Assignment VI: Memorize Themes

Objective

The goal of this assignment is to learn about how to have multiple MVVMs in your application and how to present groups of `Views` via navigation or modally and how to present/edit a bunch of info via controls and forms.

Be sure to review the Hints section below!

Also, check out the latest in the Evaluation section to make sure you understand what you are going to be evaluated on with this assignment.

Due

This assignment is due before lecture 15.

Materials

- You will start this assignment with at least your Memorize from Assignment 2. You will probably also want to incorporate the changes from the Animation lectures so that your Memorize is awesome, but that is not required.

Required Tasks

1. Your game from A2 should no longer choose a random theme, instead, its ViewModel should have a `theme var` that can be set. Other than using this `theme var` to configure the game, your `EmojiMemoryGame` ViewModel should not have any other theme-related code in it (i.e. no initializing of themes, storing of themes, etc.).
2. Your Memorize application should now show a "theme chooser" UI when it launches (instead of showing a game).
3. Use a `List` to display the themes in this chooser.
4. Each row in the `List` must display the name of the theme, the color of the theme, how many cards are in the theme and some sampling of the emoji in the theme. How it arranges this is up to you.
5. Touching on a theme in the `List` should navigate to playing a game with that theme. In other words, the `List` is in a navigating container and a `NavigationLink` surrounds each theme in the `List` (the destination of the link is an `EmojiMemoryGameView`).
6. While playing a game, the name of the theme should be on screen somewhere and you should also continue to support existing functionality from A2 like score, new game, etc. (you may rearrange the UI to be different from A2's version if you wish).
7. Provide some UI (a `Button` or whatever) to add a new theme to the `List` in your chooser.
8. The chooser must support deleting themes (you probably want use swipe to delete for this).
9. The chooser must also support editing themes modally (i.e. via sheet or popover). How you cause this sheet or popover to appear is up to you.
10. The theme editing UI must use a `Form`.
11. In the theme editing UI, allow the user to edit the name of the theme, to add emoji to the theme, to remove emoji from the theme, to specify how many cards are in the theme, and to specify the color of the theme.
12. The themes must be *persistent* (i.e. relaunching your app should *not* cause all the theme editing you've done to be lost). The games themselves are *not* persistent (just the themes).
13. You can choose whether you want to build your application for iPhone only (with `NavigationStack`) or iPhone and iPad (with `NavigationSplitView`). Also see Extra Credit #2.
14. Get your application working on a physical iOS device of your choice (as you must for your final project as well).

Hints

1. While you are welcome to include the animation code from lecture in your Memorize game too (again, highly recommended!), you'll probably want to skip the dealing animation since your game is going to be resetting each time you change its theme and the dealing will likely be annoying.
2. Your theme choosing will be an entirely different MVVM from your game's MVVM.
3. You'll likely want to start by creating a ViewModel for your theme choosing view. This is just a store for your themes. A simple array of themes that you persist into `UserDefaults` or the file system as JSON is all you'll need.
4. The trickiest part of making your themes persistent is going to be the *color* of the theme. We know we can't represent a color in our theme as a `Color` (since that's a UI representation and is not `Codable` in any case). So we strongly recommend representing a color inside your theme in this assignment as a `struct` with 4 floating point numbers: the color's red, green, blue, and alpha (transparency) level (aka `RGBA`). To aid you in this, we have included some simple code below that converts back and forth from a `Color` to an `RGBA`.
5. Since your theme `struct` is now going to represent the color internally as this 4 floating point number `struct`, you'll want to be able to reconstruct a `Color` from this `struct`. The provided `RGBA` code can do that as well. You'll probably want to add some nice API to your theme (outside of the Model code) so that other code in your app (e.g. your View) doesn't even have to know about this `RGBA` thing.
6. This application is focused on the *themes*. The playing of the game is almost incidental. These games are, essentially, "temporary UI things" that the theme choosing view is utilizing to show you the theme in action. Think of the game-playing as just "testing out the theme". It might be hard to shift your perspective from the game(s) being the focus (in A1 and A2) to the themes being the focus now, but that's what we're doing in this assignment.
7. For example, navigating away from a game and back is probably going to reset the game. The Required Tasks don't say that this is not allowed (though it is unfortunate). Fixing this is actually Extra Credit (#3). We want you focused on the UI and persistence of the themes (not the games), so don't spend time on EC3 until you've finished all of the Required Tasks.
8. You can easily get a `Binding` to a particular theme in your theme store at any time by getting the index of the theme in the store's themes array using something along the lines of `index = store.themes.firstIndex(where: { $0.id == themeid })` and then getting a `Binding` to it using `$store.themes[index]`.
9. Don't make the code in your theme editing view one gigantic `var body`. Break it down into smaller `Views` (at least one for each `Section`, for example). Shoot for 12 lines of code per `func`/computed `var`. Don't be afraid to make a new `View` if you

need to. Similarly, cleanly organize the code in your custom `View` that shows each row in the `List`.

10. One of the goals of this assignment is to start getting some experience learning to use SwiftUI API that hasn't been directly covered in lecture. For example, you'll probably want to use `Stepper` for entering the number of pairs of cards in a theme and you'll probably want to use `ColorPicker` for choosing the color of the theme. Consider using `.swipeActions` to bring up your theme editing view.
11. Don't let the part of your theme editing UI that chooses a theme's number of pairs of cards choose a number that is more than the number of emoji available in the theme! Nor should you let it choose fewer than two pairs.
12. You'll have to decide what to do if there are (or threaten to be) fewer than two emoji in the theme at any point during editing. There are multiple reasonable approaches to this situation.
13. The Required Tasks don't say anything about what sort of UI you have to employ to add or remove emoji from your theme in your theme editor. That's up to you to decide.
14. If you target iPad+iPhone, then you'll be using `NavigationSplitView`. In this scenario, it's probably simplest to not even use a `.navigationDestination` and instead use `List(selection:)` and use your selection to choose what is showing in the `detail` side of the `NavigationSplitView`. You'll want the selected theme variable to be your theme's `id` (rather than a theme itself) since you'll be editing the themes out from under the (always visible on iPad) `List` and you want the selected theme to stay selected through these changes to the themes.
15. If you target iPhone only, then you'll be using `NavigationStack` without any selection in your `List` and will need to use `.navigationDestination(for:)` for the chosen theme (and perhaps `.navigationDestination(isPresented:)` for a newly-added theme if you so choose).
16. Suggested work order (again, just a Hint, not a Required Task) …
    a. If you put a bunch of theme-related code in your game's view model in A2, rip it out of there and put it in your theme `struct` where it belongs. The only thing theme-related that should be in your game's view model is a `var` to hold the theme which it should use to create the Model and report the theme's color to the UI.
    b. Update your theme `struct` to use `RGBA` instead of a `String` to represent a color. Your game view model might want to provide some "easy access" to this for the view (perhaps via an `extension` or two?).
    c. Create a simple, persistent store view model for your themes.
    d. Create a `List` of the themes found in the store and make this `List` be the "content view" of your application.

e. Add navigation from a theme to a game view whose view model uses that theme.

f. Add a button to add a new theme to your store (it should then appear in the `List` automatically if you've created your `List` properly).

g. Add swipe to delete (`.onDelete`) to the `ForEach` inside of your `List`. This should be a couple of lines of code. If it's not, move on to the next work item and come back to this one.

h. Add some UI somewhere to cause a (blank) editor view to appear modally in a `.sheet` (or `.popover`, but `.sheet` recommended).

i. Implement the theme editor view (you will need to pass a `Binding` to it that binds to the theme in the store that it should edit).

j. It would be nice if creating a new theme automatically opened up the editor on that theme (like we did with new Palettes in EmojiArt).

RGBA

```
struct RGBA: Codable, Equatable, Hashable {
    let red: Double
    let green: Double
    let blue: Double
    let alpha: Double
}

extension Color {
   init(rgba: RGBA) {
      self.init(.sRGB, red: rgba.red, green: rgba.green, blue: rgba.blue, opacity: rgba.alpha)
   }
}

extension RGBA {
   init(color: Color) {
      var red: CGFloat = 0
      var green: CGFloat = 0
      var blue: CGFloat = 0
      var alpha: CGFloat = 0
      UIColor(color).getRed(&red, green: &green, blue: &blue, alpha: &alpha)
      self.init(red: Double(red), green: Double(green), blue: Double(blue), alpha: Double(alpha))
   }
}
```

Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. `List`
2. `Form`
3. `NavigationView`
4. Modal presentation
5. `TextField`
6. Multiple MVVMs
7. `Codable` persistence
8. `UserDefaults`

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Project does not build without warnings.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it.

Extra Credit

We try to make Extra Credit be opportunities to expand on what you've learned this week. Attempting at least some of these each week is highly recommended to get the most out of this course. How much Extra Credit you earn depends on the scope of the item in question.

If you choose to tackle an Extra Credit item, mark it in your code with comments so your grader can find it.

1. Keep track of any emoji that a user removes from a theme as a "removed" or "not included" emoji. Then enhance your Theme Editor to allow them to put removed emoji back if they change their mind. Remember these removed emoji forever (i.e. you will have to add state to your theme `struct`).
2. Write the code **both** for an iPhone-only version and for an iPad+iPhone version. They are similar and can share a lot of code, but you'll learn how to use `NavigationSplitView` from one and `NavigationStack` with `.navigationDestination` from the other.
3. Navigating away from a game and back is likely going to start the game over (depending on how you've implemented the assignment). Make it so that it does *not* reset the game.

   Here's a suggestion about how to make that work: *Hold the view models of the games being played into your theme-choosing view in an* `@State`. Use a `Dictionary` whose keys are theme `id`s and whose values are the `EmojiMemoryGame` ViewModels for the games you can navigate to. This might seem kind of weird to put ViewModels in an `@State` instead of an `@ObservedObject`, but when you actually want to *use* the ViewModel, you're going to be passing it into an `@ObservedObject` in some other `View` (i.e. your `EmojiMemoryGameView`). You can use this `Dictionary` to get the ViewModel you need each time you navigate to play the game and also to update the theme in all of the games being played (i.e. in their ViewModels) whenever the user edits any of the themes (i.e. you could do this in an `onChange(of:` <the store's themes>`)` and probably also in the `onAppear` of your theme chooser). This is all only a Hint. You do not have to do it this way.