Stanford CS193p

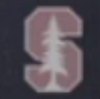
Developing Applications for iOS

Spring 2023

Lecture 8

Today

- Miscellaneous Topic: Property Observers and `.onChange(of:)`
- Animation
  - Implicit vs. Explicit Animation
  - Transitions (animating the appearance/disappearance of Views)
  - Matched Geometry Effect
  - Animating Shapes and ViewModifiers (via the `Animatable` protocol)
- Demos
  - Explicit Animation (shuffling and choosing cards)
  - Implicit Animation (celebrating a match!)
  - `Animatable` `ViewModifier` (flipping the cards over)
  - Suppressing unwanted animation (`.animation(nil)`)
  - `onAppear` animation (flying score change indications (incl. Property Observer and Tuples))
  - `TimelineView` (animating our pie-shaped countdown timer)
  - `transitions` (animating the arrival and departure of Views)
  - `matchedGeometryEffect` (dealing our cards out)



Property Observers

- Property Observers

  Remember that Swift is able to "detect" when a `struct` changes?

  Well, you can take action when it notices this if you like.

  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).

Property Observers

- `.onChange(of:) { }`

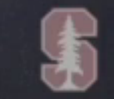
  Don't use a property observer on an `@State` variable.

  Instead, use this ViewModifier to detect a change to an `@State` or ViewModel var …

```
@State private var taps = 0

    Text("\(taps) taps")
        .onChange(of: viewModel.cards) { newCards in
            taps += 1
        }
```

  `newCards` is the value it's going to get set to.



Animation

- Important takeaways about Animation

  Only changes can be animated. Changes to what?

  - `ViewModifier` arguments (including `GeometryEffect` modifiers)
  - Shapes
  - The transition of a `View` from "existing" to "not existing" in the UI (or vice versa)

  Animation is showing the user changes that have already happened (i.e. the recent past).

  `ViewModifiers` are the primary "change agents" in the UI.
  A change to a `ViewModifier`'s arguments has to happen after the View is initially put in the UI.
  In other words, only changes in a `ViewModifier`'s arguments since it joined the UI are animated.
  Not all `ViewModifier` arguments are animatable, but most are.

  When a View arrives or departs, the entire thing is animated as a unit.
  A `View` coming on-screen is only animated if it's joining a container that is already in the UI.
  A `View` going off-screen is only animated if it's leaving a container that is staying in the UI.
  `ForEach` and `if-else` in ViewBuilders are common ways to make `Views` come and go.

Animation

- How do you make an animation "happen"?

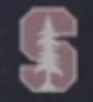
  Three ways ...

  Implicitly (automatically), by using the view modifier `.animation(Animation, value:)`.

  Explicitly, by wrapping `withAnimation(Animation) { }` around code that might change things.

  By making `Views` be included or excluded from the UI (aka "transitions").

  Again, all of the above only cause animations to "happen" if the `View` is already part of the UI (or if the `View` is joining a container that is already part of the UI)



Animation

- Implicit Animation

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

Simply add a .animation(Animation, value:) to the View you want to auto-animate.

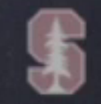
```
Text("👻")
    .opacity(card.scary ? 1 : 0)
    .rotationEffect(Angle.degrees(card.upsideDown ? 180 : 0))
    .animation(Animation.easeInOut, value: card)
```

Now whenever anything in card changes, the opacity/rotation will be animated.
All changes to arguments to animatable view modifiers preceding .animation 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 first argument to `.animation(_, value:)` 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.

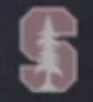


Animation

- 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, aka "Lego brick") `View`s.
  Or, more generally, on `View`s that are typically working independently of other `View`s.

  A more likely cause of animations is a change in response to some user action.
  For these changes, we want a whole bunch of `View`s to animate together.
  For that, we use Explicit Animation ...



Animation

- Explicit Animation

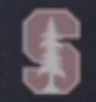
  Explicit animations create an animation transaction during which ...
  All eligible changes made as a result of executing a block of code will be animated together.

  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
}
```

  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

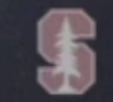
- Transitions

  Transitions specify how to animate the arrival/departure of Views.
  They only work for Views that are inside CTAAOS.
  (Containers That Are Already On-Screen.)

  Under the covers, 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 just a version of a "changes in arguments to ViewModifiers" animation.

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

  Mostly we use "pre-canned" transitions (opacity, scaling, moving across the screen).
  These can be found as static vars/funcs on the AnyTransition struct.

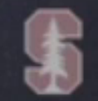


Animation

- 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 are some of the built-in transitions ...
  `AnyTransition.opacity` (uses `.opacity` modifier to fade the `View` in and out)
  `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)



Animation

- Transitions

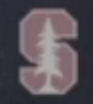
  How do we specify which kind of transition to use when a View arrives/departs?

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

```
ZStack {
    if isFaceUp {
        RoundedRectangle(cornerRadius: 10).stroke()
        Text("👻").transition(AnyTransition.scale)
    } else {
        RoundedRectangle(cornerRadius: 10).transition(AnyTransition.identity)
    }
}
```

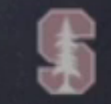
  If isFaceUp changed from false to true …
  (and ZStack was already on screen and we were explicitly animating)
  … the back would disappear instantly, Text would grow in from nothing, front RR would fade in.

  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.
- Setting Animation Details for a Transition
  - You can set an animation (curve/duration/etc.) to use for a transition.
  - `AnyTransition` structs have a `.animation(Animation)` of their own you can call.
  - This sets the `Animation` parameters to use to animate the transition.
  - e.g. `.transition(AnyTransition.opacity.animation(.linear(duration: 20)))`



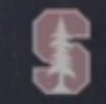
Animation

- Matched Geometry Effect

  Sometimes you want a `View` to move from one place on screen to another.
  (And possibly resize along the way.)
  If the `View` is moving to a new place in its same container, this is no problem (like shuffle).
  "Moving" like this is just animating the `.position` `ViewModifier`'s arguments.
  (`.position` is what `HStack`, `LazyVGrid`, etc., use to position the `Views` inside them.)
  This kind of thing happens automatically when you explicitly animate.

  But what if the `View` is "moving" from one container to a different container?
  This is not really possible.

  Instead, you need a `View` in the "source" position and a different one in the "destination" position.
  And then you must "match" their geometries up as one leaves the UI and the other arrives.
  So this is similar to `.transition` in that it is animating `Views`' coming and going in the UI.
  It's just that it's particular to the case where a pair of `Views`' arrivals/departures are synced.



Animation

- Matched Geometry Effect

  A great example of this would be "dealing cards off of a deck".
  The "deck" might well be its own `View` off to the side.
  When a card is "dealt" from the deck, it needs to fly from there to the game.
  But the deck and game's main `View` are not in the same `LazyVGrid` or anything.
  How do we handle this?

  We mark both `Views` using this `ViewModifier` ...

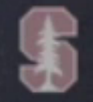
```
.matchedGeometryEffect(id: ID, in: Namespace) // ID type is a "don't care": Hashable
```

  Declare the `Namespace` as a `private var` in your `View` like this ..

```
@Namespace private var myNamespace
```

  Now write code so that only one of the 2 `Views` is ever included in the UI at the same time.
  You can do this with `if-else` in a `@ViewBuilder` or maybe via `ForEach`.
  Now, when one of the pair leaves and the other arrives at the same time,
  their size and position will be synced up and animated.

  It's possible to match geometries when both `Views` are on screen too.



Animation

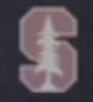
- .onAppear

  Remember that animations only work on Views that are in CTAAOS.
  (Containers That Are Already On-Screen.)
  How can you kick off an animation as soon as a View's Container arrives On-Screen?

  View has a nice function called .onAppear { }.
  It executes a closure any time a View appears on screen.

  Since, by definition, a View is on-screen when its own .onAppear { } is happening,
  it is in a CTAAOS, so any animations for it or its children that are appearing can fire.
  Of course, you'd need to use withAnimation inside .onAppear { }.

  We'll use .onAppear { } to kick off a couple of animations in demo this week.
  Especially ones that only make sense when a certain View is visible (e.g. our flying score).



Animation

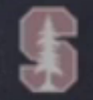
- Shape and ViewModifier Animation

  All actual animation happens in Shapes and ViewModifiers.
  (Even transitions and matchedGeometryEffects are just "paired ViewModifiers".)
  So how do they actually do their animation?

  Essentially, the animation system divides the animation's duration up into little pieces.
  (along whatever "curve" the animation uses, e.g. .linear, .easeInOut, .spring, etc.)

  A Shape or ViewModifier lets the animation system know what information it wants piece-ified.
  (e.g. our Cardify ViewModifier is going to want to divide rotation of the card into pieces.)

  During animation, the system tells the Shape/ViewModifier the current piece it should show.
  The Shape/ViewModifier makes sure its body draws appropriately at any "piece" value.



Animation

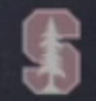
- Shape and ViewModifier Animation

  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 very often 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 gets/sets some other var(s)
  (essentially exposing them to the animation system with a different name).

  In the demo, we'll see doing this for our Cardify ViewModifier.