

# Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 11



# Gestures

## • Getting Input from the User

SwiftUI has powerful primitives for recognizing “gestures” made by the user’s fingers.

This is called multitouch (since multiple fingers can be involved at the same time).

SwiftUI pretty much takes care of “recognizing” when multitouch gestures are happening.

All you have to do is handle the gestures.

In other words, decide what to do when the user drags or pinches or taps.



# Gestures

## • Making your Views Recognize Gestures

To cause your View to start recognizing a certain gesture, you use the `.gesture` View modifier.

`myView.gesture(theGesture) // theGesture must implement the Gesture protocol`

## • Creating a Gesture

Usually `theGesture` will be created by some func or computed var you create.

Or perhaps by a local var inside the body var of your View.

Example of the former ...

```
var theGesture: some Gesture {  
    return TapGesture(count: 2)  
}
```

This happens to be a “double tap” gesture (because of the `count: 2`).

SwiftUI will recognize this TapGesture now, but it won’t do anything about it ...



# Gestures

## • Handling the Recognition of a Discrete Gesture

So how do we “do something” about a recognized gesture?

It depends on whether the gesture is discrete or not.

TapGesture is a discrete gesture.

It happens “all at once” and just does one thing when it is recognized.

LongPressGesture can be treated as a discrete gesture as well.

To “do something” when a discrete gesture is recognized, we use `.onEnded { }`.

```
var theGesture: some Gesture {  
    return TapGesture(count: 2)  
        .onEnded { /* do something */ }  
}
```

Discrete gestures also have “convenience versions” that you’re already aware of.

```
myView.onTapGesture(count: Int) { /* do something */ }  
myView.onLongPressGesture(...) { /* do something */ }
```



# Gestures

## • Non-Discrete Gestures

Other gestures are not discrete.

In those, you handle not just the fact that the gesture was recognized ...

You also handle the gesture while it is in the process of happening (fingers are moving).

Examples: **DragGesture**, **MagnificationGesture**, **RotationGesture**.

**LongPressGesture** can also be treated as non-discrete (i.e. fingers down and fingers up).



# Gestures

## • Handling Non-Discrete Gestures

You still get to do something when a non-discrete gesture ends.

```
var theGesture: some Gesture {  
    DragGesture(...)  
        .onEnded { value in /* do something */ }  
}
```

Note though that its `.onEnded` passes you a `value`.

That `value` tells you the state of the DragGesture when it ended.

What that `value` is varies from gesture to gesture.

For a DragGesture, it's a struct with things like the start and end location of the fingers.

For a MagnificationGesture it's the scale of the magnification (how far the fingers spread out).

For a RotationGesture it's the Angle of the rotation (like the fingers were turning a dial).



# Gestures

## • Handling Non-Discrete Gestures

But during a non-discrete gesture, you'll also get a chance to do something while it's happening. Every time something changes (fingers move), you will get a chance to update some state.

This state is stored in a specially marked var ...

```
@GestureState var myGestureState: MyGestureStateType = <starting value>
```

This can be a variable of any value type you want.

You can store whatever information you need for your View to draw itself during the gesture. For example, during a drag, maybe you store how far the finger has moved.

This var will always return to <starting value> when the gesture ends.

So this var can only have a changing value while a gesture is in motion.

And you can, of course, update it during the gesture ...



# Gestures

## • Handling Non-Discrete Gestures

```
@GestureState var myGestureState: MyGestureStateType = <starting value>  
var theGesture: some Gesture {  
    DragGesture(...)  
        .updating($myGestureState) { value, myGestureState, transaction in  
            myGestureState = /* usually something related to value */  
        }  
        .onEnded { value in /* do something */ }  
}
```

This `.updating` will cause the closure you pass to it to be called when the fingers move.

Note the `$` in front of your `@GestureState` var. Don't forget this.

The `value` argument to your closure is the same as with `.onEnded` (the state of the fingers).

The `myGestureState` argument to your closure is essentially your `@GestureState`.

This is the only place you can change your `@GestureState`!

We're going to ignore the `transaction` argument (it has to do with animation).



# Gestures

## • Handling Non-Discrete Gestures

There's a simpler version of `.updating` if you don't need to track any special `@GestureState`. It's called `.onChanged`.

```
var theGesture: some Gesture {  
    DragGesture(...)  
        .onChanged { value in  
            /* do something with value (which is the state of the fingers) */  
        }  
        .onEnded { value in /* do something */ }  
}
```

This makes sense only if what you're doing is related directly to the actual finger positions. Maybe you're drag-to-selecting or letting the user use their finger like a pen? But usually you aren't interested in absolute finger position like this: you want relative position. (How far the fingers moved from when they first went down.) In that case you want to use `.updating()`.



# Gestures

## • Handling Non-Discrete Gestures

In summary, when handling a non-discrete gesture while the fingers are moving ...

Collect any information you need to draw your View during the gesture into a `@GestureState`.

Add `.updating` to your gesture.

In `.updating`, use the `value` that is passed to you to update your `@GestureState`.

Understand that when the gesture ends (fingers lift), your `@GestureState` will reset.

You update only `@GestureState` (not `Model/@State`) while the fingers are down and moving.

Your `Model/@State` is then permanently updated in `.onEnded`.

Your Views draw themselves using a combination of both `@GestureState` and `Model/@State`.

As usual, all best understood via demo!

