Stanford CS193p

Developing Applications for iOS

Spring 2020

Lecture 6

Today

- Property Observers
  - "Watching" a `var` and reacting to changes
- `@State`
  - State that is entirely localized inside a `View`
  - Only used for temporary state like presentations of alerts or editing things or animation
- Animation
  - Implicit vs. Explicit Animation
  - Animating `Views` (via their `ViewModifiers` which can implement the `Animatable` protocol)
  - Transitions (animating the appearance/disappearance of `Views` by specifying `ViewModifiers`)
  - Animating `Shapes` (via the `Animatable` protocol)

Property Observers

- Property Observers

  Just a quick word on this.

  You read about these in your reading assignment.

  Property observers are essentially a way to "watch" a `var` and execute code when it changes.

  The syntax can look a lot like a computed `var`, but it is completely unrelated to that.

```
var isFaceUp: Bool {
    willSet {
        if newValue {
            startUsingBonusTime()
        } else {
            stopUsingBonusTime()
        }
    }
}
```

  Inside here, `newValue` is a special variable (the value it's going to get set to).

  There's also a `didSet` (inside that one, `oldValue` is what the value used to be).

@State

- Your View is Read Only
  - It turns out that all of your `View` `structs` are completely and utterly read-only!
  - Yes, whatever variable SwiftUI is using to hold all your `Views` is a `let`!
  - So, other than `vars` that are initialized on creation of your `View`, it's useless to have a `var`.
  - Only `lets` or computed `vars` that are read-only make any sense whatsoever.
  - Take a moment to absorb that!
- Why!?
  - There are many good design reasons for SwiftUI to do this.
  - e.g., it makes it reliable and provable for it to manage changes and efficiently redraw things.
  - And this is actually a very wonderful restriction for you.
  - Views are mostly supposed to be "stateless" (just drawing their Model all the time).
  - They don't need any state of their own! So no need for them to be non-read-only!
  - Mostly ...

@State

- When Views need State
  - It turns out there are a few rare times when a View needs some state.
  - Such storage is always temporary.
  - All permanent state belongs in your Model.
- Examples
  - You've entered an "editing mode" and are collecting changes in preparation for an Intent.
  - You've displayed another View temporarily to gather information or notify the user.
  - You want an animation to kick off so you need to set that animation's end point.
- @State
  - You must mark any vars used for this temporary state with @State …
  - `@State private var somethingTemporary: SomeType` // this can be of any type
  - Marked `private` because no one else can access this anyway (except upon creating your `View`).
  - Changes to this `@State var` will cause your `View` to redraw if necessary!
  - In that sense, it's just like an `@ObservedObject`.

@State

- When Views need State

```
@State var somethingTemporary: SomeType
```

This is actually going to make some space in the heap for this.
(It has to do that because your `View` `struct` itself is read-only, remember?)
And when your read-only `View` gets rebuilt, the new version will continue to point to it.
In other words, changes to your `View` (via its arguments) will not dump this state.

Soon we will learn what these `@` things (like `@Published` & `@ObservedObject`) are, but not yet.
For now, just know that any read-write var in your `View` must be marked with `@State`.
Use them sparingly.

Animation

- What is animation?
  - A smoothed out portrayal in your UI ...
  - ... over a period of time (which is configurable and usually brief) ...
  - ... of a change that has already happened (very recently).
  - The point of animations is to make the user experience less abrupt.
  - And to draw attention to things that are changing.
- What can get animated?
  - You can only animate changes to `Views` in containers that are already on screen (CTAAOS).
  - Which changes?
    - The appearance and disappearance of `Views` in CTAAOS.
    - Changes to the arguments to `Animatable` view modifiers of `Views` that are in CTAAOS.
    - Changes to the arguments to the creation of `Shapes` inside CTAAOS.

Animation

- How do you make an animation "go"?
  - Two ways ...
  - Implicitly, by using the view modifier `.animation(Animation)`.
  - Explicitly, by wrapping `withAnimation(Animation) { }` around code that might change things.

Animation

- Implicit Animation

  "Automatic animation." Essentially marks a `View` so that ...
  All `ViewModifier` arguments will always be animated.
  The changes are animated with the duration and "curve" you specify (next slide).

  Simply add a `.animation(Animation)` view modifier to the `View` you want to auto-animate.

```
Text("👻")
    .opacity(scary ? 1 : 0)
    .rotationEffect(Angle.degrees(upsideDown ? 180 : 0))
    .animation(Animation.easeInOut) // Swift could infer the Animation. part, of course
```

  Now whenever `scary` or `upsideDown` changes, the opacity/rotation will be animated.
  (Since changes to arguments to animatable view modifiers are animated.)
  Without `.animation()`, the changes to opacity/rotation would appear instantly on screen.

  Warning! The `.animation` modifier does not work how you might think on a container.
  A container just propagates the `.animation` modifier to all the `Views` it contains.
  In other words, `.animation` does not work not like `.padding`, it works more like `.font`.

Animation

- Animation

  The argument to `.animation()` is an `Animation` struct.

  It lets you control things about an animation …

  Its `duration`.

  Whether to `delay` a little bit before starting it.

  Whether it should `repeat` (a certain number of times or even `repeatForever`).

  Its "curve" …

- Animation Curve

  The kind of animation controls the rate at which the animation "plays out" (it's "curve") …

  `.linear` This means exactly what it sounds like: consistent rate throughout.

  `.easeInOut` Starts out the animation slowly, picks up speed, then slows at the end.

  `.spring` Provides "soft landing" (a "bounce") for the end of the animation.

Animation

- Implicit vs. Explicit Animation

  These "automatic" implicit animations are usually not the primary source of animation behavior.
  They are mostly used on "leaf" (i.e. non-container) `Views`.
  Or, more generally, on `Views` that are typically working independently of other `Views`.

  Recall that you can't implicitly animate a container view (it propagates to the `Views` inside).
  That's because in containers you start wanting to be able to coordinate the `Views`' animations.
  Essentially, a bunch of `Views` that are contained together want to animate together.
  And they likely will all animate together in response to some user action or Model change.
  That's where explicit animation comes in ...

Animation

- Explicit Animation

  Explicit animations create an animation session during which …
  All eligible changes made as a result of executing a block of code will be animated together.

  You supply the `Animation` (duration, curve, etc.) to use and the block of code.

```
withAnimation(.linear(duration: 2)) {
    // do something that will cause ViewModifier/Shape arguments to change somewhere
}
```

  Note that this is imperative code in your `View`. It will appear in closures like `.onTapGesture`.

  Explicit animations are almost always wrapped around calls to ViewModel Intent functions.
  But they are also wrapped around things that only change the UI like "entering editing mode."
  It's fairly rare for code that handles a user gesture to not be wrapped in a `withAnimation`.

  Explicit animations do not override an implicit animation.

Animation

- Transitions

  Transitions specify how to animate the arrival/departure of `Views` in CTAAOS.

  A transition is nothing more than a pair of `ViewModifiers`.
  One of the modifiers is the "before" modification of the `View` that's on the move.
  The other modifier is the "after" modification of the `View` that's on the move.
  Thus a transition is really just a version of a "changes in arguments to view modifiers" animation.

  An `asymmetric` transition has 2 pairs of `ViewModifiers`.
  One pair for when the `View` appears and another pair for when the `View` disappears.
  Example: a `View` fades in when it appears, but then flies across the screen when it disappears.

Animation

- Transitions

  How do we specify the ViewModifiers to use when a View arrives/departs the screen?

  Using .transition(). Example using two built-in transitions, .scale and .identity …

```
ZStack {
    if isFaceUp {
        RoundedRectangle()s    // default .transition is .opacity
        Text("👻").transition(.scale)
    } else {
        RoundedRectangle(cornerRadius: 10).transition(.identity)
    }
}
```

  If isFaceUp changed (while ZStack is on screen and an explicit animation is in progress) …
  … to false, the back would appear instantly, Text would shrink to nothing, front RR fade out.
  … to true, the back would disappear instantly, Text grow in from nothing, front RR fade in.

  Unlike .animation(), .transition() does not get redistributed to a container's content Views.
  So putting .transition() on the ZStack above only works if the entire ZStack came/went.
  (Group and ForEach do distribute .transition() to their content Views, however.)

Animation

- Transitions

  `.transition()` is just specifying what the `ViewModifiers` are.

  It doesn't cause any animation to occur.

  In other words, think of the word `transition` as a noun here, not a verb.

  You are declaring what transition to use, not causing the transition to occur.

  Transitions do not work with implicit animations, only explicit animations.

Animation

- Transitions

  All the transition API is "type erased".
  We use the `struct` `AnyTransition` which erases type info for the underlying `ViewModifiers`.
  This makes it a lot easier to work with transitions.

  For example, here's how you get/make a transition ...
  `AnyTransition.opacity` (fades the `View` in and out as it comes and goes)
  `AnyTransition.scale` (uses `.frame` modifier to expand/shrink the `View` as it comes and goes)
  `AnyTransition.offset(CGSize)` (use `.offset` modifier to move the `View` as it comes and goes)
  `AnyTransition.modifier(active:identity:)` (you provide the two `ViewModifiers` to use)

  You can override the animation (curve/duration/etc.) to use for a transition.
  `AnyTransition` `structs` have a `.animation(Animation)` of their own you can call.
  This is not implicit animation! Transitions do not support implicit animation.
  You're just overriding the `Animation` parameters to use if/when the transition gets animated.
  e.g. `.transition(.opacity.animation(.linear(duration: 20)))` // a VERY slow fade

Animation

- `.onAppear`

  Remember that transitions only work on `Views` that are in CTAAOS.
  (Containers that are already on-screen.)
  If you want a transition animation to occur, the `View` has to appear after its container.
  How do you coordinate this?

  `View` has a nice function called `.onAppear { }`.
  It executes a closure any time a `View` appears on screen (there's also `.onDisappear { }`).

  Use `.onAppear { }` on your container view to cause a change (usually in Model/ViewModel)
  that results in the appearance of the `View` you want to animate the transition of.
  Since, by definition, your container is on-screen when its own `.onAppear { }` is happening,
  it is a CTAAOS, so any transition animations for its children that are appearing can fire.
  Of course, you'd need to use `withAnimation` inside `.onAppear { }`.

  You'll need this for your Assignment 3, because in that card game, "dealing cards" is animated.
  We'll also see in our demo today that we can use `.onAppear { }` to kick off animations.
  Especially ones that only make sense when a certain `View` is visible.

Animation

- Shape and ViewModifier Animation

  You've probably noticed by now that all actual animation happens in Shapes and ViewModifiers.
  So how do they participate in animation?

  Essentially, the animation system divides the animation duration up into little pieces.
  A Shape or ViewModifier lets the animation system know what information it wants piece-ified.
  (e.g. our Pie Shape is going to want to divide the Angles of the pie up into pieces.)
  The animation system then tells the Shape/ViewModifier the current piece it should show.
  And the Shape/ViewModifier makes sure that its code always reflects that.

Animation

- Shape and ViewModifier Animation

  The communication with the animation system happens (both ways) with a single var.
  This var is the only thing in the Animatable protocol.
  Shapes and ViewModifiers that want to be animatable must implement this protocol.

  ```
  var animatableData: Type
  ```

  Type is a don't care.
  Well … it's a "care a little bit."
  Type has to implement the protocol VectorArithmetic.
  That's because it has to be able to be broken up into little pieces on an animation curve.

  Type is almost always a floating point number (Float, Double, CGFloat).
  But there's another struct that implements VectorArithmetic called AnimatablePair.
  AnimatablePair combines two VectorArithmetics into one VectorArithmetic!
  Of course you can have AnimatablePairs of AnimatablePairs, so you can animate all you want.

Animation

- Shape and ViewModifier Animation

  Because it's communicating both ways, this `animatableData` is a read-write `var`.

  The setting of this `var` is the animation system telling the `Shape`/`VM` which piece to draw.
  The getting of this `var` is the animation system getting the start/end points of an animation.

  Usually this is a computed `var` (though it does not have to be).
  We might well not want to use the name "`animatableData`" in our `Shape`/`VM` code
  (we want to use variable names that are more descriptive of what that data is to us).
  So the `get`/`set` very often just `get`s/`set`s some other `var`(s)
  (essentially exposing them to the animation system with a different name).

  In the demo, we'll see doing this both for our `Pie Shape` and our `Cardify ViewModifier`.

Demo

- Match Somersault
  - Let's have our emoji celebrate when there's a match!
- Card Rearrangement
  - This is just changing the .position modifier on Views.
  - Automatically animated if an animation is in progress.
- Card Flipping
  - Cardify can handle this since it is a ViewModifier.
  - It will sync up the 3D rotation animation and the face-up-ness of the card.
- Card Disappearing on Match
  - This is a "transition" animation.
- Bonus Scoring Pie Animation
  - Make our Pie slice animate.