Handling Touches in Your View

On This Page

The ability to handle multiple touches is an important feature of UIKit apps, allowing you to use multiple fingers as input to a device. Although most user interactions involve only a single finger touching the screen, the ability to handle multiple touches gives you the potential to support advanced user interactions and gestures.

If you build your apps using standard UIKit views and controls, UIKit handles touch events (including multitouch events) for you automatically. However, if you use custom views to display your content, you must handle any touch events occurring in your views. There are two ways to handle touch events yourself:

- Use gesture recognizers to track the touches; see Gesture Recognizer Basics.
- Track the touches directly in your `UIView` subclass.

Gesture recognizer offer the most flexible approach to handling events, because the same gesture recognizer class can be reused with multiple views. However, if your view's state is intricately tied to its touch input, you can incorporate your touch-handling code directly into your view.

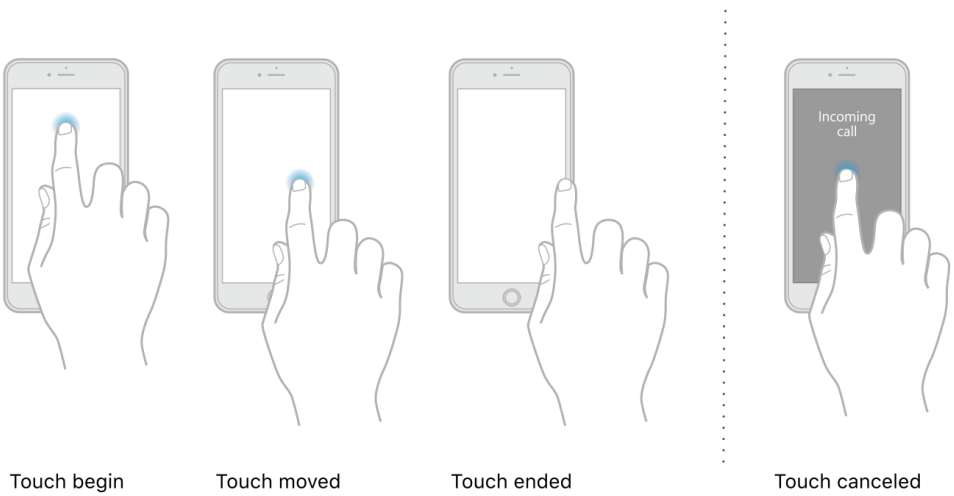
Tracking Touches in Your View

If you do not plan to use gesture recognizers with a custom view, you can handle touch events directly from the view itself. Because views are responders, they can handle multitouch events and many other types of events. When UIKit determines that a touch event occurred in a view, it calls the view's `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, or `touchesEnded:withEvent:` method. You can override those methods in your custom views and use them to provide a response to touch events.

The system may cancel an ongoing touch sequence at any time—for example, when an incoming phone call interrupts the app. When it does, UIKit notifies your view by calling the `touchesCancelled:withEvent:` method. You use that method to perform any needed cleanup of your view's data structures.

The methods you override in your views (or in any responder) to handle touches correspond to different phases of the touch event-handling process. When a finger (or Apple Pencil) touches the screen, UIKit creates a `UITouch` object, sets the touch location to the appropriate point, and sets its `phase` property to `UITouchPhaseBegan`. When the same finger moves around the screen, UIKit updates the touch location and changes the `phase` property of the touch object to `UITouchPhaseMoved`. When the user lifts the finger from the screen, UIKit changes the `phase` property to `UITouchPhaseEnded` and the touch sequence ends. Figure 9-1 illustrates the different phases of a touch event.

Figure 9-1 The phases of a touch event



UIKit creates a new `UITouch` object for each new finger that touches the screen. The touches themselves are delivered with the current `UIEvent` object. UIKit distinguishes between touches originating from a finger and from Apple Pencil, and you can treat each of them differently.

> IMPORTANT
>
> In its default configuration, a view receives only the first `UITouch` object associated with an event, even if more than one finger is touching the view. To receive the additional touches, you must set the view's

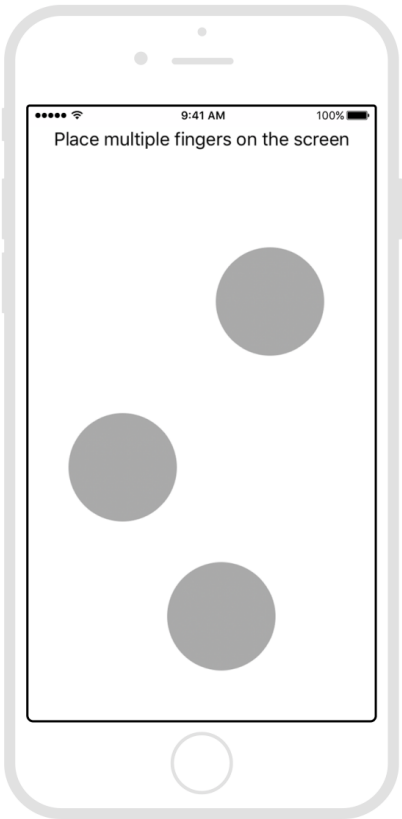
> `multipleTouchEnabled` property to `YES`. You can also configure this property in Interface Builder using the Attributes inspector.

On This Page

Handling Multitouch Input: An Example

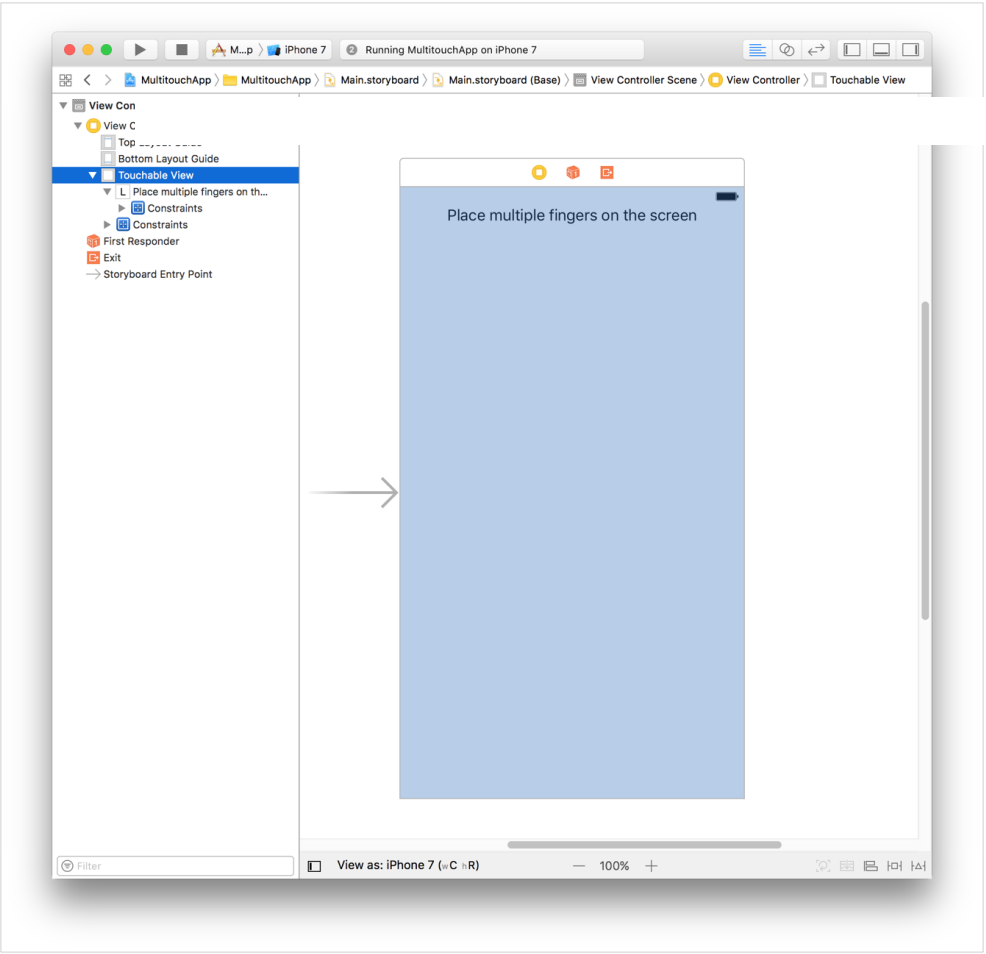
To understand how you handle touch input, consider the app shown in Figure 9-2. The app contains a single main view that draws a gray circle at each touch location. When a touch ends, the circle disappears. When the user's fingers move, the underlying circles move with them.

Figure 9-2 Handling multiple touches in a view



To create this multitouch app, start with the Single View app template in Xcode. This type of app has a single view controller whose view fills the screen. In this case, the view controller's view is a custom subclass of `UIView` called `TouchableView`. The view contains only a label initially, but more subviews are added programmatically later. Figure 9-3 shows the storyboard for the view controller.

Figure 9-3 The app's main view



On This Page

The `TouchableView` class overrides the inherited `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:` methods. These methods handle the creation and management of subviews that draw the gray circles at each touch location. Specifically, these methods do the following:

- The `touchesBegan:withEvent:` method creates a new subview at the location of each touch event.
- The `touchesMoved:withEvent:` method updates the position of the subview associated with each touch.
- The `touchesEnded:withEvent:` and `touchesCancelled:withEvent:` methods remove the subview associated with each touch that ended.

Listing 9-1 shows the main implementation of the `TouchableView` class and its touch handling methods. Each method iterates through the touches and performs the needed actions. The `touchViews` dictionary uses the `UITouch` objects as keys to retrieve the subviews being manipulated onscreen.

Listing 9-1 Handling touch events

```
class TouchableView: UIView {
    var touchViews = [UITouch:TouchSpotView]()

    override init(frame: CGRect) {
        super.init(frame: frame)
        isMultipleTouchEnabled = true
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        isMultipleTouchEnabled = true
    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            createViewForTouch(touch: touch)
        }
```

```
    }

    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {

            let view = viewForTouch(touch: touch)

            // Move the view to the new location.
            let newLocation = touch.location(in: self)
            view?.center = newLocation
        }
    }

    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            removeViewForTouch(touch: touch)
        }
    }

    override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            removeViewForTouch(touch: touch)
        }
    }

    // Other methods. . .
}
```

On This Page

Creation, management, and disposal of the subviews is handled by several helper methods, which are shown in Listing 9-2. The `createViewForTouch` method creates a new `TouchSpotView` object and adds it to the `TouchableView` object, animating the view to its full size. The `removeViewForTouch` method removes the corresponding subview and updates the class data structures. The `viewForTouch` method is a convenience method for retrieving the view associated with a given touch event.

Listing 9-2 Managing subviews

```
func createViewForTouch( touch : UITouch ) {
    let newView = TouchSpotView()
    newView.bounds = CGRect(x: 0, y: 0, width: 1, height: 1)
    newView.center = touch.location(in: self)

    // Add the view and animate it to a new size.
    addSubview(newView)
    UIView.animate(withDuration: 0.2) {
        newView.bounds.size = CGSize(width: 100, height: 100)
    }

    // Save the views internally
    touchViews[touch] = newView
}

func viewForTouch (touch : UITouch) -> TouchSpotView? {
    return touchViews[touch]
}

func removeViewForTouch (touch : UITouch ) {
    if let view = touchViews[touch] {
        view.removeFromSuperview()
        touchViews.removeValue(forKey: touch)
    }
}
```

The `TouchSpotView` class (shown in Listing 9-3) represents the custom subviews that draw the gray circles onscreen. The circular shape is created using the `cornerRadius` property of the layer, which is updated

whenever the `bounds` property of the view changes.

Listing 9-3 Implementation of the TouchSpotView class

```
class T
    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = UIColor.lightGray
    }

    // Update the corner radius when the bounds change.
    override var bounds: CGRect {
        get { return super.bounds }
        set(newBounds) {
            super.bounds = newBounds
            layer.cornerRadius = newBounds.size.width / 2.0
        }
    }
}
```

On This Page