Assignment I: Memorize

Objective

The goal of this assignment is to recreate the demonstration given in the first two lectures and then make some small enhancements. 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.

Mostly this is 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 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 3.

Materials

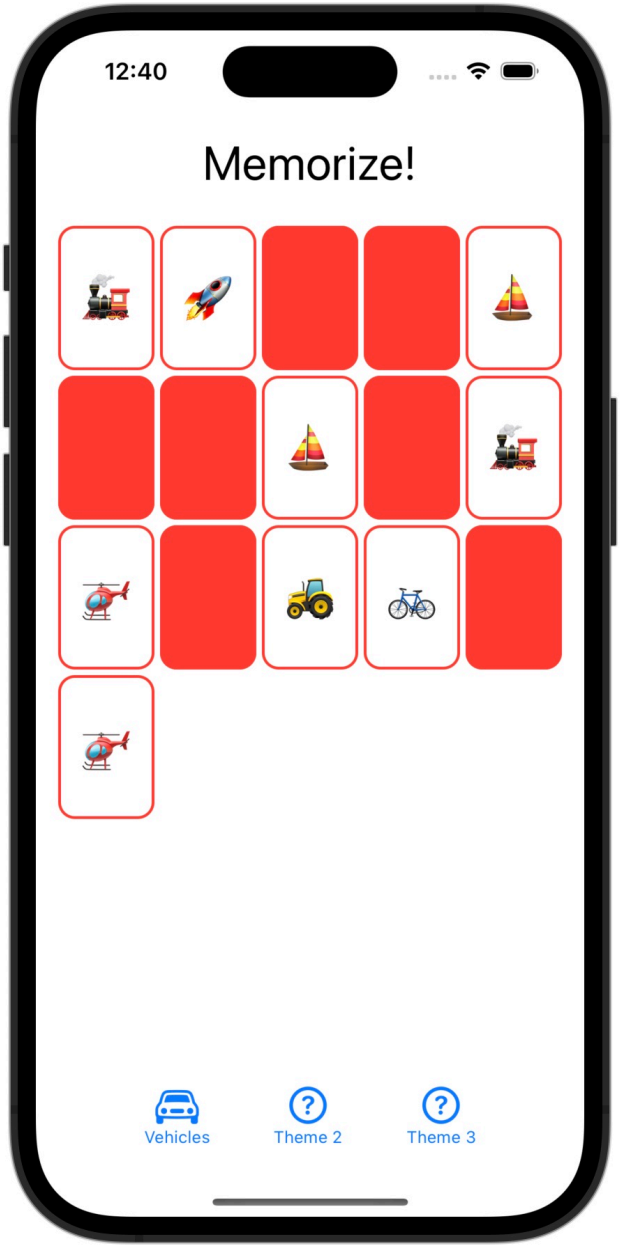
- You will need to install the (free) program Xcode 14 using the App Store on your Mac (previous versions of Xcode will not work).
- In order to recreate the demo, you will certainly need to watch the first two lectures.
- You may also want the SF Symbols application which can be downloaded from https://developer.apple.com/sf-symbols (though you can use the built-in symbol finding UI if you wish).

Required Tasks

1. Get the Memorize game working as demonstrated in lectures 1 and 2. Type in all the code. Do not copy/paste from anywhere.
2. After doing so, you can feel free to remove the ⊖ and ⊕ buttons at the bottom of the screen (or not, your choice) and all of the code that supports it.
3. Add a title "Memorize!" to the top of the screen. It's a title, so it should be in a large font.
4. Add *at least* 3 "theme choosing" buttons to your UI, each of which causes all of the cards' emojis to be replaced with new emojis from the chosen theme. A "theme" just means a collection of related emojis (for example, in lecture we basically had a "Halloween" theme).
5. The face up or face down state of the cards does *not* need to change when the user changes the theme.
6. The number of pairs of cards in each of your 3 themes should be different, but in no case fewer than 4 pairs. Note that you must put a *pair* of each emoji in the theme into the UI (the Memorize game wouldn't make much sense otherwise!).
7. The cards that appear when a theme button is touched should be in an unpredictable (i.e. random) order. In other words, the cards should be shuffled each time a theme button is chosen.
8. The theme-choosing buttons must include an image representing the theme and text describing the theme stacked on top of each other vertically.
9. The image portion of each of these theme-choosing buttons must be created using an SF Symbol which evokes the idea of the theme it chooses (like the car symbol for a Vehicles theme as shown in the Screenshot section below).
10. The text description of the theme-choosing buttons must use a noticeably smaller font than the font we chose for the emoji on the cards.
11. Change the code so that cards appear face down by default rather than face up (this should probably be the last thing you do since having them appear face up by default will be convenient as you work on all of the Required Tasks above). Once you do this, though, you will sort of be able to get a sense for what it will feel like to "play the game" once we add our game-play logic next week.
12. Your UI should work in portrait or landscape on any iPhone and look good in light mode and dark mode. This probably will not require any work on your part (that's part of the power of SwiftUI), but be sure to experiment with both orientations and both dark and light mode and with devices of various sizes too.

Screenshot

1. Screenshots are only provided in this course to help if you are having trouble visualizing what the Required Tasks are asking you to do. Screenshots are **not** part of the Required Tasks themselves (i.e. your UI does **not** have to look exactly like what you see below).



Hints

1. We are only prototyping the components of our UI this week. We'll implement our actual game play next week.
2. It's perfectly fine to have your application start out blank (i.e. no cards) until the user chooses one of your theme buttons. But if you remove all the Halloween emojis from the `emojis` array in the lecture code, be sure to explicitly set the type of the `emojis` `var` (since if you remove all the Halloween emojis from the array, Swift will no longer be able to infer that the `emojis` `var` is an `Array<String>`).

   ```
   var emojis: Array<String> = []
   ```

3. You will almost certainly want to make `emojis` in your `ContentView` be a `@State` `var` (since you're going to be changing the contents of this `Array` as you choose themes).
4. Unless you are doing the Extra Credit, choosing a theme shows *all* the emojis in that theme. Therefore, you will not need the `cardCount` `var` anymore and you will have to fix up the `ForEach`.
5. In Swift, you can combine two arrays with a + sign, e.g., `let combinedArray = array1 + array2`. This is only a Hint (not a Required Task).
6. Variables in your `View` (e.g. `var`s and `let`s) cannot be initialized using the values of other variables in your `View` (since the order of initialization is not guaranteed). Swift will complain if you try to do this.
7. Shuffling the cards might be easier than you think. Be sure to familiarize yourself with the documentation for `Array`. Note that there are some seemingly identical functions in `Array`, one of which is a verb and other is an adjective that is the past-tense of that verb. Try to figure out the difference (though you can use either one). In Swift, we generally prefer using the "past tense verb form" version. You'll find out why next week.
8. Other than reviewing the documentation for `Array`, you are not expected to use any aspect of Swift/SwiftUI that was not shown in lecture (unless you are doing the Extra Credit). You're welcome to use other stuff if you want, but the assignment can be done using only the Swift/SwiftUI features shown in lecture.
9. Remember how we avoided repeating code in lecture by creating the `cardCountAdjuster` `func`? You'll probably want to do the same thing when it comes to creating your theme-choosing buttons, though a computed `var` might suffice here.
10. For now, avoid choosing `func`tion parameter (aka argument) names that are exactly the same as the name of a variable in your `View` (e.g. `emojis`). This probably won't even come up unless you are doing the Extra Credit (esp. EC1).
11. Remember that the size of an SF Symbol `Image` is affected by `.font()`. SF Symbols are often interleaved with surrounding `Text` and so `Image(systemName:)` conveniently adjusts the size of the `Image` depending on the `.font()` it is modified with. It's

probably a good idea to use a larger size for your theme-choosing button images (but not for the text captions underneath them since a Required Task prohibits that). An SF Symbol `Image` is also affected by the `.imageScale()` modifier (which is affecting the size of the image *relative* to the font).

12. It would probably make sense for the baselines of the `Text` part of each of your theme-chooser buttons to be lined up. This is not a Required Task, but a good solution will consider this. Adventurous folks might want to check out the documentation for `VerticalAlignment` to really do this exactly right.

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. Xcode 14
2. Swift 5.8
3. Writing code in the in-line function that supplies the value of a `View`'s `body` `var`
4. Syntax for passing closures (aka in-line functions) (i.e. code in `{ }`) as arguments
5. Understanding the syntax of a ViewBuilder (e.g. "bag of Lego") function
6. Using basic building block `View`s like `Text`, `Button`, `Spacer`, etc.
7. Putting `View`s together using `VStack`, `HStack`, etc.
8. Modifying `View`s (using `.font()`, etc.)
9. Using `@State` (we'll learn much more about this construct later, by the way)
10. Very simple use of `Array`
11. The SF Symbols application
12. Putting system images into your UI using `Image(systemName:)`
13. Looking things up in the documentation (`Array` and possibly `Font`)
14. `Int.random(in:)` (Extra Credit)

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 SwiftUI API and knows how the Memorize game code from lectures 1 and 2 works, but should not assume that they already know your (or any) solution to the assignment.

Extra Credit

These are provided purely as a way to challenge yourself a bit. No actual "extra credit" is earned by doing these. This section might better be called "Extra Fun Things To Do".

1. Associate a (hopefully sensible) `Color` with each theme. In other words, whenever a "Halloween" theme is chosen, the cards are `.orange`, but if a "Water" theme is chosen, they are `.blue`. This will require you to have some `@State` that keeps track of this color.

2. Make a random number of pairs of cards appear each time a theme button is chosen. The function `Int.random(in: Range<Int>)` can generate a random integer in any range, for example, `let foo = Int.random(in: 15...75)` would generate a random integer between 15 and 75 (inclusive). Always show at least 2 pairs of cards though. You might need to figure out how to do a `for-in` loop in Swift. Or you could take advantage of the fact that the subscript of an `Array` is allowed to be a `Range`, e.g. `myArray[2..<6]` is legal. Take care to not violate the note in Required Task 6.

3. Try to come up with some sort of equation that relates the *number of cards* in the game to the width you pass when you create your `LazyVGrid`'s `GridItem(.adaptive(minimum:maximum:))` such that each time a theme button is chosen, the `LazyVGrid` makes the cards as big as possible without having to scroll.

   For example, if 8 cards are shown, the cards should be pretty big, but if 24 cards are shown, they should be smaller. The cards should still have our 2/3 aspect ratio.

   It doesn't have to be perfect either (i.e. if there are a few extreme combinations of device size and number of cards requiring scrolling, that's okay). The goal is to make it noticeably better than always using `65` or `80` is.

   It's probably impossible to pick a width that makes the cards fit just right in both Portrait and Landscape, so optimize for Portrait and just let your `ScrollView` kick in if necessary when the user switches to Landscape.

   Your "equation" can include some `if-else`'s if you want (i.e. it doesn't have to be a single purely mathematical expression) but you don't want to be special-casing every single number from 4 to 48 cards or some such. Try to keep your "equation" code *efficient* (i.e. not a lot of lines of code, but still works pretty well in the vast majority of situations).

   The type of the arguments to `GridItem(.adaptive(minimum:))` is a `CGFloat`. It's just a normal floating point number that we use for drawing. You know what kind of results `65` gives you, so you're going to have to experiment with other numbers up and down from there.

   You likely will want to put your calculation in a `func` or a `var`. Maybe something like:

   ```
   func widthThatBestFits(cardCount: Int) -> CGFloat { ... }, or ...

   var cardWidth: CGFloat { ... }
   ```

By the way, if you want to multiply a `CGFloat` by an `Int`, you need to create a `CGFloat` from the `Int`, e.g. `let x: CGFloat = CGFloat(anInt) * aCGFloat`.

Also remember that if your `func` or `var` has more than one line of code in it, you will need to use the `return` keyword to return a value from your `func` or `var`. In lecture it just so happened that all of our `func`s and `var`s had a single line of code. That may or may not be the case for your code.

Next week we'll dive into trying to really do this Extra Credit item right (and we'll consider not only the adaptive minimum and count of cards, but also the font size and the amount of space we have available to draw our cards in).