# iPhone Application Programming
# Lecture 8: Event Handling, Gesture Recognizers and Core Motion

Nur Al-huda Hamdan
Media Computing Group
RWTH Aachen University

Winter Semester 2015/2016

http://hci.rwth-aachen.de/iphone

## Learning Objectives

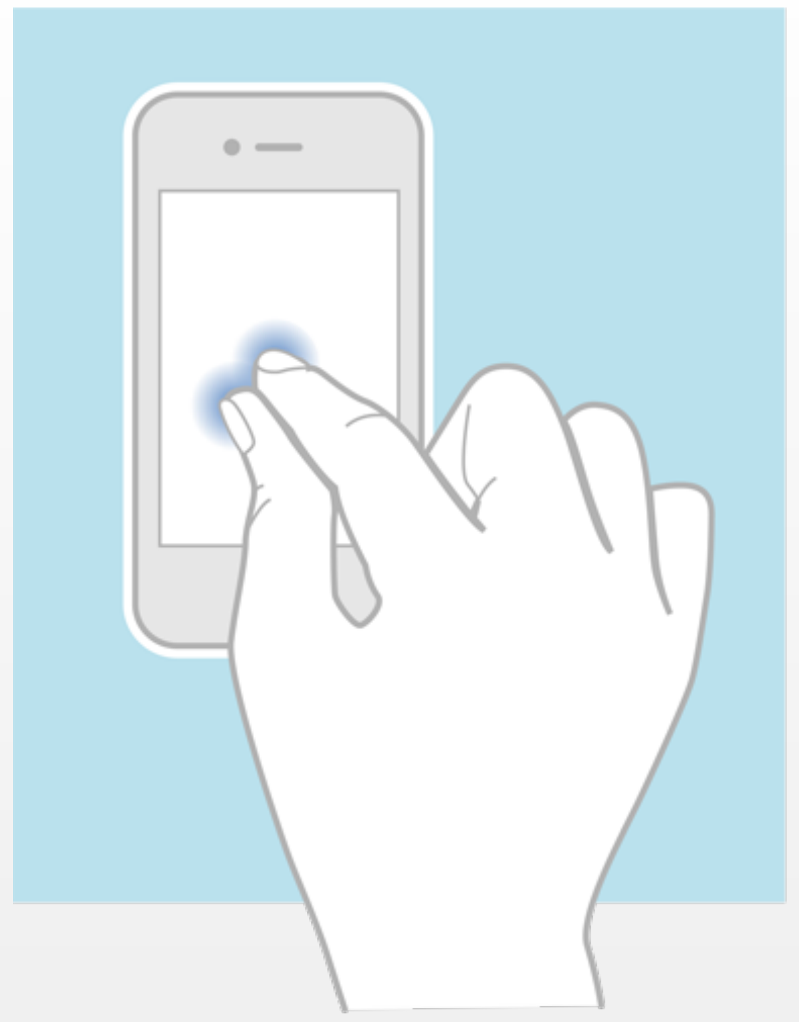
- How users interact with iOS devices: concepts and code snippets
- Touch, Multi-Touch
- Gesture Recognizers
- Core Motion

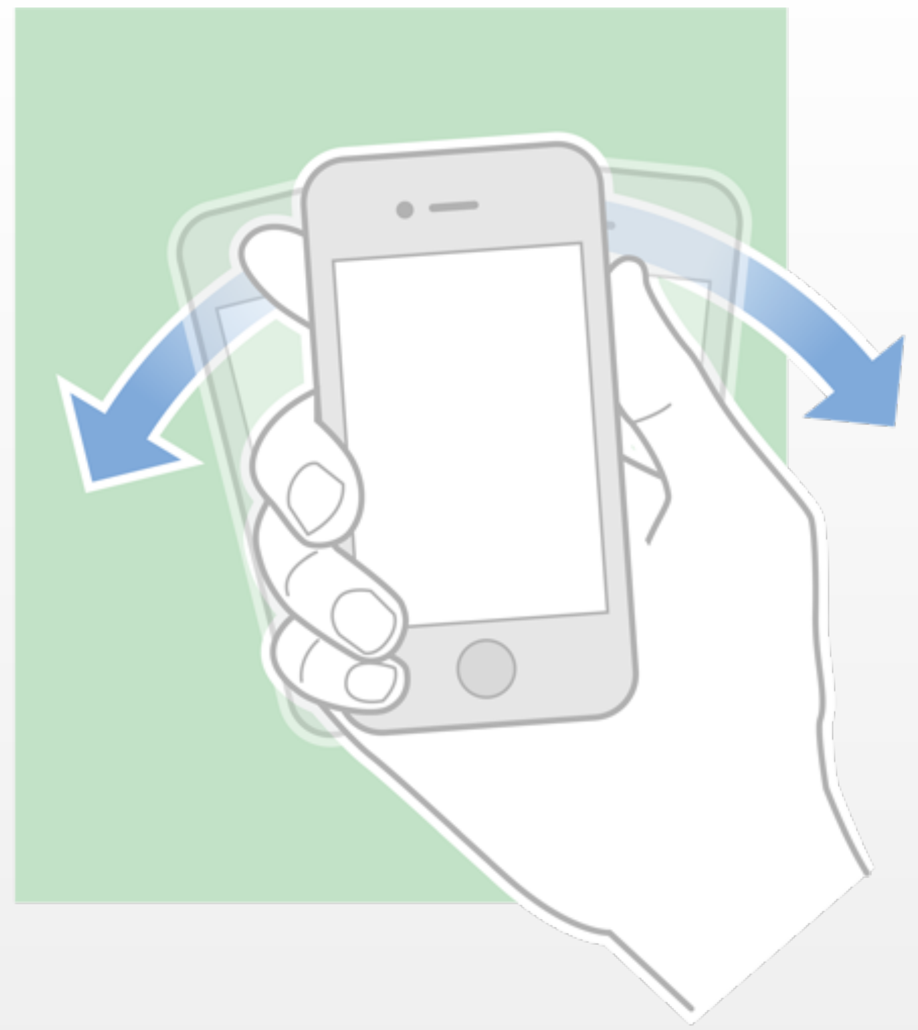
# User-Generated Events in iOS

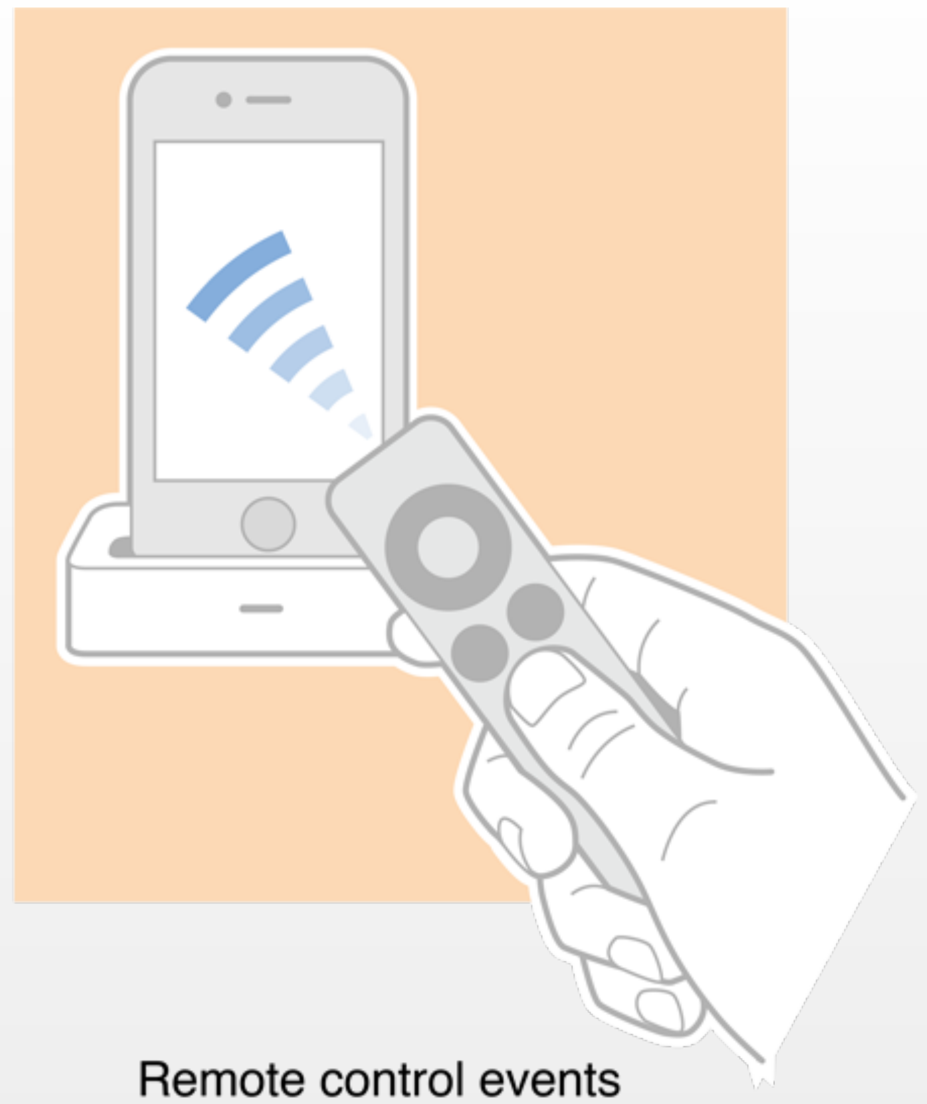
- User actions (interaction) are propagated as events in iOS
- Events types: multi-touch, motion, or remote control events (for controlling multimedia)

```
enum UIEventType : Int {
    case Touches
    case Motion
    case RemoteControl
    case Presses
}
```

```
enum UIEventSubtype : Int {
    case None
    case MotionShake
    case RemoteControlPlay
    case RemoteControlPause
    case RemoteControlStop
    case RemoteControlTogglePlayPause
    case RemoteControlNextTrack
    case RemoteControlPreviousTrack
    case RemoteControlBeginSeekingBackward
    case RemoteControlEndSeekingBackward
    case RemoteControlBeginSeekingForward
    case RemoteControlEndSeekingForward
}
```

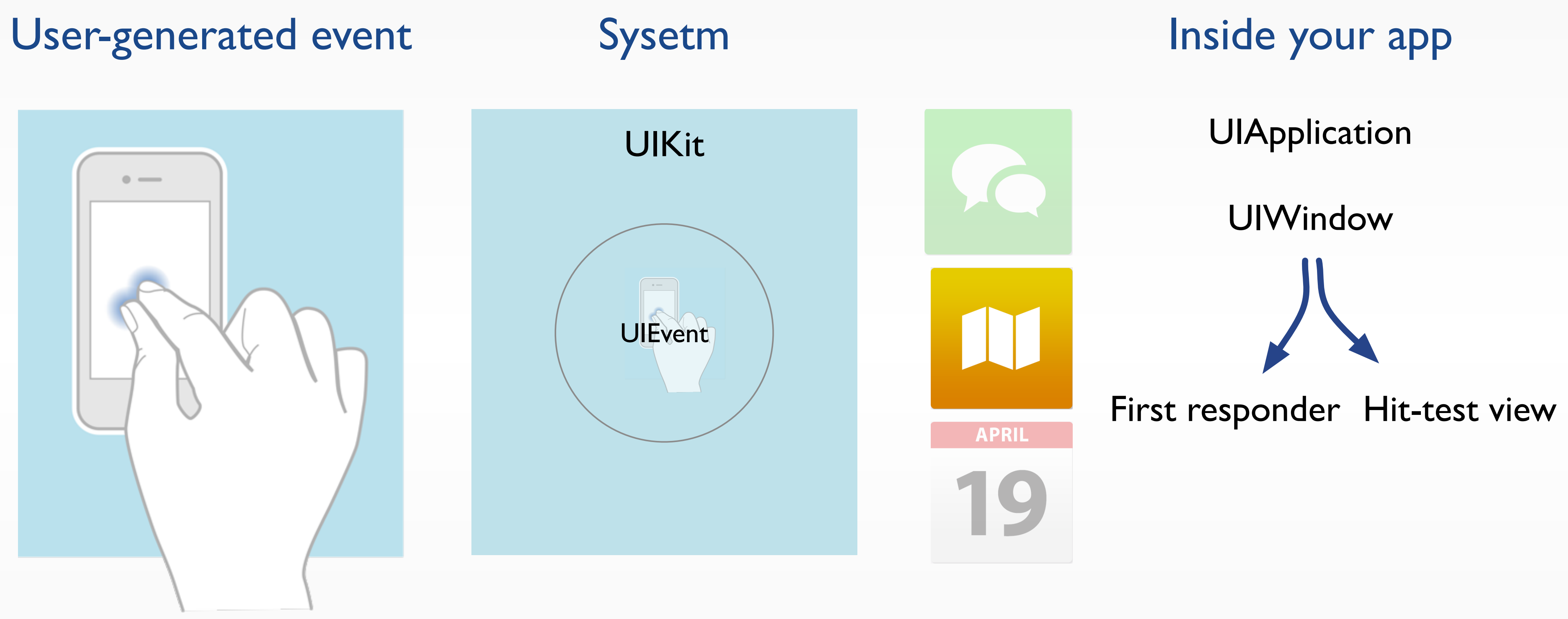

Multitouch events


Accelerometer events


Remote control events

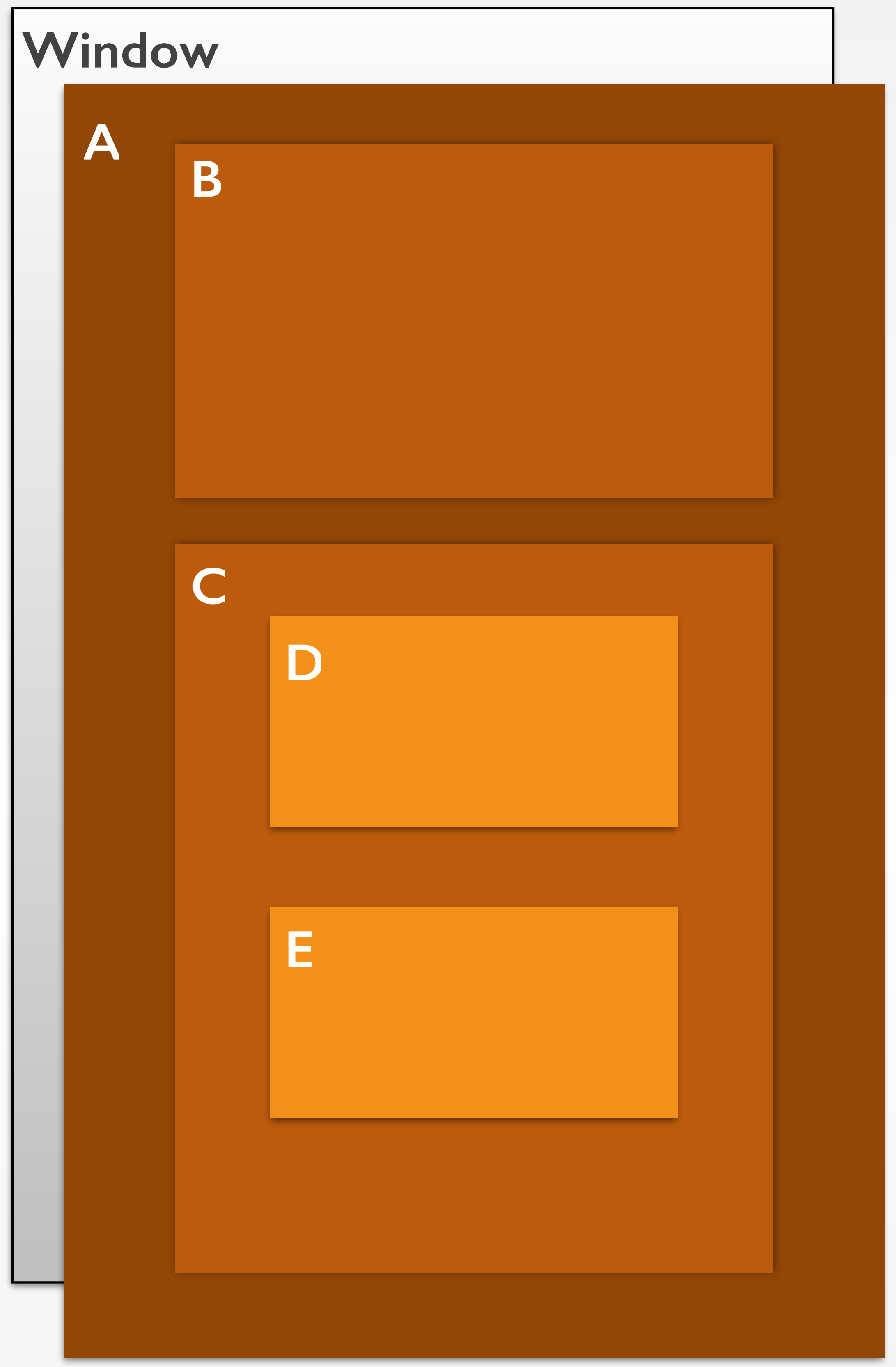
# Event Delivery

- When a user-generated event occurs, UIKit creates an event object (UIEvent)
- The event is sent to the active app
- In an app, the singleton UIApplication object takes the event from the top of the queue and dispatches it for handling
- The event is sent to the app's main window object, which passes the event to an initial object for handling
    - For touch events, the window object first tries to deliver the event to the **hit-test view** where the touch occurred
    - Motion and remote control events are sent to the **first responder** (often a view controller that receives an event first)



# Hit-Testing

- iOS uses hit-testing to find the view that is under a touch
- Hit-testing checks 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 outmost view that contains the touch becomes the hit-test view
- The touch event is then passed for the hit-test view for handling
- 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 until the system finds an object that can handle it



# Hit-Testing

- Two methods in UIView
- No need to implement them, but can override them
- point is in the receiver's local coordinate system (bounds)

```
   override func hitTest(point: CGPoint, withEvent event:
UIEvent?) -> UIView?
    {
        if self.alpha > 0.01
            && self.hidden == false
            && self.userInteractionEnabled == true
            && self.pointInside(point, withEvent: event)
        {
            for subview in self.subviews
            {
                if let hitView = subview.hitTest(point,
withEvent:event)
                {
                    return hitView
                }
                return self
            }

        }
        return nil
    }
```

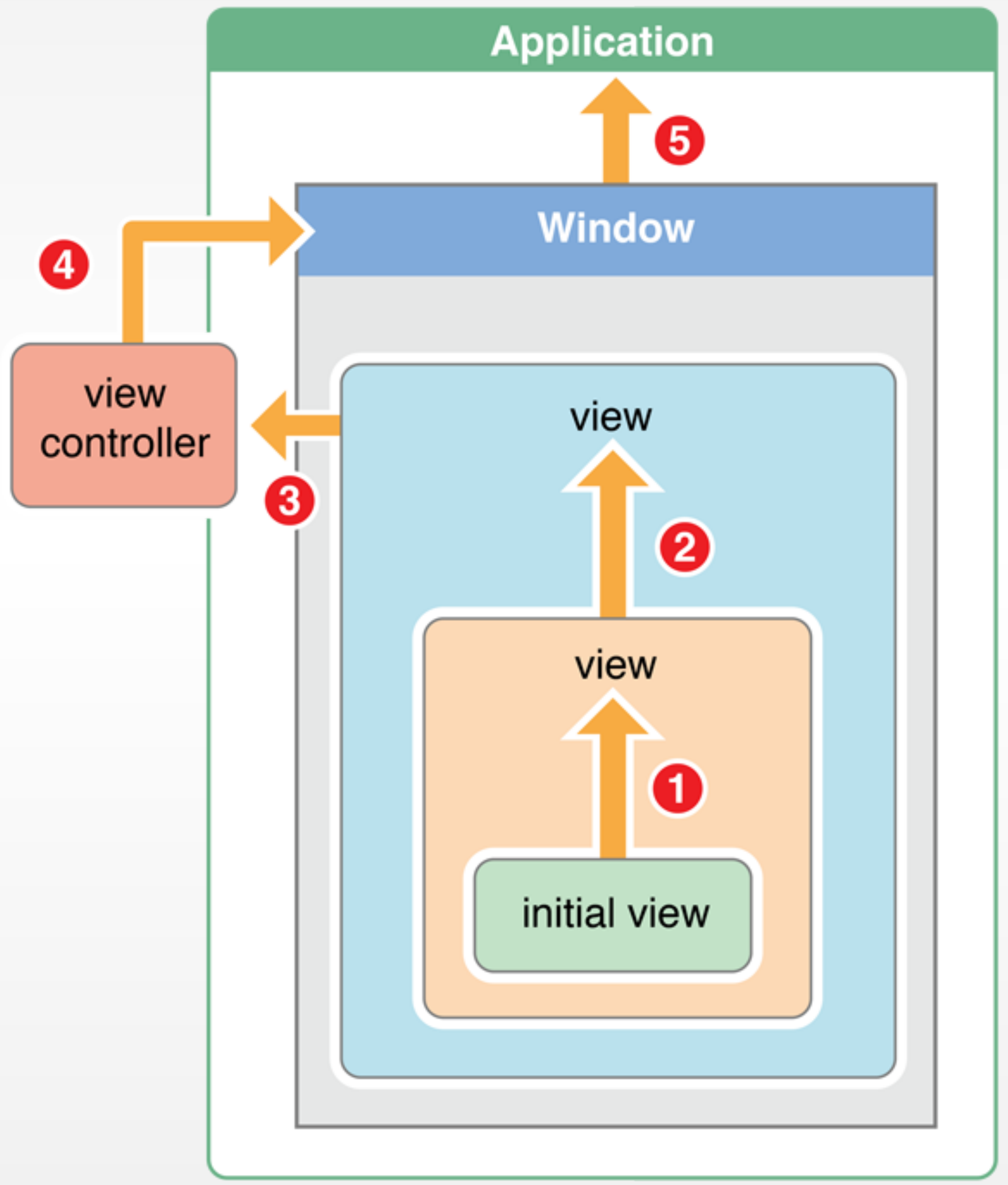
```
   override func pointInside(point: CGPoint, withEvent
event: UIEvent?) -> Bool
    {
        var touchBounds:CGRect = self.bounds
        return CGRectContainsPoint(touchBounds, point)
    }
```

# The Responder Chain

- A series of linked responder objects.
- The UIResponder class is the base class for all responder objects
    - Example, UIApplication, UIViewController, and UIView classes are responders
- The chain starts with the first responder and ends with the application object
- The first responder is designated to receive events first
- An object becomes the first responder by doing two things:
    - Overriding the canBecomeFirstResponder method to return YES
    - Receiving a becomeFirstResponder message. If necessary, an object can send itself this message in ViewDidAppear method

# Moving Up the Chain

- If the initial object, hit-test view or the first responder, doesn't handle an event, UIKit passes it to the next responder in the chain
- A responder shows it wants to handle an event by implementing the right event handling methods, or pass it along by calling nextResponder method
- This process continues until a responder handles the event or there are no more responders
- From initial view > super views > view controller > window > app

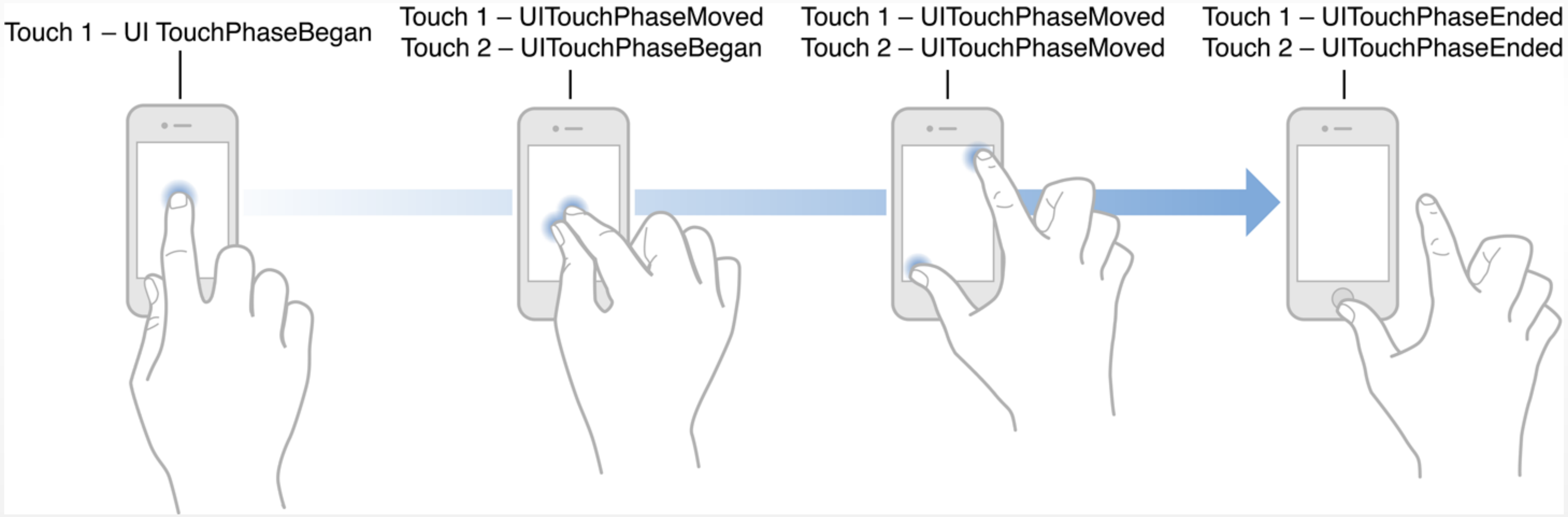


# Who Uses the Responder Chain?

- Touch events
- Motion events
- Remote control events
- Action messages, when a control, e.g., a button, has no target-action specified
- Editing-menu messages. iOS uses a responder chain to find an object that implements the necessary methods, such as cut, copy, and paste
- Text editing, when a user taps a text field/view, that view automatically becomes the first responder. The virtual keyboard appears and the text field/view becomes the focus of editing

```
//To dismiss the keyboard when pressing the return key
textField.delegate = self //self must conform to UITextFieldDelegate

func textFieldShouldReturn(textField: UITextField!) -> Bool
    {
        textField.resignFirstResponder()
        return true
    }
```

# Multitouch Sequence and Touch Phases


Touch 1 – UI TouchPhaseBegan
Touch 1 – UITouchPhaseMoved
Touch 2 – UITouchPhaseBegan
Touch 1 – UITouchPhaseMoved
Touch 2 – UITouchPhaseMoved
Touch 1 – UITouchPhaseEnded
Touch 2 – UITouchPhaseEnded

# UITouch

- A touch, represent by UITouch object, is the presence or movement of a single finger on the screen
- A touch object persists throughout a multi-touch sequence
    - Tip: never retain a touch object when handling an event. Instead, copy the information of a touch in your code
- A multitouch sequence begins when a finger touches the screen and ends when the last finger is lifted
- A UIEvent encompasses all touches that occur during a multitouch sequence
- When a finger moves a UIEvent object is sent to a *responder* object for event handling

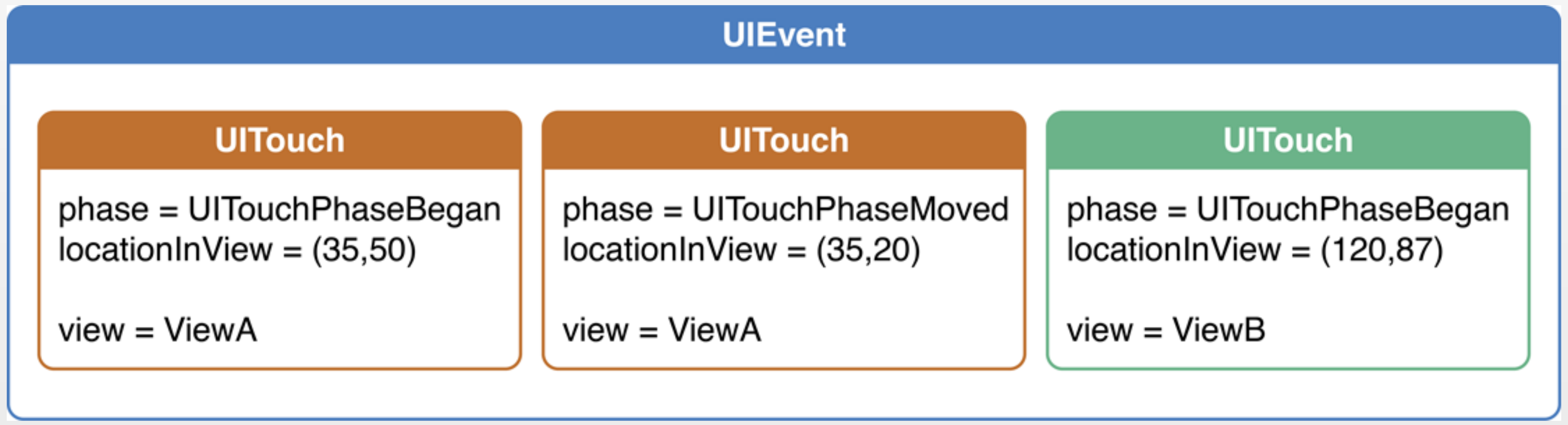
UITouch Properties:
The view or window in which the touch occurred
The location of the touch within the view or window
The approximate radius of the touch
The **force** of the touch (on devices that support 3D Touch)
The timestamp, when the touch occurred
The number of taps
The phase of touch
The **type** of touch
The gesture recognizer it belongs to (if any)

```
enum UITouchPhase : Int
{
        case Began
        case Moved
        case Stationary
        case Ended
        case Cancelled
    }
```

```
enum UITouchType : Int
{
        case Direct
        case Indirect
        case Stylus
    }
```

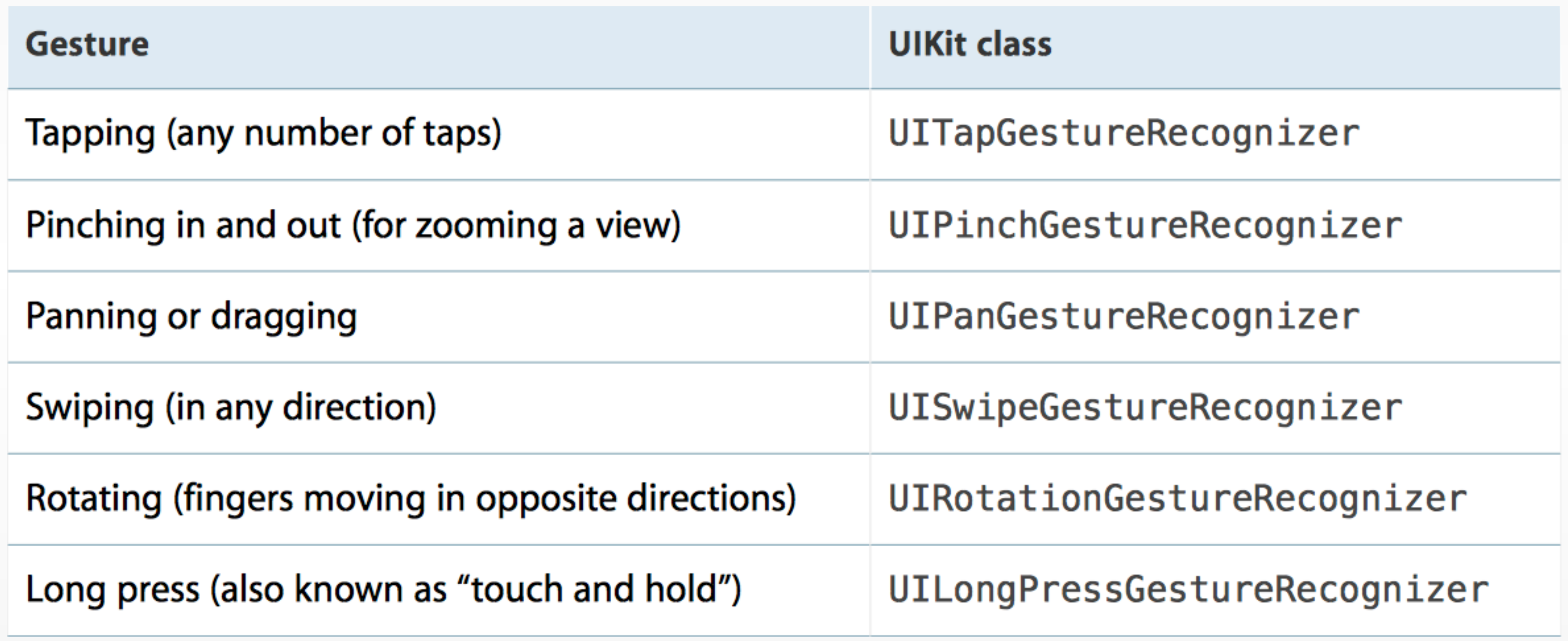
# Touch-Handling Methods

- During a multitouch sequence, an app sends these messages when a change in a touch object occurs (the methods correspond to UITouch phase property):
    - touchesX(_:withEvent:). Finger(s) touched the screen
    - X = [Began (finger(s) touched the screen) | Moved (finger(s) moved on the screen)| Ended (finger(s) lift off the screen) | Cancelled (system call)]
- These calls are made to the hit-test view, where the touches occurred
- 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
- UIEvent contains all touches in the multi-touch sequence



# Gestures Recognizers

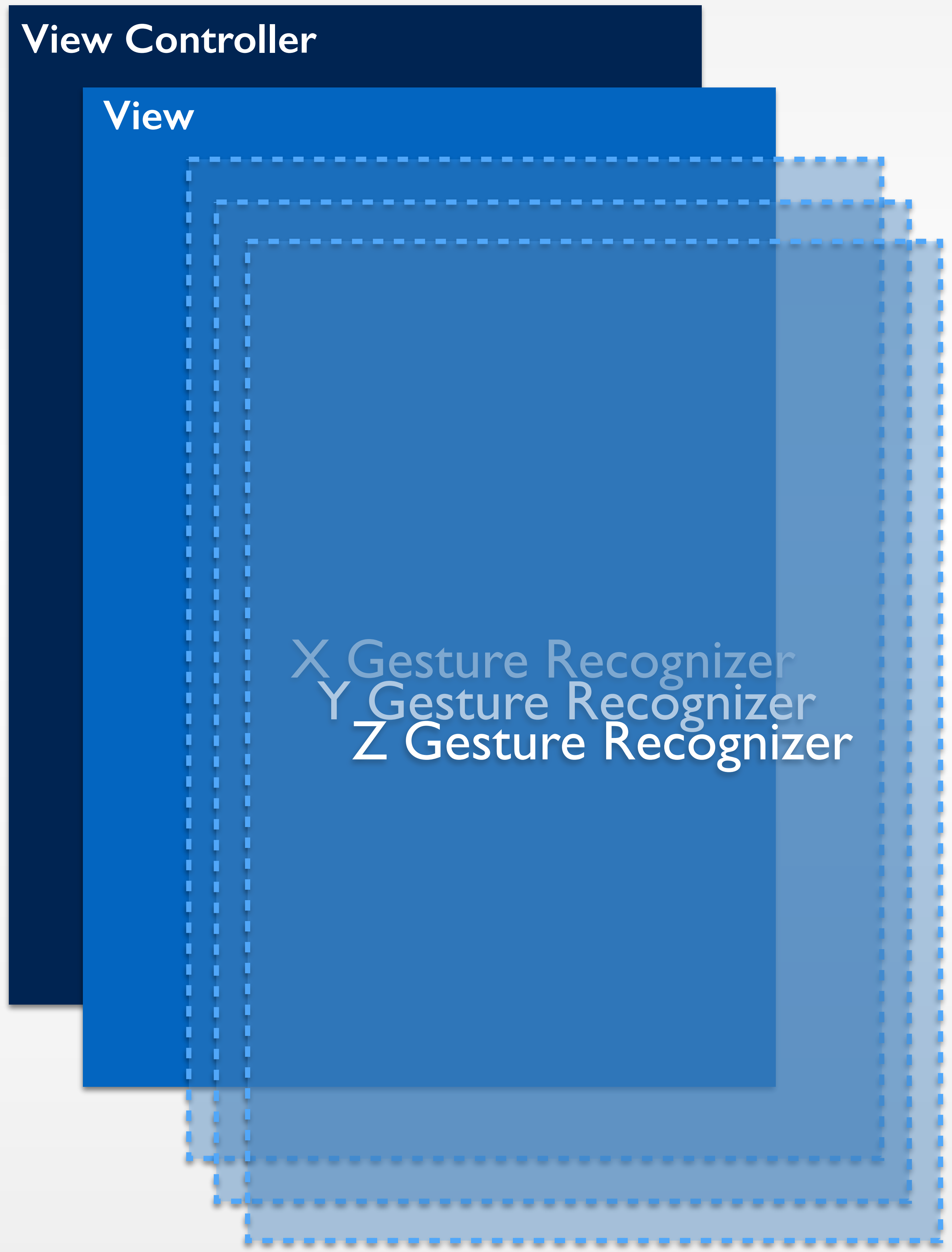
- iOS recognizes gestures, a combination of touches, and responds to them intuitively, e.g., UIKit recognizers scrolling in UISlider and tapping in UIButton
- Gesture recognizers decouple the logic for recognizing a gesture and acting on that recognition
- UIKit provides a set of built-in gesture recognizers that can be attached to views
- All recognizers are of type UIGestureRecognizer
- An app should respond to gestures only in ways that users expect



| 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` |

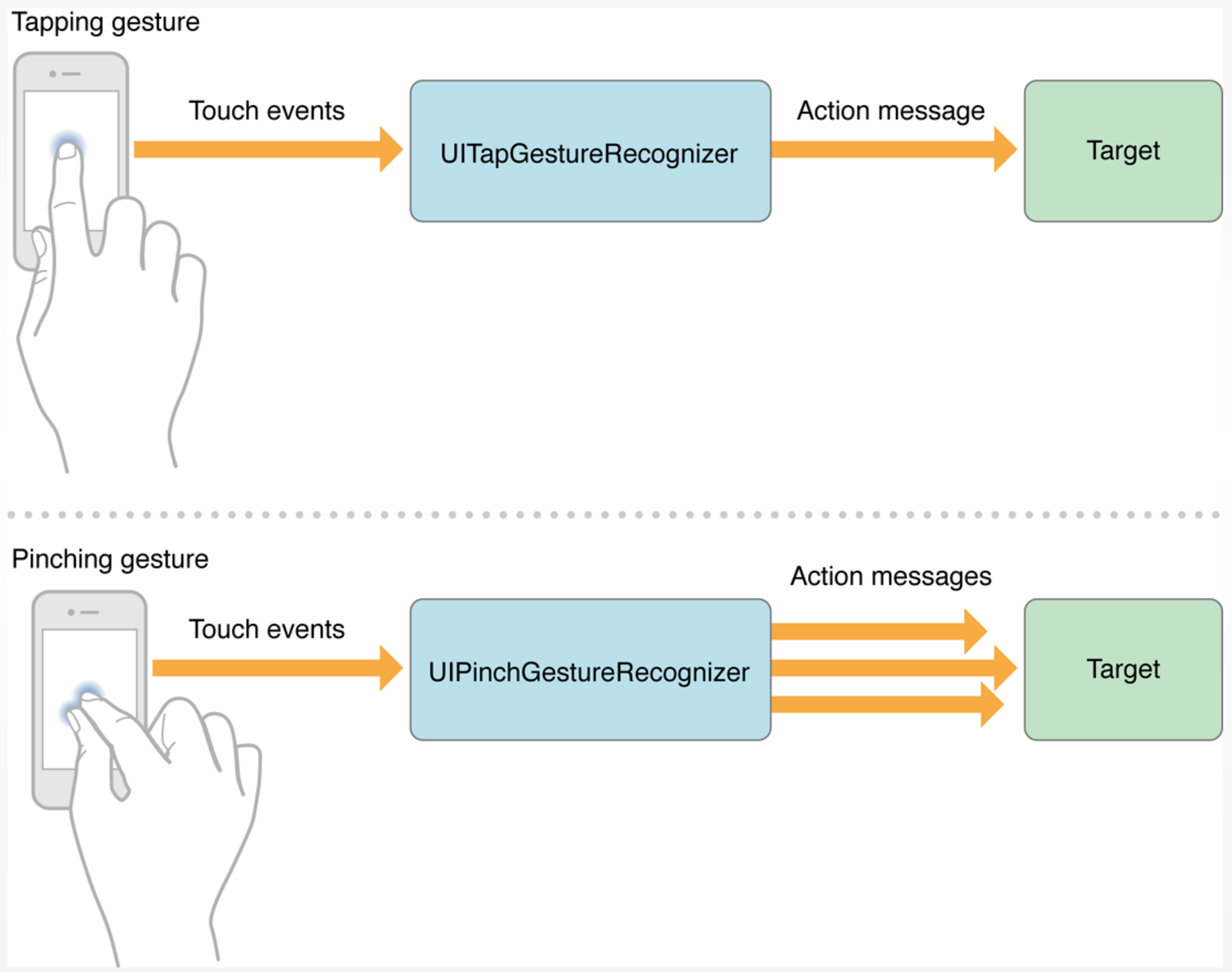
# Gestures Recognizers and Views

- Every view can have multiple gesture recognizers, but every gesture recognizer can be attached to one view only
- Views have the property
  - var gestureRecognizers: [UIGestureRecognizer]?
- A gesture recognizer attached to a view, interprets if incoming touches correspond to a specific gesture, and sends an action to target object
- A gesture recognizer can have more than one target-action pair.
- The action methods invoked can have no parameters or one of the gesture recognizer type, e.g., func rotate (sender: UIRotationGestureRecognizer)
- The passed parameter provide additional information. For example, ask a UIRotationGestureRecognizer object for the angle of rotation and locationInView:



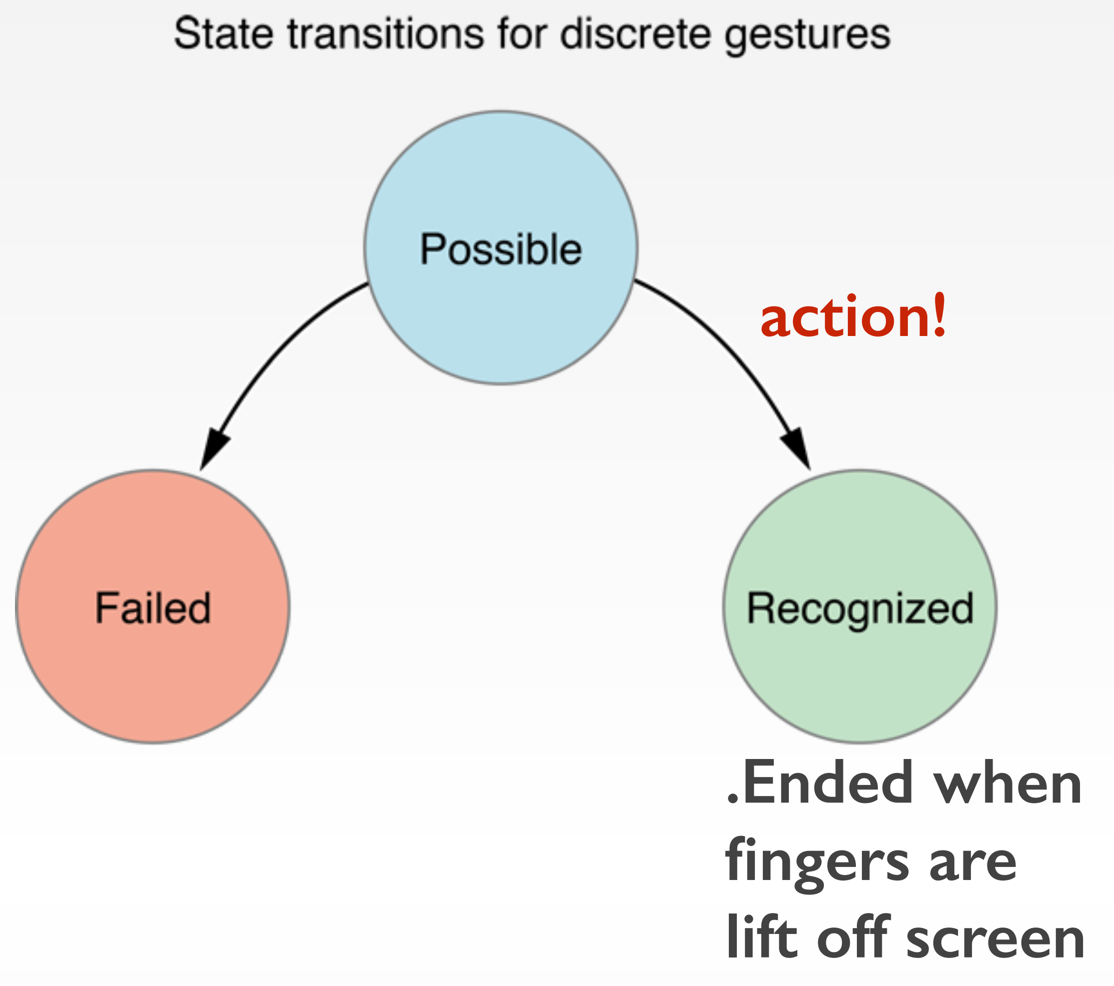
# Discrete and Continuous Gestures

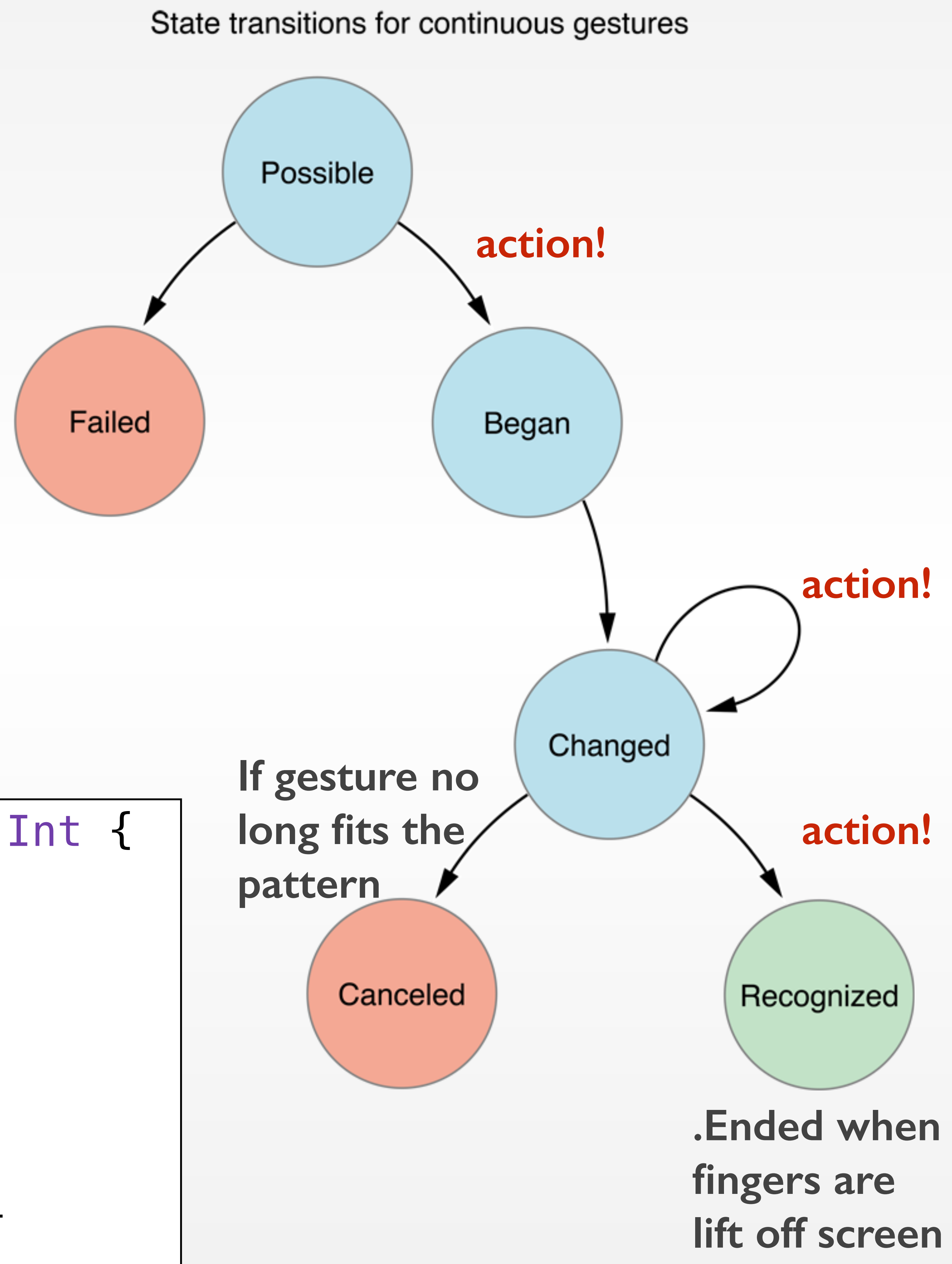
- A gesture recognizer can interpret either discrete or continuous gesture
- A discrete gesture, such as a double tap, occurs once in a multi-touch sequence, and only one action is sent
- A continuous gesture (over a period of time) such as a rotation gesture sends an action message for each incremental change until the multi-touch sequence ends



# Gesture Recognizers Finite State Machine

- Gesture recognizers transition from one state to another in a predefined way
- All gesture recognizers start in the Possible state, analyze any multitouch sequences that they receive, and during analysis they either recognize or fail
- Failing to recognize a gesture means the gesture recognizer transitions to the Failed state
- For continuous gestures, the recognizer transitions from Possible to Began when the gesture is first recognized
    - Ended state is an alias for the Recognized state


State transitions for discrete gestures
Possible
action!
Failed
Recognized
.Ended when fingers are lift off screen


State transitions for continuous gestures
Possible
action!
Failed
Began
action!
Changed
If gesture no long fits the pattern
action!
Canceled
Recognized
.Ended when fingers are lift off screen

```
enum UIGestureRecognizerState : Int {
        case Possible
        case Began
        case Changed
        case Ended
        case Cancelled
        case Failed
        static var Recognized:
UIGestureRecognizerState { get }
    }
```

# Managing Multiple Gesture Recognizers

- A view can have more than one gesture recognizer attached to it
- The order of attaching the recognizers to a view has no effect on which one receives and processes user touches first
- Can control three behaviours
  - 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

# Delivering Touches to Views with Recognizers

- When a touch occurs, the touch object is passed from the UIApplication object to the UIWindow object.
- The window first sends touches to any gesture recognizers attached to the view where the touches occurred (or to that view's superviews), before it passes the touch to the view object itself
- As long as the gesture recognizer is in possible state, the view receives the touch objects in the corresponding began/moved/ended method
- If the recognizer moves to Began state, touches are cancelled for the view
- If the recognizer moves to Failed state, touches are sent to the corresponding began/moved/ended method in the view
- UIGestureRecognizer properties can override this default behaviour

# Gesture Delivery

- A window delivers touch events to a gesture recognizer before it delivers them to the (hit-tested) view attached to the recognizer
- If a recognizer analyzes a multi-touch sequence and recognizes its gesture (.Recognized), all the touches of the gesture become owned by the recognizer and their updates are never sent to the view (the ones who have been sent to the view are cancelled)
- Otherwise, if the recognizer doesn't recognize its gesture, the view receives all touches in the sequence
- UIGestureRecognizer properties
    - cancelsTouchesInView(default true): described above
    - delaysTouchesBegan (default false), delaysTouchesEnded (default false): if true, do not send touches to began/ended methods of the view until the recognizer fails

# Custom Gesture Reconginzers

- import UIKit.UIGestureRecognizerSubclass
- Override
  - touchesBegan(_:withEvent:)
  - touchesMoved(_:withEvent:)
  - touchesEnded(_:withEvent:)
  - touchesCancelled(_:withEvent:)
  - reset()
    - If your gesture recognizer transitions to Ended, Canceled, or Failed, this method is called. Reset any internal state so that your recognizer is ready for a new attempt
- You must update the self.state of the recognizer yourself in each of these methods (you need a bridging header)

## Motion

# Motion Events

- Users generate motion events when they move, shake, or tilt the device
- Motion events are detected by the device hardware: accelerometer, gyroscope, and magnetometer
- Source of motion *data* depends on the type of motion
  - Device physical orientation (without orientation vector) – UIDevice
  - Shaking the device – UIKit event handling methods
  - Precise motion data – Core Motion

# Device Physical Orientation

- Device orientation doesn't alway match interface orientation (status bar)
- Access the device physical orientation using UIDevice (singleton class)
    - Turn on the accelerometer
    - Ask for orientation or register to receive orientation notifications

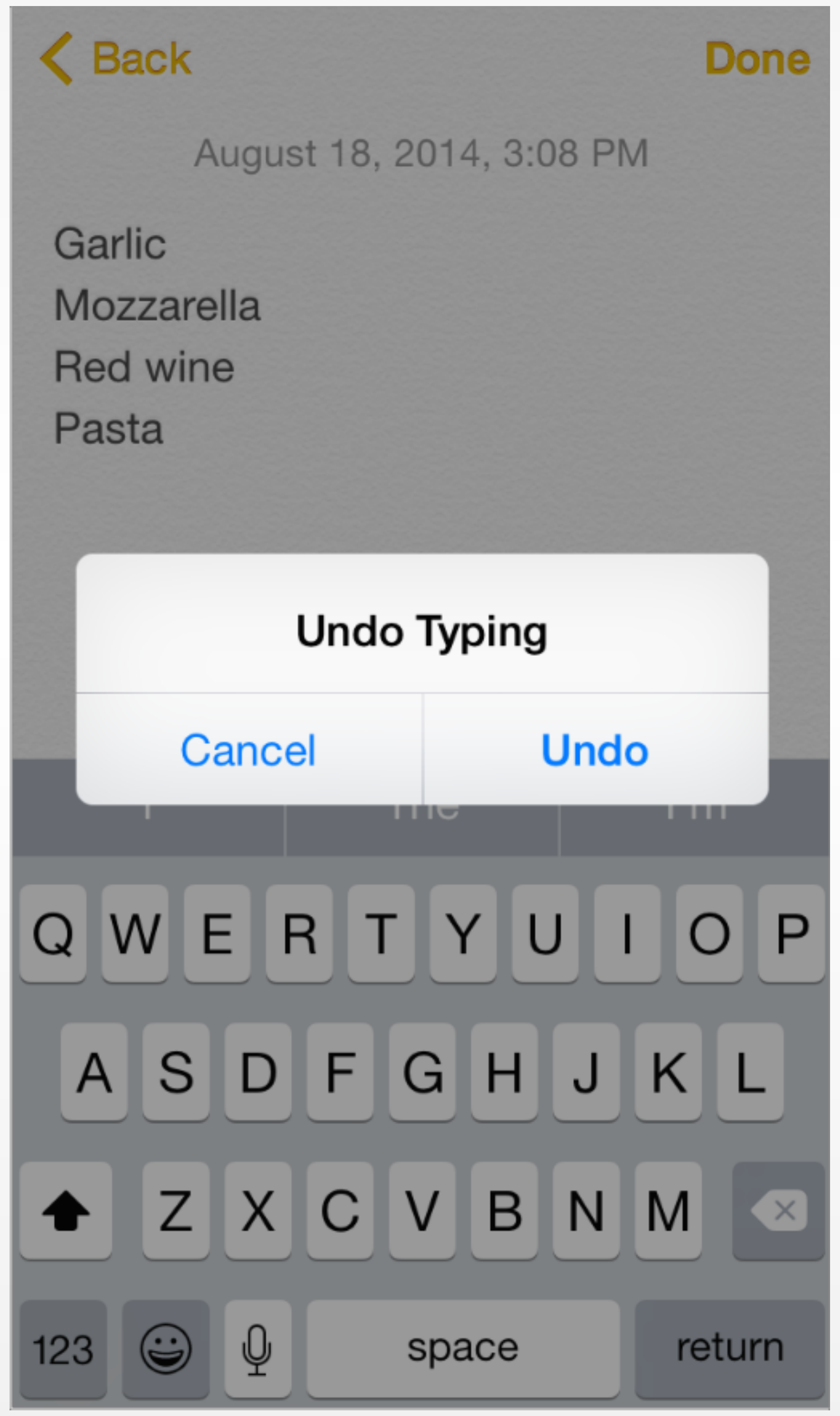
```
enum UIDeviceOrientation :
Int {
    case Unknown
    case Portrait
    case PortraitUpsideDown
    case LandscapeLeft
    case LandscapeRight
    case FaceUp
    case FaceDown
}
```

```
enum UIInterfaceOrientation :
Int {
    case Unknown
    case Portrait
    case PortraitUpsideDown
    case LandscapeLeft
    case LandscapeRight
}
```

```
//Turn on the accelerometer
UIDevice.beginGeneratingDeviceOrientationNotifications

NSNotificationCenter.defaultCenter().addObserver(self,
selector: "deviceOrientationChanged:", name:
UIDeviceOrientationDidChangeNotification, object: nil)

if UIApplication.sharedApplication().statusBarOrientation
== UIDevice.currentDevice().orientation
{…}
//Always call
UIDevice.endGeneratingDeviceOrientationNotifications
```

# Shake-Motion Events

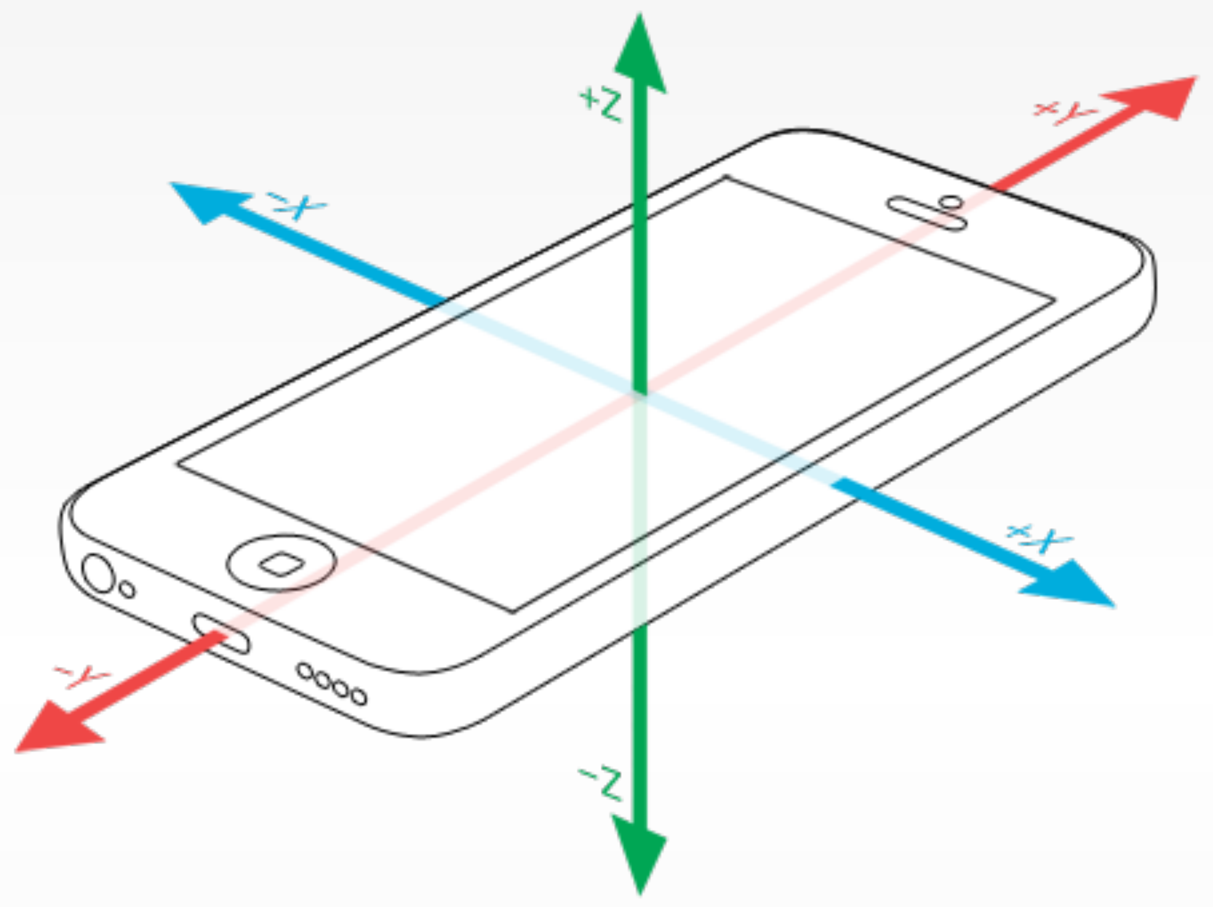
- When users shake a device, iOS evaluates if accelerometer data meets certain criteria to detect a shake
- If a shake is detected, a UIEvent object is created and sent to the currently active app for processing
- To receive shake events
  - Designate a responder object as the first responder
  - Implement these event-handling methods
    - motionX(_:withEvent:)
    - X = [Began and/or Ended | Cancelled]
- If a shaking-motion event is not handled and UIApplication.applicationSupportsShakeToEdit is true (the default), iOS displays the undo-redo user interface



iOS Undo Architecture

# Core Motion

- Access to built-in motion sensors: accelerometer, gyroscope, magnetometer/compass, and barometer (altitude)
  - Store and processed data
- M7, M8 and M9 motion processors provide access to stored motion activity: step counts, distance moved, stairs climbed, and movement type (walking, cycling, automotive,etc.).



Accelerometer and gyroscope axes

# Accelerometer vs. Gyroscope

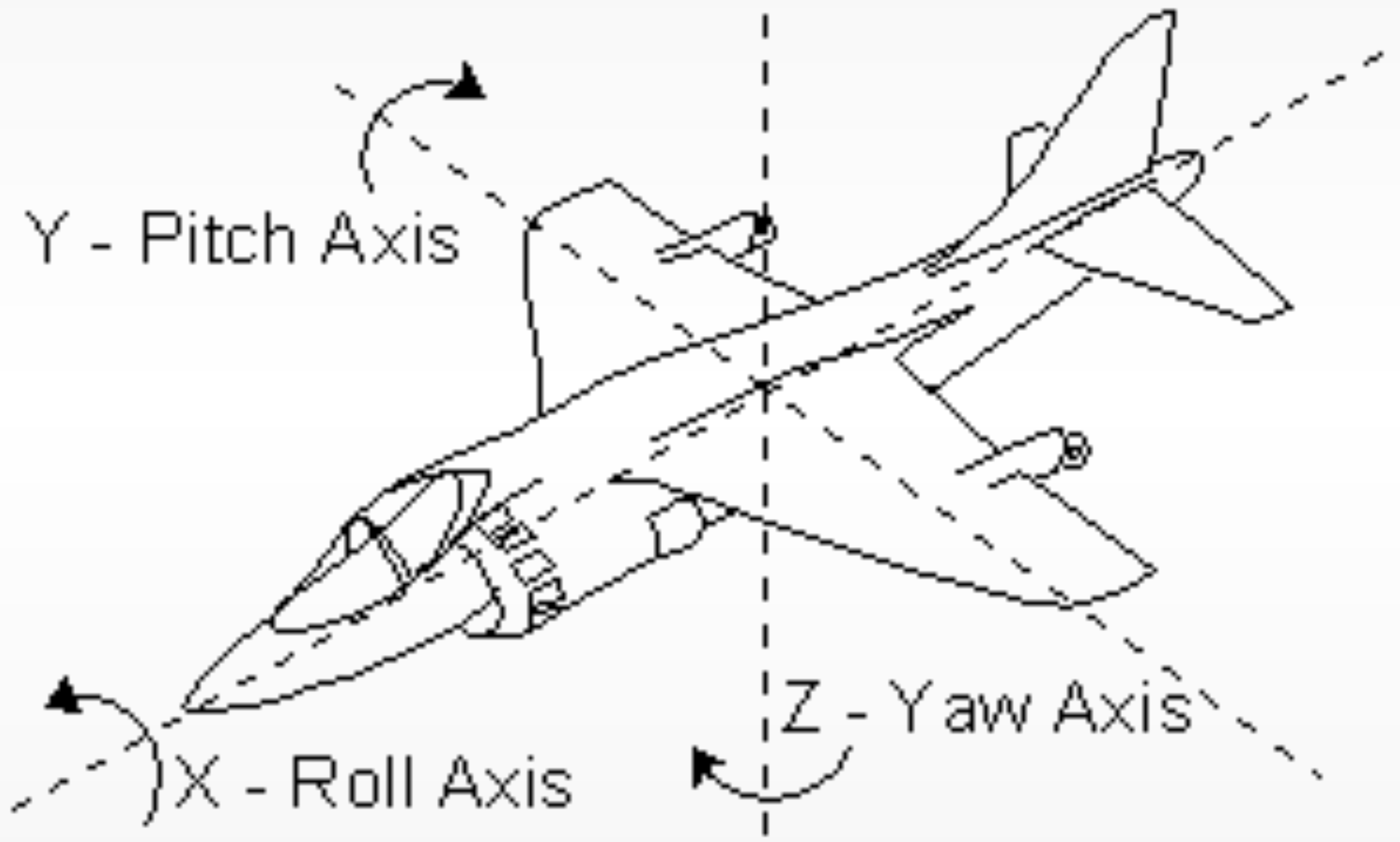
# CMMotionManager

- CMMotionManager is the main class to access motion data (use as single shared instance)
- Four motion types: accelerometer, gyro, magnetometer, and deviceMotion
- Tow access schemas: pull (any time) or push (based on time interval)
- In the example code, X = [Accelerometer | Gyro | DeviceMotion | Magnetometer]

```
//Check for availability
let manager = CMMotionManager()
if manager.XAvailable {
    // ...
}

//Setup the update interval
manager.XUpdateInterval = 0.1 // > 0.1 coarse less battery
//Pull
manager.startXUpdates()
manager.XData

//Push (in a bg thread)
manager.startXUpdatesToQueue(NSOperationQueue()) {
    [weak self] data, error in

    //Motion data processing data.X.y

    NSOperationQueue.mainQueue().addOperationWithBlock {
        //UI updates
    }
}

manager.stopXUpdates()
```

# CMDeviceMotion

- DeviceMotion encapsulates (more useful) measurements of the attitude, rotation rate, and acceleration of a device
- Use Gyroscope with accelerometer to improve acceleration data and provide device attitude
- Acceleration = gravity (used for device tilt) + user acceleration (shake-like motion)
- Attitude is the device position in 3D Space
  - Three representations of the device's orientation: Euler angles, a quaternion, and a rotation matrix
  - Euler angles: roll, pitch, yaw
  - Orientation is always in reference to some frame



## Motion Activities

- In iOS8 CMStepCounter was deprecated and replaced with CMPedometer
- iOS now provides high-level activity data (what the user is actually doing)
- Use CMMotionActivityManager to start/stop activity updates
- Updates are delivered as instances of CMMotionActivity class
- A CMMotionActivity object contains all data for each motion event
    - Activity state (Bool): stationary, running, walking, automotive, cycling, unknown (if the device has just turned on)
    - Other properties: startDate, confidence (confidence of data accuracy = low/medium/high)
- Access realtime data with push schema, or access data from a time period

## Next Time

- The slides and demos from this lecture will be uploaded to our website
- This week's reading assignment will be on the website today
- Next lecture is about maps and location in iOS with Krishna