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.

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

You should finish this assignment before watching Lecture 13.

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. When the `theme var` in your `EmojiMemoryGame` ViewModel is set to something that was different than it was before, you should restart the game (i.e. the equivalent of hitting the New Game button).
3. Your Memorize application should now show a "theme chooser" UI when it launches. See the video from Lecture 1 for some ideas, but your UI does not have to look exactly like what is shown there.
4. Use a `List` to display the themes.
5. Each row in the `List` shows the name of the theme, the color of the theme, how many cards in the theme and some sampling of the emoji in the theme.
6. Touching on a theme in the `List` navigates (i.e. the `List` is in a `NavigationView`) to playing a game with that theme.
7. While playing a game, the name of the theme should be on screen somewhere and you should also continue to support existing functionality like score, new game, etc. (but you may rearrange the UI to be different from A2's version if you wish).
8. Navigating from being in the middle of playing a game back to the theme chooser and then back to the game should not restart that game unless the theme for that game was changed (i.e. because of Required Task 2).
9. Provide some UI to add a new theme to the `List` in your chooser.
10. The chooser must support an Edit Mode where you can delete themes and where tapping on the row will bring up a modally-presented (i.e. via sheet or popover) theme editor UI for that theme rather than navigating to playing a game with that theme.
11. The theme editor must use a `Form`.
12. In the theme editor, 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.
13. The themes must be *persistent* (i.e. relaunching your app should *not* cause all the theme editing you've done to be lost).
14. Your UI should work and look nice on both iPhone and iPad.
15. Get your application work on a physical iOS device of your choice.

Hints

1. Your theme chooser will be an entirely different MVVM from your game's MVVM.
2. You'll likely want to start by creating a ViewModel for your theme chooser. 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.
3. The trickiest part of making your themes persistent is going to be the *color* of the theme. You won't be able to store a color in your theme `struct` using `Color` because `Color` is a UI thing (and your theme `struct` is part of a Model now and thus UI-independent) and also because `Color` is not `Codable` (and you need to persist your theme `structs`). So we strongly recommend representing your color inside your theme `struct` as a `struct` with four `Doubles`: the color's red, green, blue, and alpha (opacity) level (aka `RGBA`). To aid you in this, we have included some simple `extensions` to `Color` to support converting between such a `struct` (we've called it `RGBAColor`) and `Color`. Thankfully, `Codable` *does* know how to automatically encode/decode a `struct` with four `Doubles` in it.
4. These `extensions` we've provided for `Color` to `RGBAColor` conversion cannot live in your Model's `.swift` file. That's because these `extensions` are UI code. They reference `Color` which is a UI thing. You should not even be `importing` `SwiftUI` into your theme `struct`'s code (since it's part of your Model), so the `extensions` we've provided should not even compile if put in your Model's code. Put them in an appropriate place somewhere else.
5. You could even enhance your theme `struct` so it *seems* like it is actually storing a `Color` by adding your own `extension` to your theme `struct`: a *computed* `var color: Color` that sets and gets your internal `RGBAColor`. You might also want to add an extra `init` in this `extension` that lets you create one of your theme `structs` using a `Color` instead of an `RGBAColor`. This `extension` should be trivial to write using the `inits` in the `extension` to `Color` we've provided. It also cannot live in your Model's code space (again because it won't even compile in a file that does not `import SwiftUI` and your Model file(s) should not be doing that).
6. Since this new themes-storing ViewModel is only a store (i.e. it does not do any logic), there's no reason not to make its array of themes non-`private` (i.e. you don't need to cover every kind of change you could make to a theme with a `function` in this new ViewModel, just let the `View` manipulate the array of themes directly).

7. Picking a good architecture for the data in an application is crucial to making your code clean and easy to understand. Normally we don't propose any specific data architecture so you can experience this design aspect yourself (as you did in A3), but we will suggest something in this case because there's a lot of UI to do in this assignment and laying a lot of data-structure-architecture work on you might distract from that. Our suggestion? Put the ***source of truth*** *for the games being played into your theme chooser's* `View` *in an* `@State` which is a `Dictionary` whose keys are theme identifiers and whose values are the `EmojiMemoryGame` ViewModels for the games you can navigate to. You can use this `Dictionary` each time you navigate to play the game (to get the ViewModel to use) and to update the theme in all of the games being played whenever the user edits any themes (i.e. `on` any `change of` the themes and probably `on` the first `appear`ance of your theme chooser too). This is only a Hint. You do not have to do it this way.
8. This application is focussed on the *themes*. The playing of the game is almost incidental. The game views are constantly coming and going (which is why you would never put the source of truth for each game into an `@StateObject` in its `EmojiMemoryGameView`, for example) and the games are even being reset automatically when its theme is edited, so these games are, essentially, "temporary UI things" that the theme-choosing `View` is utilizing (which is why making them `@State` in your theme-choosing `View` makes sense). 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, but that's what we're doing in this assignment.
9. You'll definitely want to "autosave" any changes to your theme store.
10. Don't make the code in your theme editor `View` gigantic. Break it down into smaller `View`s (maybe one for each `Section`, for example). Similarly, cleanly organize the code in your custom `View` that is showing each row in the `List`.
11. Your theme editor `View` will need a `Binding` to whatever theme it is editing. Consider using the subscript added to a `RangeReplaceableCollection` of `Identifiable`s in EmojiArt's `UtilityExtensions.swift` when you pass the `Binding` into it. Or you can subscript by an index into your theme store's array if you're not comfortable with that `extension`. Don't forget that your theme `struct` will have to be `Hashable` if you want to create a `Binding` to that subscript.
12. One of the goals of this assignment is to give you 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.
13. `ColorPicker` takes a `Binding` to the `Color` it is picking. It is legal to create a `Binding` to a computed `var`. (You don't have to do that if you don't want to, this is just a Hint.)

14. Don't let the part of your theme editor UI that chooses a theme's number of pairs of cards allow the user to choose a number that is more than the number of emoji available in the theme! Nor should you let the user choose fewer than two pairs.
15. 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.
16. 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 design.

RGBAColor

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

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

extension RGBAColor {
   init(color: Color) {
      var red: CGFloat = 0
      var green: CGFloat = 0
      var blue: CGFloat = 0
      var alpha: CGFloat = 0
      if let cgColor = color.cgColor {
         UIColor(cgColor: cgColor).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. `EditMode`
7. Multiple MVVMs
8. `UserDefaults`
9. Using new API (`Stepper`, `ColorPicker`, etc.)

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.

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`).