Gestures in-depth

In this reading, you will be introduced to SwiftUI Gestures. Gestures are elements that can be attached to views and give them the ability to detect from simple to sophisticated touches.

If you would like to follow along with this reading, please download the LittleLemon_Logo image from here:

LittleLemon_Logo
ZIP File

You will need to add the image to the `Assets.xcassets` folder before you start.

onTapGesture

`onTapGesture` is the most basic of the gestures and serves to detect taps on views.

To use it, type a syntax like this:

```
struct ContentView: View {
  var body: some View {
    Text("Little Lemon Restaurant").onTapGesture {
        print("Text Tapped!")
    }
  }
}

```

This gesture detects single taps to the `Text` element and will print **Text Tapped!** on Xcode's console as soon as the tap is detected.

If you want to detect multiple taps, you can use a variation of this command:

```
struct ContentView: View {
  var body: some View {
    Text("Little Lemon Restaurant").onTapGesture(count:2) {
        print("Text Tapped!")
    }
  }
}

```

This code will detect double taps to a view.

onLongPressGesture

`onLongPressGesture` is another of SwiftUI's gestures and allows you to detect long presses.

The default syntax detects long presses of 0.5 seconds to a view, meaning that the user must keep a single finger touching the view for, at least, half a second, for the touch to be detected.

```
struct ContentView: View {
  var body: some View {
    Text("Little Lemon Restaurant").onLongPressGesture {
        print("Long Press Detected!")
    }
  }
}

```

As soon as the long press is detected, **Long Press Detected!** will be printed on XCode's console.

onLongPressGesture variations

`onLongPressGesture` has other parameters that can be used to fine-tune what you consider a long press.

The syntax goes like this:

```
struct ContentView: View {
    var body: some View {
        Text("Little Lemon Restaurant")
            .onLongPressGesture(minimumDuration: 4, maximumDistance: 15, perform: {
                print("Long Press Detected!")
            },
            onPressingChanged: { state in
                print (state)
            })
    }
}
```

`minimumDuration` – The minimum duration of the long press that must elapse before the gesture succeeds. In this example, long presses will be detected if the user touches the view with one finger for at least 4 seconds.

`maximumDistance` – The maximum distance that the fingers or cursor performing the long press can move before the gesture fails. In this case, this parameter has been adjusted to 15, meaning that the finger or cursor (on a Mac or iPad, if the user is using a mouse) can move 15 points in any direction and the long press will still be detected.

`Perform` – The action to perform when a long press is recognized. This closure receives no value.

`onPressingChanged` – A closure to run when the pressing state of the gesture changes, passing the current state as a parameter. State is a Boolean variable that will be true, as soon as a finger touches or a mouse click happens and false, as soon as the long press gesture itself is detected.

DragGesture()

The `DragGesture` is another gesture that can be applied to views. This is a bit more complex because it requires a variable controlling the view's position.

The syntax goes like this:

```
1   struct ContentView: View {
2       @State private var offsetValue = CGSize.zero
3       var body: some View {
4           Image("littleLemonLogo")
5           .offset(offsetValue)
6               .gesture(
7                   DragGesture()
8                       .onChanged { gesture in
9                           offsetValue = gesture.translation
10                      }
11              )
12      }
13  }
14
```

This code works like this: a `DragGesture`, on line 7, is attached to the Little Lemon Logo image, line 4. `offsetValue`, initialized with CGSize equal to zero on line 2, stores the image's offset. The offset modifier on line 5 is the one that really moves the view.

The `onChanged` closure on line 8 is the one that does all the magic. As soon as the drag gesture is in progress, the closure continuously receives the relative gesture and from that is extracted the translation and stored in `offsetValue`, line 9.

RotationGesture

Another of the most used gestures around is the `RotationGesture`.

As the name says, it detects rotations to a view and its syntax goes like this:

```
1   struct ContentView: View {
2       @State private var amount = Angle.zero
3       @State private var finalAmount = Angle.zero
4       var body: some View {
5           Image("littleLemonLogo")
6               .rotationEffect(amount + finalAmount)
7               .gesture(
8                   RotationGesture()
9                       .onChanged { value in
10                          amount = value
11                      }
12                      .onEnded { value in
13                          finalAmount += amount
14                          amount = .zero
15                      }
16              )
17      }
18  }
19
20
```

This code works like this: two variables are defined, amount, on line 2, and `finalAmount`, on line 3. These variables store the relative amount of rotation, that is, how much rotation was applied to a view at one time and the total amount of rotation applied to a view, adding all rotations that happened while the application started respectively.

The modifier `rotationEffect` (line 6) is the one that rotates the Little Lemon Restaurant logo, on line 5.

`RotationGesture` is applied to the logo, on line 8 and has two closures: `onChanged` (line 9) and onEnded (line 12).

`onChanged` is triggered continuously as the rotation gesture is in progress and receives a value that corresponds to the relative rotation. This value is stored in the amount variable on line 10.

`onEnded` is triggered when the `RotationGesture` ends (line 12). When this closure runs, the code adds the amount to `finalAmount` and sets amount to zero.

MagnificationGesture

This code allows you to use the pinch gesture to magnify views and its syntax is like this:

```
1   struct ContentView: View {
2       @State private var amount = 0.0
```

```
2       @State private var amount = 0.0
3       @State private var finalAmount = 1.0
4
5
6       var body: some View {
7           Image("littleLemonRestaurant")
8               .scaleEffect(finalAmount + amount)
9               .gesture(
10          MagnificationGesture()
11                  .onChanged { value in
12                      amount = value - 1
13                  }
14                  .onEnded { value in
15                      finalAmount += amount
16                      amount = 0
17                  }
18              )
19      }
20  }
21
```

This code works similarly to the **`RotationGesture`**.

Two variables are defined, amount, on line 2, and **`finalAmount`**, on line 3. These variables store the relative amount of magnification, that is, how much magnification was applied to a view at one time and the total amount of magnification applied to a view, adding all magnification that happened while the application started, respectively.

The modifier **`scaleEffect`** (line 8) is the one that magnifies the Little Lemon Restaurant logo, on line 7.

`MagnificationGesture` is applied to the logo, on line 10 and has two closures: **`onChanged`** (line 11) and **`onEnded`** (line 14).

`onChanged` is triggered continuously as the magnification gesture is in progress and receives a value that corresponds to the relative magnification. This value is stored in the amount variable on line 12. This value is 1 subtracted from it, to obtain the relative magnification, because 1 is the normal scale

`onEnded` is triggered when the **`MagnificationGesture`** ends (line 14). When this closure runs, the code adds the amount to **`finalAmount`** and sets amount to zero.

Other more sophisticated gestures exist that are out of the scope of this course but are useful to know, like:

Sequenced

SwiftUI lets you create gestures which are sequences of other gestures, allowing you to run actions only when two gestures occur in a particular order.

The following code shows how to add two gestures, a **`LongPress`** and a **`DragGesture`**, that happen in sequence. The image view waits for the long press before it can process the drag.

```
1   struct ContentView: View {
2       @State private var message = "You can long press and then drag"
3
4
5       var body: some View {
6           let longPress = LongPressGesture()
7               .onEnded { _ in
8               print("Long Press detected. Now you can drag me")
9               }
10
11
12          let drag = DragGesture()
13              .onChanged{ _ in
14                  print ("Dragging…")
15              }
16
17
18          let sequence = longPress.sequenced(before: drag)
19
20
21          Image("littleLemonLogo")
22          .gesture(sequence)
23      }
24  }
25
```

Note: For clarity, we have not added the part that moves the image, but if you drag the image, after long tapping it, you will see the message "Dragging…" printed on Xcode's console.

There are other more advanced gestures that can be used and are outside the scope of this course:

- **`SimultaneousGesture`** contains two gestures that can happen at the same time with neither of them preceding the other.
- **`ExclusiveGesture`** is a gesture that consists of two gestures where only one of them can succeed.

Conclusion

In this reading, you were introduced to SwiftUI Gestures, which can be attached to views to allow the detection of taps, clicks, long presses, swipes and other gestures.

Mark as completed

Like Dislike Report an issue