Assignment II: More Memorize

Objective

The goal of this assignment is to continue to recreate the demonstrations given through lecture 5 (now that we've added game logic) and then make some bigger enhancements (new game button, themes and scoring). It is important that you understand what you are doing with each step of recreating the demo from lecture so that you are prepared to do those enhancements.

This continues to be about experiencing the creation of a project in Xcode and typing code in from scratch. **Do not copy/paste any of the code from anywhere.** Type it in and watch what Xcode does as you do so.

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 6.

Materials

Xcode (as explained in assignment 1).

Your solution to Assignment 1.

Required Tasks

1. Get the Memorize game working as demonstrated in lecture (through lecture 5, your choice). Type in all the code. Do not copy/paste from anywhere.
2. If you're starting with your assignment 1 code, remove your theme-choosing buttons and (optionally) the title of your game. The shuffle button added in lecture is also optional.
3. Add a "New Game" button to your UI (anywhere you think is best) which begins a brand new game.
4. Add a formal concept of a "Theme" to your Model. A Theme consists of a **name** for the Theme, a set of **emoji** to use, a **number of pairs** of cards to show (which is not necessarily the total number of emojis available in the theme), and an appropriate **color** to use to draw the cards.
5. Support at least 6 distinct Themes in your game.
6. Each new game should use a *randomly chosen* Theme to display its cards.
7. If the number of pairs of emoji to show in a Theme is fewer than the number of emojis that are available in that Theme, then it should not just always use the first few emoji in the Theme. It must randomly use *any* of the emoji in the Theme. In other words, do not have any "dead emoji" in your code that can never appear in a game.
8. A new Theme should be able to be added to your game with a single line of code.
9. The cards in a new game should all start face down and shuffled.
10. Show the Theme's name in your UI. You can do this in whatever way you think looks best.
11. Keep score in your game by penalizing 1 point for every *previously seen* card that is involved in a *mismatch* and awarding 2 points for every *match* (whether or not the cards involved have been "previously seen"). See Hints below for a more detailed explanation. The score is allowed to be negative if the user is bad at Memorize.
12. Display the score in your UI. You can do this in whatever way you think looks best.

Hints

1. Your ViewModel's connection to its Model can consist of more than a single `var model`. It can be any number of `var`s. The "Model" is a *conceptual* entity, not a single `struct`.
2. A Theme is a completely separate thing from a `MemoryGame` (even though both are part of your application's Model). You should not need to modify a single line of code in `MemoryGame.swift` to support Themes (though you will of course have to modify `MemoryGame.swift` to support scoring).
3. A game is represented purely by an `EmojiMemoryGame` and a Theme. So creating a new game is simply a matter of creating new ones of both of these. Since we `@Published` our `EmojiMemoryGame var` in our ViewModel, any changes to it (including completely replacing it) will cause our UI to automatically react and update.
4. A Theme is part of your Model, so it must be UI-independent. Representing a color in a UI-independent way is surprisingly nuanced (not just in Swift, but in general). We recommend, therefore, that you represent a color in your Theme as a simple `String` which is the name of the color, e.g. "orange". Then let your ViewModel do one of its most important jobs which is to "interpret" the Model for the View. It can provide the View access to the current Theme's color in a UI-*dependent* representation (like SwiftUI's `Color struct`, for example). You might find a `switch` statement useful for this interpretation, but a cascading `if-else` is fine too. A `Dictionary` could also be a cool solution to this problem.
5. You don't have to support every named color in existence (a dozen or so is fine), but it'd probably be a good idea to do something sensible if your Model contains a color (e.g. "fuchsia") that your ViewModel does not know how to interpret.
6. We'll learn a better (though still not perfect) way to represent a color in a UI-independent fashion later in the quarter.
7. You might find `Array`'s `randomElement()` function useful in this assignment (though note that this function (understandably) returns an Optional, so be prepared for that!). This is just a Hint, not a Required Task. The `static` function `Int.random(in:)` could also be used if you prefer.
8. There is no requirement to use an Optional in this assignment (though you are welcome to do so if you think it would be part of a good solution).
9. You'll very likely want to keep the `static func createMemoryGame()` from lecture to create a new memory game. But that function needs a little bit more information to do its job now, so you will almost certainly have to add an argument to it.
10. On the other hand, you obviously *won't* need the `static let emojis` from last week's lecture anymore because emojis are now obtained from the current Theme.

11. It's quite likely that you will need to add an `init()` to your ViewModel. That's because you'll probably have one `var` whose initialization depends on another `var`. You can resolve this kind of catch-22 in an `init()` because, in an `init()`, you can control the *order* in which `var`s get initialized (whereas, when you use property initializers to initialize `var`s, the order is undetermined, which is why property initializers are not allowed to reference other (non-`static`) `var`s).
12. The code in your ViewModel's `init()` might look very, very similar to the code involved with your New Game mechanism since you obviously want to start a new game in both of these places. Don't worry if you end up with some code duplication here (you probably don't quite know enough Swift yet to factor this code out).
13. A card has "already been seen" only if it has, at some point, been face up *and then is turned back face down*. So tracking "seen" cards is probably something you'll want to do when you are changing a card from face up to face down (not the other way around).
14. Make sure you think carefully about where code goes to solve the various problems in this assignment (i.e. in the View, or in the ViewModel, or in the Model?). This assignment is mostly about MVVM, so this is very important to get right.
15. If you started a game by flipping over 🐧 + 👻, then flipping over a ✏️ + 🏀, then flipping over two 👻s, your score would be 2 because you'd have scored a match (and no penalty would be incurred for the flips involving 🐧, ✏️ or 🏀 because they have not (yet) been involved in a *mismatch*). If you then flipped over the same 🐧 again + 🐼, then flipped 🏀 + that same 🐧 yet again, your score would drop 3 full points down to -1 overall because that 🐧 card had already been seen (on the very first flip) and subsequently was involved in two separate mismatches (scoring -1 for each mismatch) and the 🏀 was also mismatched after already having been seen (-1). If you then flip the 🐧 + the other 🐧 that you finally found, you'd get 2 points for a match and be back up to 1 total point. 2 - 1 - 1 - 1 + 2 = 1.
16. The "already been seen" concept is about specific *cards* that have already been seen, not *emoji* that have been seen.
17. We're not making this a Required Task (yet), but try to put the keyword `private` or `private(set)` on any `var`iables or `func`tions where you think it would be appropriate.

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. MVVM
2. Intent functions
3. `init` functions
4. Type Variables (i.e. `static`)
5. Access Control (i.e. `private`)
6. `Array`
7. Closures

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. You can assume that the reader knows the iOS API and knows how the Memorize game code from the lectures works, but should <u>not</u> assume that they already know your (or any) solution to the assignment.

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. Don't let your code blow up if it tries to use an invalidly-specified Theme. For example, ignore attempts by a Theme to try to show more cards than it has emoji to represent or to show fewer than 2 pairs of cards (because the game makes no sense in that scenario). Ditto if a Theme specifies a color that cannot be handled by your code (see Hint 5). This sort of "internal consistency protection" is something you should be wiring into all code you write, so making it "extra credit" here is underplaying its importance. However, we're trying to focus the Required Tasks in this early assignment on the key SwiftUI concepts from the week.
2. Allow the creation of some Themes where the number of cards to show is not a specific, fixed number but is, instead, *a random number* (which changes with every New Game). We're not saying that *every* Theme now shows a random number of cards, just that *some* Themes can now be created in such a way as to show a random number of cards each time that Theme is used. In other words, the properties of a Theme now include whether the Theme shows a fixed number of cards or a random number of cards. Maybe an Optional would be helpful on this one?
3. Support a *gradient* as one of the available "colors" for a Theme. Hint: `fill()` and `strokeBorder()` can take a `Gradient` as an argument (just like they do a `Color`). This is a "learning to look things up in the documentation" exercise. It's also a bit tricky because `CardView`, which fills and strokes the card, does not have access to the ViewModel. You'll have to give `CardView` the information it needs to do its job properly, but maybe it's overkill to pass your entire ViewModel to your `CardView` just for this one, small task. You decide what you think is the cleanest solution.
4. Modify the scoring system to give more points for choosing cards more quickly (in other words, implement *time*-based scoring).

   For example, maybe you could give 200 points for each match (and -100 for a mismatch), but every time you choose the first card of a pair, it starts a timer which starts reducing the 200 points you can earn for a match by 20 points per second? You'd have to decide how the mismatch penalty is affected by timing as well (if at all).

   Another option might be to have the score be a timer which counts up (or down from some "goal time") and then gets slowed down by matches (and accelerated by mismatches). Using this "score is a time interval" strategy, you'll want to keep track of both the game's start time (which is constantly being adjusted by matches and mismatches) and likely also the game's end time so that you can show a final score once there are no unmatched cards left (since the score clock should stop ticking once the game is over).

You are free to use whatever scoring strategy you think makes for the best game play and fairest scoring. The idea here is to challenge yourself by attempting something not covered in lecture.

You will definitely want to familiarize yourself with the `Date struct`. It is how all things time-related are dealt with in Swift.

Be clear about where the logic for this scoring belongs in the MVVM architecture of your application.

It's beyond the scope of this assignment for you to have your View be "reacting" in real time as the score is changing. Instead you'll probably want to just update the score each time a card is chosen. However, if you're going to show your score as a time interval rather than as points then `Text` view has a really cool `.timer` style that you could use: `Text(startTime, style: .timer)` *will* show a constantly ticking timer of how long it's been (in seconds) since `startTime`.