Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 6

Today

- Layout
  - How does Swift apportion space to its `Views`?
  - Demo: Picking the right size for our cards given the space we're offered.
- `@ViewBuilder`
  - More about how this works.
  - Demo: Creating our own "combiner View" (`AspectVGrid`).

Layout

- How is the space on-screen apportioned to the Views?

  It's amazingly simple ...

  1. Container Views "offer" some or all of the space offered to them to the Views inside them
  2. Views then choose what size they want to be (they are the only ones who can do so)
  3. Container Views then position the Views inside of them

Layout

- HStack and VStack

  Stacks divide up the space that is offered to them and then offer that to the Views inside.
  It offers space to its "least flexible" (with respect to sizing) subviews first …

  Example of an "inflexible" View: `Image` (it wants to be a fixed size).
  Another example (slightly more flexible): `Text` (always wants to size to exactly fit its text).
  Example of a very flexible View: `RoundedRectangle` (always uses any space offered).

  After an offered View(s) takes what it wants, its size is removed from the space available.
  Then the stack moves on to the next "least flexible" Views.
  Very flexible views (i.e. those that will take all offered space) will share evenly (mostly).
  Rinse and repeat.

  After the Views inside the stack choose their own sizes, the stack sizes itself to fit them.
  If any of the Views in the stack are "very flexible", then the stack will also be "very flexible."

Layout

- HStack and VStack

  There are a couple of really valuable Views for layout that are commonly put in stacks ...

  `Spacer(minLength: CGFloat)`

  Always takes all the space offered to it.

  Draws nothing.

  The `minLength` defaults to the most likely spacing you'd want on a given platform.

  `Divider()`

  Draws a dividing line cross-wise to the way the stack is laying out.

  For example, in an `HStack`, `Divider` draws a vertical line.

  Takes the minimum space needed to fit the line in the direction the stack is going.

  Takes all the space offered to it in the other (cross-wise) direction.

Layout

- HStack and VStack

  Stack's choice of who to offer space to next can be overridden with `.layoutPriority(Double)`.
  In other words, `layoutPriority` trumps "least flexible".

```
HStack {
    Text("Important").layoutPriority(100) // any floating point number is okay
    Image(systemName: "arrow.up") // the default layout priority is 0
    Text("Unimportant")
}
```

  The Important Text above will get the space it wants first.
  Then the Image would get its space (since it's less flexible than the Unimportant Text).
  Finally, Unimportant would have to try to fit itself into any remaining space.
  If a Text doesn't get enough space, it will elide (e.g. "Swift is..." instead of "Swift is great!").

Layout

- HStack and VStack

  Another crucial aspect of the way stacks lay out the Views they contain is alignment.

  When a VStack lays Views out in a column, what if the Views are not all the same width?
  Does it "left align" them? Or center them? Or what?

  This is specified via an argument to the stack …
  `VStack(alignment: .leading) { . . . }`

  Why `.leading` instead of `.left`?
  Stacks automatically adjust for environments where text is right-to-left (e.g. Arabic or Hebrew).
  The `.leading` alignment lines the things in the VStack up to the edge where text starts from.

  Text baselines can also be used to align (e.g. `HStack(alignment: .firstTextBaseline) { }`).

  You can even define your own "things to line up" alignment guides.
  But that's a bit beyond the scope of this course.
  So we're just going to use the built-ins (text baselines, `.center`, `.top`, `.trailing`, etc.).

Layout

- `LazyHStack` and `LazyVStack`
  - These "lazy" versions of the stack don't build any of their `Views` that are not visible.
  - They don't take up all the space offered to them even if they have flexible views inside.
  - You'd use these when you have a stack that is in a `ScrollView`.
- `LazyHGrid` and `LazyVGrid`
  - We've already seen how these lay out their `Views`.
  - Sizes its `Views` based on info given to the `Lazy*Grid` (e.g. the `columns:` argument).
  - The other direction can grow and shrink as more `Views` are added.
  - Also does not take all the space offered to it if it doesn't need it all.
- `Grid`
  - Allocates space to its `Views` both horizontally and vertically
    (notice that there's no `H` or `V` in its name).
  - Lots of alignment options across columns and rows (e.g. `.grid*()` modifiers).
  - Often used as a sort of "spreadsheet" `View` or "table of data" `View`.

Layout

- ScrollView
  - ScrollView takes all the space offered to it.
  - The views inside it are sized to fit along the axis your scrolling on.
- ViewThatFits
  - Takes a list of container views (e.g. an HStack and a VStack) and choose the one that fits.
  - This is great when laying out for landscape vs portrait.
  - Or when laying out with dynamic type sizes (things with large fonts might not fit horizontally).
- Form and List and OutlineGroup and DisclosureGroup
  - These are sort of like "really smart VStacks" (scrolling, selection, hierarchy, etc.).
  - We'll talk about them later in the quarter.
- Custom implementations of the Layout protocol
  - You can do a custom "offer space, let Views choose their size, then position them" process.
  - Implement a View that implements the Layout protocol (sizeThatFits, placeSubviews).

Layout

- ZStack

  ZStack sizes itself to fit its children.

  If even one of its children is fully flexible size, then the ZStack will be too.

- .background modifier

  ```
  Text("hello").background(Rectangle().foregroundColor(.red))
  ```

  This is similar to making a ZStack of this Text and Rectangle (with the Text in front).

  However, there's a big difference in layout between this and using a ZStack to stack them.

  In this case, the resultant View will be sized to the Text (the Rectangle is not involved).

  In other words, the Text solely determines the layout of this "mini-ZStack of two things".

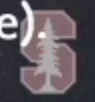
- .overlay modifier

  Same layout rules as .background, but stacked the other way around.

  ```
  Circle().overlay(Text("Hello"), alignment: .center)
  ```

  This will be sized to the Circle (i.e. it will be fully-flexible sized).

  The Text will be stacked on top of the Circle (with the specified alignment inside the Circle).



Layout

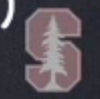
- Modifiers

  Remember that `View` modifier functions (like `.padding`) themselves return a `View`.
  That `View`, conceptually anyway, "contains" the `View` it's modifying.

  Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`).
  But it is possible for a modifier to be involved in the layout process itself.

  For example the `View` returned by `.padding(10)` will offer the `View` that it is modifying
  a space that is the same size as it was offered, but reduced by 10 points on each side.
  The `View` returned by `.padding(10)` would then choose a size for itself which is
  10 points larger on all sides than the `View` it is modifying ended up choosing.

  Another example is a modifier we've already used: `.aspectRatio`.
  The `View` returned by the `.aspectRatio` modifier takes the space offered to it
  and picks a size for itself that is either smaller (`.fit`) to respect the ratio
  or bigger (`.fill`) to use all the offered space (and more, potentially) and respect the ratio.
  (yes, a `View` is allowed to choose a size for itself that is larger than the space it was offered!)
  `.aspectRatio` then offers the space it chose to the `View` it is modifying (as its "container").



Layout

- Views that take all the space offered to them

  Most Views simply size themselves to take up all the space offered to them.
  For example, Shapes usually draw themselves to fit (like RoundedRectangle).

  Custom Views (like CardView) do this too whenever sensible.
  But they really should adapt themselves to any space offered to look as good as possible.
  For example, we made CardView scale its font size to one that makes its emoji fill the space.

  But what if a View needs to know how much space it was offered in order to adapt?
  e.g., what if we wanted to pick a card size so that our LazyVGrid would not need to scroll?

  Put the View inside a container View called a GeometryReader ...

Layout

- GeometryReader

  You wrap this GeometryReader View around what would normally appear in your View's body …

```
var body: View {
    GeometryReader { geometry in // using trailing closure syntax for content: parameter
        . . .
    }
}
```

  The geometry parameter is a GeometryProxy.

```
struct GeometryProxy {
    var size: CGSize
    func frame(in: CoordinateSpace) -> CGRect
    var safeAreaInsets: EdgeInsets
}
```

  The size var is the amount of space that is being "offered" to us by our container.
  Now we can, for example, pick a card size appropriate to that sized space.

  GeometryReader itself (it's just a View) always accepts all the space offered to it.

Layout

- Safe Area

  Generally, when a View is offered space, that space does not include "safe areas".
  The most obvious "safe area" is the notch on an iPhone X and later.
  Surrounding Views might also introduce "safe areas" that Views inside shouldn't draw in.

  But it is possible to ignore this and draw in those areas anyway on specified edges ...

```
ZStack { ... }.edgesIgnoringSafeArea([.top]) // draw in "safe area" on top edge
```

Back to Memorize

- Layout Demo

  Use `GeometryReader` to pick the right size for our cards to fit the space offered.

@ViewBuilder

- @ViewBuilder

  Based on a mechanism in Swift to enhance a `var` to have special functionality.
  It's a simple mechanism for supporting a more convenient syntax for lists of `View`s.

  Developers can apply it to any of their functions that return something that conforms to `View`.
  If applied, the function still returns something that conforms to `View`
  But it will do so by interpreting the contents as a list of `View`s and combines them into one.

  That one `View` that it combines it into might be a `TupleView` (for two or more `View`s).
  Or it could be a `_ConditionalContent View` (when there's an `if-else` in there).
  Or it could even be `EmptyView` (if there's nothing at all in there; weird, but allowed).
  And it can be any combination of the above (`if`'s inside other `if`'s, etc.).

  Note that some of this is not yet fully public API (like `_ConditionalContent`).
  But we don't actually care what `View` it creates for us when it combines the `View`s in the list.
  It's always just `some View` as far as we're concerned.

@ViewBuilder

- @ViewBuilder

  Any func or read-only computed var can be marked with @ViewBuilder.

  If so marked, the contents of that func or var will be interpreted as a list of Views.

  For example, if we wanted to factor out the Views we use to make the front of a Card …

```
func front(of card: Card) -> some View {
    let shape = RoundedRectangle(cornerRadius: 20)
    shape.fill(.white)
    shape.stroke()
    Text(card.content)
}
```

  This is not legal syntax for a function. It's just a list of Views.

@ViewBuilder

- @ViewBuilder

  Any func or read-only computed var can be marked with @ViewBuilder.
  If so marked, the contents of that func or var will be interpreted as a list of Views.

  For example, if we wanted to factor out the Views we use to make the front of a Card …

```
@ViewBuilder
func front(of card: Card) -> some View {
    let shape = RoundedRectangle(cornerRadius: 20)
    shape.fill(.white)
    shape.stroke()
    Text(card.content)
}
```

  And it would be legal to put simple if-else's to control which Views are included in the list.
  (But this is just the front of our card, so we don't need any ifs.)

  The above would return a TupleView<RoundedRectangle,RoundedRectangle,Text>.

@ViewBuilder

- @ViewBuilder
  - You can also use `@ViewBuilder` to mark a parameter of a function or an init.
  - That argument's type must be "a function that returns a View".
  - ZStack, HStack, VStack, ForEach, LazyVGrid, etc. all do this (their `content:` parameter).
  - We'll show how this works later in demo.

Back to Memorize

- `@ViewBuilder`

  Let's understand better how something like `HStack` or `LazyVGrid` takes its arguments.

  We'll do this by creating our own "combiner `View`" called `AspectVGrid`.

  It's like `LazyVGrid`, but it resizes its content `Views` to fit the space offered to it.

Today

- Layout
  - How does Swift apportion space to its `Views`?
  - Demo: Picking the right size for our cards given the space we're offered.
- `@ViewBuilder`
  - More about how this works.
  - Demo: Creating our own "combiner View" (`AspectVGrid`).
- Shape
  - Drawing your own custom shape.
  - Demo: Drawing (but not yet animating) a "pie-shaped" countdown timer on our Memorize cards.