Implementing a Custom Gesture Recognizer

On This Page

When the built-in UIKit gesture recognizers do not provide the behavior you want, you can define custom gesture recognizers. UIKit defines highly configurable gesture recognizers to handle touch sequences for taps, long presses, pans, swipes, rotations, and pinches. For other touch sequences, or to handle gestures that involve button presses, you can define a custom gesture recognizer. You might also use a custom gesture recognizer to simplify the event-handling code in your app. For example, the *SpeedSketch: Leveraging touch input for a drawing application* sample uses a gesture recognizer to capture input and display it onscreen, as shown in Figure 5-1.

Figure 5-1 Touch input captured by a custom gesture recognizer

To define a custom gesture recognizer, subclass `UIGestureRecognizer` (or one of its subclasses). At the top of your source file, import the `UIGestureRecognizerSubclass.h` header file, as shown in Listing 5-1. This header file defines the methods and properties that you must override to implement your custom gesture recognizer.

Listing 5-1 Importing the UIGestureRecognizerSubclass behavior

OBJECTIVE-C

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

SWIFT

```
import UIKit
import UIKit.UIGestureRecognizerSubclass
```

In your custom subclass, implement whatever methods you need to process events. For example, if your gesture consists of touch events, implement the `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:` methods. Use incoming events to update the `state` property of your gesture recognizer. UIKit uses the gesture recognizer states to coordinate interactions with other objects in your interface.

Understanding the Gesture Recognizer State Machine

When implementing a custom gesture recognizer, you must maintain its state machine. UIKit uses this state machine to ensure that your gesture recognizer interacts properly with other objects. For example, UIKit normally prevents two gestures attached to the same view from reaching the recognized state at the same time. UIKit also uses the state of your gesture recognizer to determine when to notify any associated target objects.

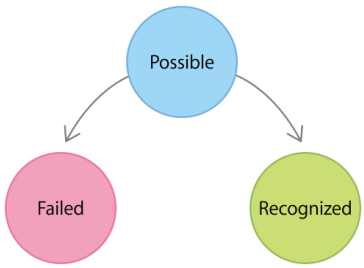
On This Page

A gesture recognizer always begins in the *possible* state, which indicates that it is ready to begin processing events. From the *possible* state, discrete and continuous gesture recognizers follow different paths until they reach the *recognized*, *failed*, or *cancelled* state. A gesture recognizer remains in one of those end states until the current event sequence ends, at which point UIKit resets the gesture recognizer and returns it to the *possible* state.

Managing State Transitions for a Discrete Gesture Recognizer

When implementing a discrete gesture recognizer, you can change the `state` property to one of two possible values: `UIGestureRecognizerStateRecognized` or `UIGestureRecognizerStateFailed`. Figure 5-2 shows the state diagram for these transitions. If incoming events successfully match your gesture, change the state to `UIGestureRecognizerStateRecognized`. However, if events do not match your intended gesture, change the state to `UIGestureRecognizerStateFailed` as soon as you detect the failure.

Figure 5-2 The states of a discrete gesture



When your gesture recognizer transitions to the `UIGestureRecognizerStateRecognized` state, UIKit calls the action methods of any associated target objects. UIKit does not call any action methods when the gesture recognizer transitions to the `UIGestureRecognizerStateFailed` state.

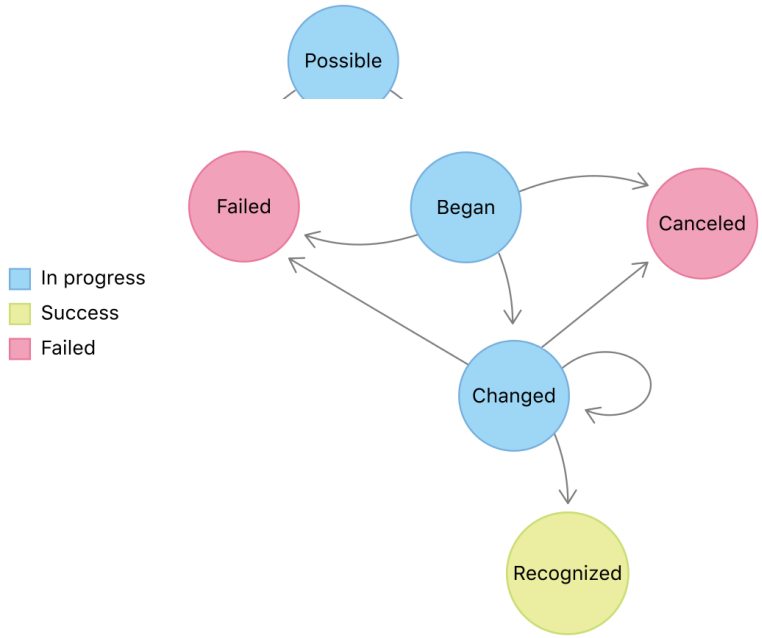
For an example of how to implement a discrete gesture recognizer, see Implementing a Discrete Gesture Recognizer.

Managing State Transitions for a Continuous Gesture Recognizer

When implementing a continuous gesture recognizer, you must handle many more state transitions. Figure 5-3 shows the state diagram for a continuous gesture recognizer. The state transitions you make can be broken down into three general phases:

1. An initial event moves the gesture recognizer to the Began state.
2. Subsequent events move the gesture recognizer to the Changed state.
3. A final event ends the gesture.

Figure 5-3 The states of a continuous gesture



When a new event sequence starts, move your gesture recognizer to the `UIGestureRecognizerStateBegan` state. You might also use the beginning of the gesture to cache any information you need to track the gesture. For any subsequent events, move your gesture recognizer to the `UIGestureRecognizerStateChanged` state and update any cached values. Keep your gesture recognizer in the `UIGestureRecognizerStateChanged` state until an event indicates the success or failure of the gesture, at which point you move the gesture recognizer to the appropriate state. For example, you might set the state to `UIGestureRecognizerStateRecognized` in your gesture recognizer's `touchesEnded:withEvent:` method to signify that the user finished the gesture successfully.

When your gesture recognizer transitions to the `UIGestureRecognizerStateBegan`, `UIGestureRecognizerStateChanged`, or `UIGestureRecognizerStateRecognized` states, UIKit calls the action methods of any associated targets. UIKit does not call any action methods when your gesture recognizer transitions to other states.

When events indicate a failed gesture, move the gesture recognizer to the `UIGestureRecognizerStateFailed` state immediately. UIKit normally permits only one gesture recognizer at a time to notify its client. Moving your custom gesture recognizer to the failed state gives other gesture recognizers an opportunity to notify their clients.

Handling Cancellation

Cancellation of a gesture occurs when an event sequence is interrupted by another system event, such as an incoming phone call. Gesture cancellation prevents your app from performing actions that the user may not have intended.

When the system cancels a gesture, UIKit calls the `touchesCancelled:withEvent:` or `pressesCancelled:withEvent:` method of your gesture recognizer. When that happens, move your gesture recognizer to the `UIGestureRecognizerStateCancelled` state immediately and perform any needed cleanup. When you move your gesture recognizer to that state, UIKit stops calling the gesture recognizer's associated action methods.

Resetting the Gesture Recognizer State Machine

Implement the `reset` method and use it to return your gesture recognizer to its initial configuration. Before delivering events in a new event sequence, UIKit calls the `reset` method of every gesture recognizer that received touches or is in the `UIGestureRecognizerStateFailed`, `UIGestureRecognizerStateCancelled`, or `UIGestureRecognizerStateRecognized` state. In addition to calling the `reset` method, UIKit automatically changes each gesture recognizer's `state` property back to `UIGestureRecognizerStatePossible` so that it can respond to new event sequences.

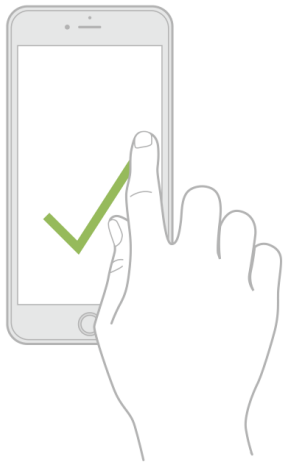
Implementing a Discrete Gesture Recognizer

If your gesture involves a specific pattern of events, consider implementing a discrete gesture recognizer for it. A gesture recognizer remains in the `UIGestureRecognizerStatePossible` state until events indicate that your gesture succeeded or failed, at which point you change its state. The advantage of discrete gesture recognizers is that they are simpler to implement because they require fewer state transitions. One disadvantage is that they can easily be preempted by continuous gestures attached to the same view.

On This Page

Figure 5-4 shows a checkmark gesture, which is created by tracing one finger down and to the right and then back up and to the right. Because the gesture follows a specific path, it makes sense to use a discrete gesture recognizer.

Figure 5-4 A custom checkmark gesture



Defining the Conditions for Success

Before implementing your gesture recognizer code, define the conditions for which recognition should occur. The conditions for matching a checkmark gesture are as follows:

- Only the first finger to touch the screen is tracked. All others are ignored.
- The touch always moves left to right.
- The touch moves downward initially but then changes direction and moves upward.
- The upward stroke ends higher on the screen than the initial touch point.

Saving Gesture-Related Data

With the conditions defined, add properties to your gesture recognizer to track any needed information. For the checkmark gesture, the gesture recognizer needs to know the starting point of the gesture so that it can compare that point to the final point. It also needs to know whether the user's finger is moving downward or upward.

Listing 5-2 shows the first part of a custom `CheckmarkGestureRecognizer` class definition. This class stores the initial touch point and the current phase of the gesture. The class also stores the `UITouch` object associated with the first finger so that it can ignore any other touches.

Listing 5-2 Beginning of the CheckmarkGestureRecognizer class

```
enum CheckmarkPhases {
    case notStarted
    case initialPoint
    case downStroke
    case upStroke
}

class CheckmarkGestureRecognizer : UIGestureRecognizer {
    var strokePhase : CheckmarkPhases = .notStarted
    var initialTouchPoint : CGPoint = CGPoint.zero
    var trackedTouch : UITouch? = nil

```

```
    // Overridden methods to come. . .
}
```

On This Page

Processing Touch Events

Listing 5-3 shows the `touchesBegan:withEvent:` method, which sets up the initial conditions for recognizing the gesture. The gesture fails immediately if the initial event contains two touches. If there is only one touch, the touch object is saved in the `trackedTouch` property. Because UIKit reuses `UITouch` objects, and therefore overwrites their properties, this method also saves the location of the touch in the `initialTouchPoint` property. After the first touch occurs, any new touches added to the event sequence are ignored.

Listing 5-3 Getting the first touch

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent) {
    super.touchesBegan(touches, with: event)
    if touches.count != 1 {
        self.state = .failed
    }

    // Capture the first touch and store some information about it.
    if self.trackedTouch == nil {
        self.trackedTouch = touches.first
        self.strokePhase = .initialPoint
        self.initialTouchPoint = (self.trackedTouch?.location(in: self.view))!
    }
    else {
        // Ignore all but the first touch.
        for touch in touches {
            if touch != self.trackedTouch {
                self.ignore(touch, for: event)
            }
        }
    }
}
```

When touch information changes, UIKit calls the `touchesMoved:withEvent:` method. Listing 5-4 shows the implementation of this method for the checkmark gesture. This method verifies that the first touch is the correct one, which it should be because all subsequent touches were ignored. It then looks at the movement of that touch. When the initial movement is down and to the right, this method sets the `strokePhase` property to `downStroke`. When the motion changes direction and starts moving upward, the method changes the stroke phase to `upStroke`. If the gesture deviates from this pattern in any way, the method sets the gesture's state to failed.

Listing 5-4 Tracking the touch movement

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent) {
    super.touchesMoved(touches, with: event)
    let newTouch = touches.first

    // There should be only the first touch.
    guard newTouch == self.trackedTouch else {
        self.state = .failed
        return
    }

    let newPoint = (newTouch?.location(in: self.view))!
    let previousPoint = (newTouch?.previousLocation(in: self.view))!
    if self.strokePhase == .initialPoint {
        // Make sure the initial movement is down and to the right.
        if newPoint.x >= initialTouchPoint.x && newPoint.y >= initialTouchPoint.y {
            self.strokePhase = .downStroke
        }
        else {
```

```
            self.state = .failed
        }
    }
    els
        // Always keep moving left to right.
        if newPoint.x >= previousPoint.x {
            // If the y direction changes, the gesture is moving up again.
            // Otherwise, the down stroke continues.
            if newPoint.y < previousPoint.y {
                self.strokePhase = .upStroke
            }
        }
        else {
            // If the new x value is to the left, the gesture fails.
            self.state = .failed
        }
    }
    else if self.strokePhase == .upStroke {
        // If the new x value is to the left, or the new y value
        // changed directions again, the gesture fails.
        if newPoint.x < previousPoint.x || newPoint.y > previousPoint.y {
            self.state = .failed
        }
    }
}
```


At the end of the touch sequence, UIKit calls the `touchesEnded:withEvent:` method. Listing 5-5 shows the implementation of this method for the checkmark gesture. If the gesture has not already failed, this method determines whether the gesture was moving upward when it ended and determines whether the final point is higher than the initial point. If both conditions are true, the method sets the state to `UIGestureRecognizerStateRecognized`; otherwise, the gesture fails.

Listing 5-5 Determining whether the gesture succeeded

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent) {
    super.touchesEnded(touches, with: event)

    let newTouch = touches.first
    let newPoint = (newTouch?.location(in: self.view))!
    // There should be only the first touch.
    guard newTouch == self.trackedTouch else {
        self.state = .failed
        return
    }

    // If the stroke was moving up and the final point is
    // above the initial point, the gesture succeeds.
    if self.state == .possible &&
        self.strokePhase == .upStroke &&
        newPoint.y < initialTouchPoint.y {
        self.state = .recognized
    }
    else {
        self.state = .failed
    }
}
```

Resetting the Gesture Recognizer

In addition to tracking the touches, the `CheckmarkGestureRecognizer` class implements the `touchesCancelled:withEvent:` and `reset` methods. The class uses these methods to reset the gesture recognizer's local properties to appropriate values. Listing 5-6 shows the implementations of these methods.

Listing 5-6 Cancelling and resetting the discrete gesture

```
override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent) {
    super.touchesCancelled(touches, with: event)
    sel
    self.strokePhase = .notStarted
    self.trackedTouch = nil
    self.state = .cancelled
}

override func reset() {
    super.reset()
    self.initialTouchPoint = CGPoint.zero
    self.strokePhase = .notStarted
    self.trackedTouch = nil
}
```

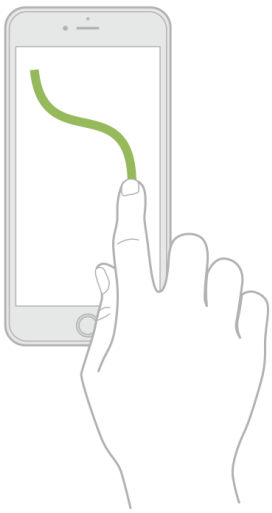
On This Page

Implementing a Continuous Gesture Recognizer

For gestures that do not easily match a specific pattern, or when you want to use a gesture recognizer to gather touch input, create a continuous gesture recognizer. A continuous gesture recognizer lets you encapsulate your event-handling logic in one place and reuse that logic in multiple views. Although continuous gesture recognizers require a little more effort to implement the state machine, they also perform tasks that would be difficult with a discrete gesture recognizer, such as capturing free-form input.

Figure 5-5 shows a free-form gesture whose input you might use to draw paths onscreen. Although you could use a pan gesture recognizer to capture the input, your action method would need to handle all of the phases of the capture process, which would add to its complexity. Using a custom gesture recognizer, you can simplify your code by distributing your logic to various methods of your subclass. Using a custom gesture recognizer also means that you can write your code for capturing the path once and reuse it in multiple views.

Figure 5-5 A free-form gesture



For a custom gesture recognizer that captures touch input, there are no explicit conditions that trigger a failure of the gesture. Instead, the gesture recognizer captures touch input until the touch sequence ends or is cancelled by the system. While the gesture is ongoing, the gesture recognizer places the touch data into a temporary buffer. Clients of the gesture recognizer use their action method to fetch that buffer and apply it temporarily to the app's content. For example, a client might use that data to draw the path onscreen. Only when the touch sequence ends successfully would those target objects commit the data permanently to the app's data structures.

Saving Gesture-Related Data

A continuous gesture recognizer that tracks touch events needs a way to store that information. You cannot simply store references to the `UITouch` objects that you receive because UIKit reuses those objects and

overwrites any old values. Instead, you must define custom data structures to store the touch information you need.

Listing 5-7 shows the definition of a `StrokeSample` struct, whose purpose is to store the location associated with a touch. In
such as the timestamp or the force of the touch.

On This Page

Listing 5-7 Managing the touch data

```
struct StrokeSample {
    let location: CGPoint

    init(location: CGPoint) {
        self.location = location
    }
}
```

Listing 5-8 shows the partial definition of a `TouchCaptureGesture` class used to capture touch information. This class stores touch data in the `samples` property, which is an array of `StrokeSample` structs. The class also stores the `UITouch` object associated with the first finger so that it can ignore any other touches. The implementation of the `initWithCoder:` method ensures that the `samples` property is initialized properly when loading the gesture recognizer from an Interface Builder file.

Listing 5-8 Properties of the TouchCaptureGesture class

```
class TouchCaptureGesture: UIGestureRecognizer, NSCoding {
    var trackedTouch: UITouch? = nil
    var samples = [StrokeSample]()

    required init?(coder aDecoder: NSCoder) {
        super.init(target: nil, action: nil)

        self.samples = [StrokeSample]()
    }

    func encode(with aCoder: NSCoder) { }

    // Overridden methods to come. . .
}
```

Processing Touch Events

Listing 5-9 shows the `touchesBegan:withEvent:` method of the TouchCaptureGesture class. The gesture fails immediately if the initial event contains two touches. If there is only one touch, the touch object is saved in the `trackedTouch` property and the custom `addSample` helper method creates a new `StrokeSample` struct with the touch data. After the first touch occurs, any new touches added to the event sequence are ignored.

Listing 5-9 Beginning the capture of touches

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent) {
    if touches.count != 1 {
        self.state = .failed
    }

    // Capture the first touch and store some information about it.
    if self.trackedTouch == nil {
        if let firstTouch = touches.first {
            self.trackedTouch = firstTouch
            self.addSample(for: firstTouch)
            state = .began
        }
    }
    else {
        // Ignore all but the first touch.
        for touch in touches {
```

```
            if touch != self.trackedTouch {
                self.ignore(touch, for: event)
            }

    }
}

func addSample(for touch: UITouch) {
    let newSample = StrokeSample(location: touch.location(in: self.view))
    self.samples.append(newSample)
}
```


The `touchesMoved:withEvent:` and `touchesEnded:withEvent:` methods (shown in Listing 5-10) record each new sample and update the gesture recognizer's state. Setting the state to `UIGestureRecognizerStateEnded` is equivalent to setting the state to `UIGestureRecognizerStateRecognized` and results in a call to the gesture recognizer's action method.

Listing 5-10 Managing the touch input

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    self.addSample(for: touches.first!)
    state = .changed
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    self.addSample(for: touches.first!)
    state = .ended
}
```

Resetting the Gesture Recognizer

Always implement the `touchesCancelled:withEvent:` and `reset` methods in your gesture recognizers and use them to perform any cleanup. Listing 5-11 shows the implementation of these methods for the `TouchCaptureGesture` class. Both methods restore the gesture recognizer's properties to their initial values.

Listing 5-11 Cancelling and resetting the continuous gesture

```
override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {
    self.samples.removeAll()
    state = .cancelled
}

override func reset() {
    self.samples.removeAll()
    self.trackedTouch = nil
}
```

Copyright © 2017 Apple Inc. All rights reserved. Terms of Use | Privacy Policy | Updated: 2017-03-21