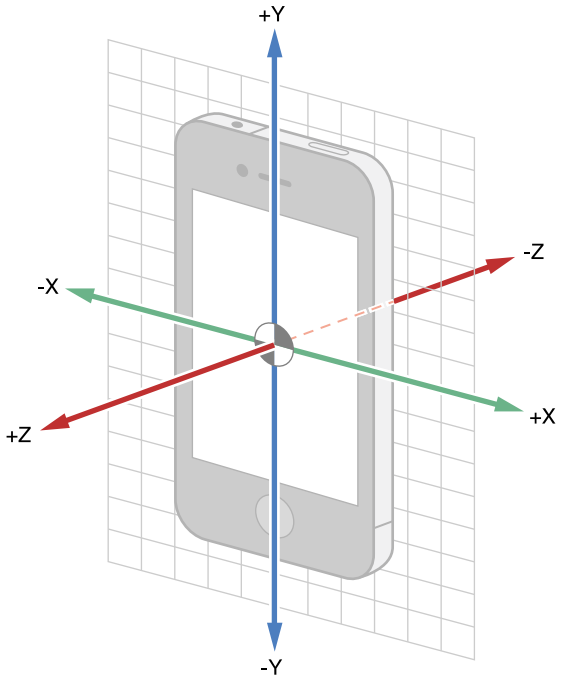
Event frequency (Hz)	Usage
10–20	Suitable for determining a device's current orientation vector.
30–60	Suitable for games and other apps that use the accelerometer for real-time user input.
70–100	Suitable for apps that need to detect high-frequency motion. For example, you might use this interval to detect the user hitting the device or shaking it very quickly.

You can set the reporting interval to be as small as 10 milliseconds (ms), which corresponds to a 100 Hz update rate, but most app operate sufficiently with a larger interval.

Handling Accelerometer Events Using Core Motion

The accelerometer measures velocity over time along three axes, as shown in Figure 4-1. With Core Motion, each movement is captured in a `CMAccelerometerData` object, which encapsulates a structure of type `CMAcceleration`.

Figure 4-1 The accelerometer measures velocity along the x, y, and z axes



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

- `startAccelerometerUpdates`—the pull approach

  After you call this method, Core Motion continually updates the `accelerometerData` property of `CMMotionManager` with the latest measurement of accelerometer activity. Then, you periodically sample this property, usually in a render loop that is common in games. If you adopt this polling approach, set the update-interval property (`accelerometerUpdateInterval`) to the maximum interval at which Core Motion performs updates.

- `startAccelerometerUpdatesToQueue:withHandler:`—the push approach

  Before you call this method, assign an update interval to the `accelerometerUpdateInterval` property, create an instance of `NSOperationQueue` and implement a block of type `CMAccelerometerHandler` that handles the accelerometer updates. Then, call the `startAccelerometerUpdatesToQueue:withHandler:` method 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.

Listing 4-4 (page 62) illustrates this approach.

Listing 4-4 is extracted from the *MotionGraphs* sample code project, which you can examine for more context. In this app, the user moves a slider to specify an update interval. The `startUpdatesWithSliderValue:` method uses the slider value to compute the new update interval. Then, it creates an instance of the `CMMotionManager` class, checks to make sure that the device has an accelerometer, and assigns the update interval to the motion manager. This app uses the push approach to retrieve accelerometer data and plot it on a graph. Note that it stops accelerometer updates in the `stopUpdates` method.

Listing 4-4 Accessing accelerometer data in MotionGraphs

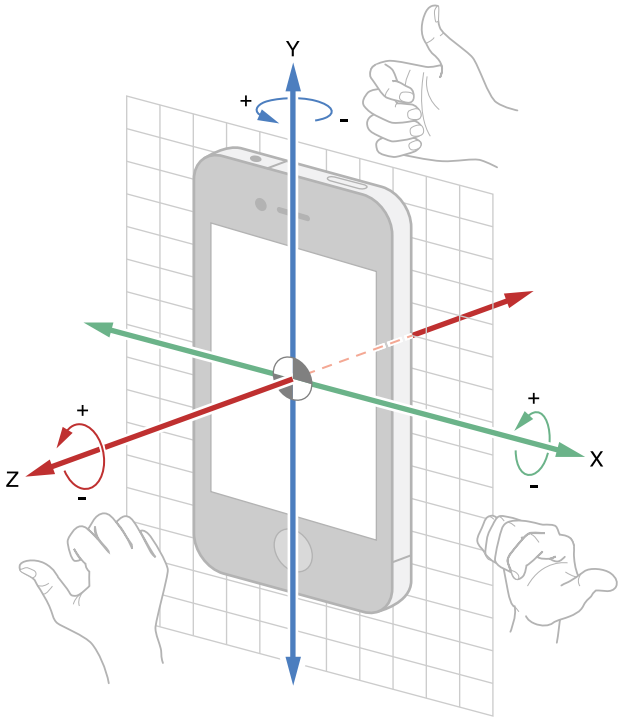
```
static const NSTimeInterval accelerometerMin = 0.01;


- (void)startUpdatesWithSliderValue:(int)sliderValue {


     // Determine the update interval
     NSTimeInterval delta = 0.005;
     NSTimeInterval updateInterval = accelerometerMin + delta * sliderValue;


     // Create a CMMotionManager
     CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];
     APLAccelerometerGraphViewController * __weak weakSelf = self;


     // Check whether the accelerometer is available
     if ([mManager isAccelerometerAvailable] == YES) {
          // Assign the update interval to the motion manager
          [mManager setAccelerometerUpdateInterval:updateInterval];
          [mManager startAccelerometerUpdatesToQueue:[NSOperationQueue mainQueue]
 withHandler:^(CMAccelerometerData *accelerometerData, NSError *error) {
               [weakSelf.graphView addX:accelerometerData.acceleration.x
y:accelerometerData.acceleration.y z:accelerometerData.acceleration.z];
               [weakSelf setLabelValueX:accelerometerData.acceleration.x
y:accelerometerData.acceleration.y z:accelerometerData.acceleration.z];
          }];
     }
```

```
    self.updateIntervalLabel.text = [NSString stringWithFormat:@"%f",
updateInterval];
}



- (void)stopUpdates {
    CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];
    if ([mManager isAccelerometerActive] == YES) {
        [mManager stopAccelerometerUpdates];
    }
}
```

Handling Rotation Rate Data

A gyroscope measures the rate at which a device rotates around each of the three spatial axes, as shown in Figure 4-2.

Figure 4-2 The gyroscope measures rotation around the x, y, and z axes



Each time you request a gyroscope update, Core Motion takes a biased estimate of the rate of rotation and returns this information in a `CMGyroData` object. `CMGyroData` has a `rotationRate` property that stores a `CMRotationRate` structure, which captures the rotation rate for each of the three axes in radians per second. Note that the rotation rate measured by a `CMGyroData` object is biased. You can get a much more accurate, unbiased measurement by using the `CMDeviceMotion` class. See "Handling Processed Device Motion Data" (page 65) for more information.

When analyzing rotation-rate data—specifically, when analyzing the fields of the `CMRotationMatrix` structure—follow the "right-hand rule" to determine the direction of rotation, as shown in Figure 4-2. 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 four fingers. A negative rotation goes away from the tips of those fingers.

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

- `startGyroUpdates`—the pull approach

  After you call this method, Core Motion continually updates the `gyroData` property of `CMMotionManager` with the latest measurement of gyroscope activity. Then, you periodically sample this property. If you adopt this polling approach, set the update-interval property (`gyroUpdateInterval`) to the maximum interval at which Core Motion performs updates.

- `startGyroUpdatesToQueue:withHandler:`—the push approach

  Before you call this method, assign an update interval to the `gyroUpdateInterval` property, create an instance of `NSOperationQueue`, and implement a block of type `CMGyroHandler` that handles the gyroscope updates. Then, call the `startGyroUpdatesToQueue:withHandler:` method 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.

  Listing 4-5 demonstrates this approach.

Listing 4-5 is also extracted from the *MotionGraphs* sample code project, and is nearly identical to Listing 4-4. The app uses the push approach to retrieve gyroscope data so that it can plot the data onscreen.

Listing 4-5 Accessing gyroscope data in MotionGraphs

```
static const NSTimeInterval gyroMin = 0.01;


- (void)startUpdatesWithSliderValue:(int)sliderValue {
```

```
    // Determine the update interval
    NSTimeInterval delta = 0.005;
    NSTimeInterval updateInterval = gyroMin + delta * sliderValue;

    // Create a CMMotionManager
    CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];
    APLGyroGraphViewController * __weak weakSelf = self;

    // Check whether the gyroscope is available
    if ([mManager isGyroAvailable] == YES) {
        // Assign the update interval to the motion manager
        [mManager setGyroUpdateInterval:updateInterval];
        [mManager startGyroUpdatesToQueue:[NSOperationQueue mainQueue]
withHandler:^(CMGyroData *gyroData, NSError *error) {
            [weakSelf.graphView addX:gyroData.rotationRate.x
y:gyroData.rotationRate.y z:gyroData.rotationRate.z];
            [weakSelf setLabelValueX:gyroData.rotationRate.x
y:gyroData.rotationRate.y z:gyroData.rotationRate.z];
        }];
    }
    self.updateIntervalLabel.text = [NSString stringWithFormat:@"%f",
updateInterval];
}

- (void)stopUpdates{
    CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];
    if ([mManager isGyroActive] == YES) {
        [mManager stopGyroUpdates];
    }
}
```

Handling Processed Device Motion Data

If a device has both an accelerometer and a gyroscope, Core Motion offers a device-motion service that processes raw motion data from both sensors. Device motion uses sensor fusion algorithms to refine the raw data and generate information for a device's attitude, its unbiased rotation rate, the direction of gravity on a device,

and the user-generated acceleration. An instance of the `CMDeviceMotion` class encapsulates all of this data. Additionally, you do not need to filter the acceleration data because device-motion separates gravity and user acceleration.

You can access attitude data through a `CMDeviceMotion` object's `attitude` property, which encapsulates a `CMAttitude` object. Each instance of the `CMAttitude` class encapsulates three mathematical representations of attitude:

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

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`—the pull approach

  After you call this method, 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. Then, you periodically sample this property. If you adopt this polling approach, set the update-interval property (`deviceMotionUpdateInterval`) to the maximum interval at which Core Motion performs updates.

  Listing 4-6 illustrates this approach.

- `startDeviceMotionUpdatesToQueue:withHandler:`—the push approach

  Before you call this method, assign an update interval to the `deviceMotionUpdateInterval` property, create an instance of `NSOperationQueue`, and implement a block of the `CMDeviceMotionHandler` type that handles the accelerometer updates. Then, call the `startDeviceMotionUpdatesToQueue:withHandler:` method 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.

Listing 4-6 uses code from the *pARk* sample code project to demonstrate how to start and stop device motion updates. The `startDeviceMotion` method uses the pull approach to start device updates with a reference frame. See "Device Attitude and the Reference Frame" (page 67) for more about device motion reference frames.

Listing 4-6 Starting and stopping device motion updates

```
- (void)startDeviceMotion {
     // Create a CMMotionManager
     motionManager = [[CMMotionManager alloc] init];

     // Tell CoreMotion to show the compass calibration HUD when required
     // to provide true north-referenced attitude
     motionManager.showsDeviceMovementDisplay = YES;
     motionManager.deviceMotionUpdateInterval = 1.0 / 60.0;

     // Attitude that is referenced to true north
     [motionManager
startDeviceMotionUpdatesUsingReferenceFrame:CMAttitudeReferenceFrameXTrueNorthZVertical];
}



- (void)stopDeviceMotion {
     [motionManager stopDeviceMotionUpdates];
}
```

Device Attitude and the Reference Frame

A `CMDeviceMotion` object contains information about a device's attitude, or orientation in space. Device attitude is always measured in relation to a reference frame. Core Motion establishes the reference frame when your app starts device-motion updates. Then, `CMAttitude` gives the rotation from that initial reference frame to the device's current reference frame.

In the Core Motion reference frame, the z-axis is always vertical, and the x- and y-axis are always orthogonal to gravity, which makes the gravity vector [0, 0, -1]. This is also known as 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:

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

You can change the reference frame that `CMAttitude` uses. To do that, cache the attitude object that contains the reference frame and pass it as the argument to `multiplyByInverseOfAttitude:`. The attitude argument receiving the message changes so that it represents the change in attitude from the passed-in reference frame.

Most apps are interested in the change in device attitude. 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 is rendered based on how the device's attitude changed from the start of a pitch. Listing 4-7 illustrates how you might do this.

Listing 4-7 Getting the change in attitude prior to rendering

```
-(void) startPitch {
    // referenceAttitude is a property
    self.referenceAttitude = self.motionManager.deviceMotion.attitude;
}

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

In this example, after the `multiplyByInverseOfAttitude:` method returns, `currentAttitude` represents the change in attitude from `referenceAttitude` to the most recently sampled `CMAttitude` instance.

Remote Control Events

Remote control events let users control an app's multimedia. If your app plays audio or video content, you might want it to respond to remote control events that originate from either transport controls or external accessories. (External accessories must conform to Apple-provided specifications.) iOS converts commands into `UIEvent` objects and delivers the events to an app. The app sends them to the first responder and, if the first responder doesn't handle them, they travel up the responder chain. For more information about the responder chain, see "The Responder Chain Follows a Specific Delivery Path" (page 34).

This chapter describes how to receive and handle remote control events. The code examples are taken from the *Audio Mixer (MixerHost)* sample code project.

Preparing Your App for Remote Control Events

To receive remote control events, your app must do three things:

- **Be the first responder.** The view or view controller that presents the multimedia content must be the first responder.
- **Turn on the delivery of remote control events.** Your app must explicitly request to begin receiving remote control events.
- **Begin playing audio.** Your app must be the "Now Playing" app. Restated, even if your app is the first responder and you have turned on event delivery, your app does not receive remote control events until it begins playing audio.

To make itself capable of becoming first responder, the view or view controller should override the `canBecomeFirstResponder` method of the `UIResponder` class to return `YES`. It should also send itself the `becomeFirstResponder` method at an appropriate time. For example, a view controller might use the `becomeFirstResponder` method in an override of the `viewDidAppear:` method, as in Listing 5-1. This example also shows the view controller "turning on" the delivery of remote control events by calling the `beginReceivingRemoteControlEvents` method of `UIApplication`.

Listing 5-1 Preparing to receive remote control events

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
```

```
    // Turn on remote control event delivery
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];

    // Set itself as the first responder
    [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. It should also resign first-responder status in the `viewWillDisappear:` method, as shown in Listing 5-2.

Listing 5-2 Ending the receipt of remote control events

```
- (void)viewWillDisappear:(BOOL)animated {

    // Turn off remote control event delivery
    [[UIApplication sharedApplication] endReceivingRemoteControlEvents];

    // Resign as first responder
    [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 then, based on the subtype, send the appropriate message to the object that presents the audio or video content. Listing 5-3 sends play, pause, and stop messages to an audio object. Other remote control `UIEvent` subtypes are possible, see *UIEvent Class Reference* for details.

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;
        }
    }
}
```

Testing Remote Control Events on a Device

Test that your app is properly receiving and handling remote control events with the Now Playing Controls. These controls are available on recent iOS devices that are running iOS 4.0 or later. To access these controls, press the Home button twice, then flick right along the bottom of the screen until you find the audio playback controls. These controls send remote control events to the app that is currently or was most recently playing audio. The icon to the right of the playback controls represents the app that is currently receiving the remote control events.

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

Document Revision History

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

Date	Notes
2013-01-28	Reordered the chapters to present gesture recognizers first and conceptual information about event delivery and the responder chain later. Also updated content to include the most recent information for iOS 6.0.
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