Stanford CS193p

Developing Applications for iOS
Fall 2017-18

Today

- Views
  - PlayingCard demo continued
- Gestures
  - Getting multitouch input from users
- Demo: Manipulating our Playing Card
  - Swiping, tapping and pinching

Demo

- PlayingCard continued …
  - Now that we have our PlayingCard Model, time to implement our Controller and View
  - Creating a custom `UIView` subclass
  - Drawing with Core Graphics and `UIBezierPath`
  - `UIView`'s `contentMode` (i.e. `redraw` vs. scaling the bits on `bounds` change)
  - Drawing with transparency
  - More `NSAttributedString` dictionary keys … `UIFont` and `NSParagraphStyle`
  - `UIFontMetrics` scaling for users who want larger fonts
  - Managing subviews of your custom `UIView`
  - Using `isHidden`
  - `CGAffineTransform`
  - Constraint Priority
  - `Assets.xcassets` and drawing with `UIImage`
  - `@IBDesignable` and `@IBInspectable`
  - Using `didSet` to ensure redraws and relayouts when properties change

Gestures

- We've seen how to draw in a `UIView`, how do we get touches?
  - We can get notified of the raw touch events (touch down, moved, up, etc.)
  - Or we can react to certain, predefined "gestures." The latter is the way to go!
- Gestures are recognized by instances of `UIGestureRecognizer`
  - The base class is "abstract." We only actually use concrete subclasses to recognize.
- There are two sides to using a gesture recognizer
  1. Adding a gesture recognizer to a `UIView` (asking the `UIView` to "recognize" that gesture)
  2. Providing a method to "handle" that gesture (not necessarily handled by the `UIView`)
- Usually the first is done by a Controller
  - Though occasionally a `UIView` will do this itself if the gesture is integral to its existence
- The second is provided either by the `UIView` or a `Controller`
  - Depending on the situation. We'll see an example of both in our demo.

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.

  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.

  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's `didSet` code gets called when iOS hooks up this outlet at runtime

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.

  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's `didSet` code gets called when iOS hooks up this outlet at runtime

  Here we are creating an instance of a concrete subclass of `UIGestureRecognizer` (for pans)

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's `didSet` code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of `UIGestureRecognizer` (for pans)
  The `target` gets notified when the gesture is recognized (here it's the Controller itself)

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  The target gets notified when the gesture is recognized (here it's the Controller itself)
  The action is the method invoked on recognition (that method's argument? the recognizer)

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  The target gets notified when the gesture is recognized (here it's the Controller itself)
  The action is the method invoked on recognition (that method's argument? the recognizer)
  Here we ask the UIView to actually start trying to recognize this gesture in its bounds

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  The target gets notified when the gesture is recognized (here it's the Controller itself)
  The action is the method invoked on recognition (that method's argument? the recognizer)
  Here we ask the UIView to actually start trying to recognize this gesture in its bounds
  Let's talk about how we implement the handler …

Gestures

- A handler for a gesture needs gesture-specific information
  - So each concrete subclass provides special methods for handling that type of gesture
- For example, `UIPanGestureRecognizer` provides 3 methods

```
func translation(in: UIView?) -> CGPoint // cumulative since start of recognition
func velocity(in: UIView?) -> CGPoint    // how fast the finger is moving (points/s)
func setTranslation(CGPoint, in: UIView?)
```

  - This last one is interesting because it allows you to reset the translation so far
  - By resetting the translation to zero all the time, you end up getting "incremental" translation
- The abstract superclass also provides state information

```
var state: UIGestureRecognizerState { get }
```

  - This sits around in `.possible` until recognition starts
  - For a continuous gesture (e.g. pan), it moves from `.began` thru repeated `.changed` to `.ended`
  - For a discrete (e.g. a swipe) gesture, it goes straight to `.ended` or `.recognized`.
  - It can go to `.failed` or `.cancelled` too, so watch out for those!

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

We are only going to do anything when the finger moves or lifts up off the device's surface

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

We are only going to do anything when the finger moves or lifts up off the device's surface

`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

We are only going to do anything when the finger moves or lifts up off the device's surface

`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)

Here we get the location of the pan in the `pannableView`'s coordinate system

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

We are only going to do anything when the finger moves or lifts up off the device's surface

`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)

Here we get the location of the pan in the `pannableView`'s coordinate system

Now we do whatever we want with that information

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the `action` was `pan(recognizer:)`

We are only going to do anything when the finger moves or lifts up off the device's surface

`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)

Here we get the location of the pan in the `pannableView`'s coordinate system

Now we do whatever we want with that information

By resetting the `translation`, the next one we get will be incremental movement

Gestures

- UIPinchGestureRecognizer

```
var scale: CGFloat              // not read-only (can reset)
var velocity: CGFloat { get } // scale factor per second
```

- UIRotationGestureRecognizer

```
var rotation: CGFloat           // not read-only (can reset); in radians
var velocity: CGFloat { get } // radians per second
```

- UISwipeGestureRecognizer

Set up the direction and number of fingers you want

```
var direction: UISwipeGestureRecoginzerDirection // which swipe directions you want
var numberOfTouchesRequired: Int                 // finger count
```

Gestures

- UITapGestureRecognizer

  This is discrete, but you should check for `.ended` to actually do something.

  Set up the number of taps and fingers you want …

```
var numberOfTapsRequired: Int    // single tap, double tap, etc.
var numberOfTouchesRequired: Int // finger count
```

- UILongPressRecognizer

  This is a continuous (not discrete) gesture (i.e. you'll get `.changed` if the finger moves)

  You still configure it up-front …

```
var minimumPressDuration: TimeInterval // how long to hold before its recognized
var numberOfTouchesRequired: Int       // finger count
var allowableMovement: CGFloat         // how far finger can move and still recognize
```

  Very important to pay attention to `.cancelled` because of drag and drop

Demo Code

Download the demo code from today's lecture.