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

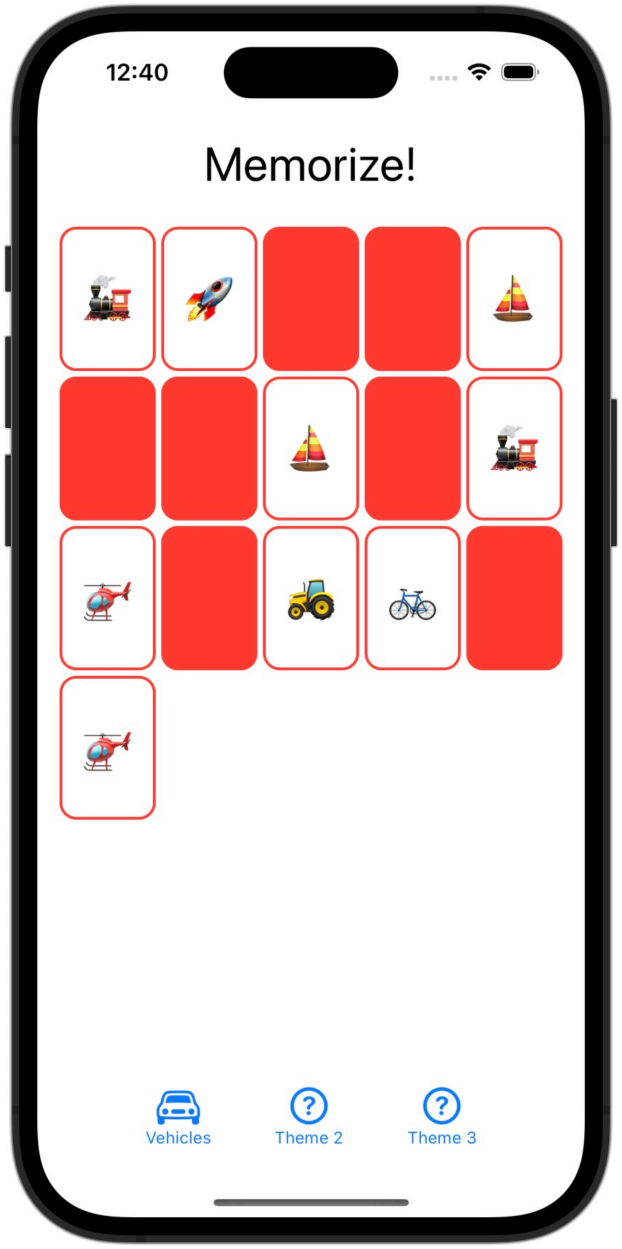
- You will need to install the (free) program Xcode 14 using the App Store on your Mac (previous versions of Xcode will not work).
- In order to recreate the demo, you will certainly need to watch the first two lectures.
- You may also want the SF Symbols application which can be downloaded from https://developer.apple.com/sf-symbols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

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 <u>not</u> 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 not 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`.

Assignment III: Set

Objective

The goal of this assignment is to give you an opportunity to create your first app completely from scratch by yourself. It is similar enough to the first two assignments that you should be able to find your bearings, but different enough to give you the full experience!

Since the goal here is to create an application from scratch, **do not start with your assignment 2 code, start with New → Project in Xcode**.

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 9. You have an extra couple of days to work on this assignment because it is larger in scope than assignments 1 and 2. It is effectively a "first midterm" in this course, so start working on it early.

Materials

- You can use any of the code from lecture (e.g. `AspectVGrid`).
- You will want to review the rules to the game of Set.

Required Tasks

1. Implement a game of solo (i.e. one player) Set.
2. As the game play progresses, try to keep all the cards visible and as large as possible. In other words, cards should get smaller (or larger) as more (or fewer) appear on-screen at the same time. It's okay if you want to enforce a minimum size for your cards and then revert to scrolling when there are a very large number of cards. Whatever way you deal with "lots of cards" on screen, it must always still be possible to play the game (i.e. cards must always be recognizable, even when all 81 are in play at the same time).
3. Cards can have any aspect ratio you like, but they must all have the *same* aspect ratio at all times (no matter their size and no matter how many are on screen at the same time). In other words, cards can be appearing to the user to get larger and smaller as the game goes on, but the cards cannot be "stretching" into different aspect ratios as the game is played.
4. The symbols on cards should be proportional to the size of the card (i.e. large cards should have large symbols and smaller cards should have smaller symbols).
5. Users must be able to select up to 3 cards by touching on them in an attempt to make a Set (i.e. 3 cards which match, per the rules of Set). It must be clearly visible to the user which cards have been selected so far.
6. After 3 cards have been selected, you must indicate whether those 3 cards are a match or mismatch. You can show this any way you want (colors, borders, backgrounds, whatever). Anytime there are 3 cards currently selected, it must be clear to the user whether they are a match or not (and the cards involved in a non-matching trio must look different than the cards look when there are only 1 or 2 cards in the selection).
7. Support "deselection" by touching already-selected cards (but only if there are 1 or 2 cards (not 3) currently selected).
8. When any card is touched on and there are already 3 **matching** Set cards selected, then …
   a. as per the rules of Set, replace those 3 matching Set cards with new ones from the deck
   b. if the deck is empty then the space vacated by the matched cards (which cannot be replaced since there are no more cards) should be made available to the remaining cards (i.e. which may well then get bigger)
   c. if the touched card was *not* part of the matching Set, then select that card
   d. if the touched card *was* part of a matching Set, then select no card

9. When any card is touched and there are already 3 **non-matching** Set cards selected, *deselect* those 3 non-matching cards and *select* the touched-on card (whether or not it was part of the non-matching trio of cards).
10. You will need to have a "Deal 3 More Cards" button (per the rules of Set).
    a. when it is touched, *replace* the selected cards if the selected cards make a Set
    b. or, if the selected cards do *not* make a Set (or if there are fewer than 3 cards selected, including none), add 3 new cards to join the ones already on screen (and do not affect the selection)
    c. disable or hide this button if the deck is empty
11. You also must have a "New Game" button that starts a new game (i.e. back to 12 randomly chosen cards).
12. To make your life a bit easier, you can replace the "squiggle" appearance in the Set game with a rectangle.
13. You must author your own `Shape` `struct` to do the diamond.
14. Another life-easing change is that you can use a semi-transparent color to represent the "striped" shading. Be sure to pick a transparency level that is clearly distinguishable from "solid".
15. You can use any 3 colors as long as they are clearly distinguishable from each other.
16. You must use an `enum` as a meaningful part of your solution.
17. You must use a closure (i.e. a function as an argument) as a meaningful part of your solution.
18. Your UI should work in portrait or landscape on any iOS device. This probably will not require any work on your part (that's part of the power of SwiftUI), but be sure to experiment with running on different simulators/Previews in Xcode to be sure.

Hints

1. Feel free to use `AspectVGrid` to lay out your cards if you'd like (since it generally lays out its `View`s in the way the Required Tasks proscribe). You are not required to do so, however. You can also modify it if you want (especially if you want to enforce a minimum card size).
2. Make sure you think clearly about what is in your Model, what is in your ViewModel and what is in your View. Always ask yourself "is this about how the Set game is *played* or about how it is *presented*?"
3. Your Model should clearly reveal the status of all the cards that are or ever have been in the deck.
4. In any complexity trade-off between View and ViewModel, make your View simpler.
5. Your Model doesn't really have complexity "trade-offs" because it is just trying to present a UI-independent programming interface that plays the game of Set as elegantly as possible. The ViewModel has to adapt to your Model's design (if your Model design is a good one, this shouldn't be too difficult for your ViewModel).
6. Don't forget that the View is just always a reflection of the Model. This is "reactive," "declarative" UI programming. The Model changes and the View is just *declared* to look like something completely based on the current state of the Model (accessed by the View through the ViewModel of course). Try to break free from the "imperative" model of programming you've probably grown up with (i.e. you call a function and something happens and then you call another function and something else happens, etc.). That's not how we do UI in SwiftUI.
7. It'd probably be good MVVM design not to hardwire "display-oriented" things like colors or even shape and shading names into the names of things in your Model. Imagine having themes for your Set game just as you did for Memorize. Remember that your Model knows little to nothing about how the game is going to be *presented* to the user. "Penguin Set" anyone?
8. A fairly simple way to draw the cards is to draw the symbols using an aspect ratio that is 3 times the aspect ratio of the cards. In other words, if your card aspect ratio is 2/3, then the aspect ratio of each symbol would be 2/1 (twice as wide as it is high). A `GeometryReader` to might be useful to figure out what the card's aspect ratio is when you are drawing the symbols on a card. Be careful about the effects of padding and stack spacing on this though.
9. Be careful to test your "end game" (i.e. when the deck runs out). To make testing this easier, maybe you make any 3 cards match in testing mode—that way you can get to the end of the game quickly. Or test with a partial deck.
10. Don't forget to put proper access control on all your `var`s and `func`s.

11. We are going to be covering animation in lecture next week and thus before this assignment is due. Assignment 4 is going to have you adding animation to your Set game. Finish a non-animated version of Set and submit it as A3 before moving onto an animated version for A4.

12. Remember that you can turn a computed `var` (or a `func`) that returns `some View` into a "ViewBuilder" by putting `@ViewBuilder` in front of it. This can be convenient if you just want a function that uses `if-else` or `switch` to pick from a list of `View`s.

13. You might also be tempted to return `some Shape` from a function. That is rarely done because there's *no such thing* as a "@ShapeBuilder" (i.e. something like `@ViewBuilder` for `Shape`s). You can, however, accept `some Shape` as an *argument* to a function. Example: `func applyShading(to shape: some Shape) -> some View`.

14. Swift has the type `Bool` built into it. A `Bool` is a variable with two states (`true` or `false`). Unfortunately, Swift has no built-in type for a variable that has *three* states. You might consider inventing such a thing because a Set game has an awful lot of instances of things with three states.

15. Your custom `Shape` (the diamond) will probably not be able to do `strokeBorder` (because that modifier only works on `InsettableShapes`). Just use `stroke` (hopefully your symbols should be nowhere near the edge of your card).

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. All the things from assignments 1 and 2, but *from scratch* this time
2. Access Control
3. `Shape`
4. `GeometryReader`
5. `enum`
6. 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.

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. Draw the actual squiggle instead of using a rectangle.
2. Draw the actual striped "shading" instead of using a semi-transparent color.
3. Keep score somehow in your Set game. You can decide what sort of scoring would make the most sense.
4. Give higher scores to players who choose matching Sets faster (i.e. incorporate a time component into your scoring system).
5. Figure out how to penalize players who chose Deal 3 More Cards when a Set was actually available to be chosen.
6. Add a "cheat" button to your UI.
7. Support two players. No need to go overboard here. Maybe just a button for each user (one upside-down at the top of the screen maybe?) to claim that they see a Set on the board. Then that player gets a (fairly short) amount of time to actually choose the Set or the other person gets as much time as they want to try to find a Set (or maybe they get a longer, but not unlimited amount of time?). Maybe hitting "Deal 3 More Cards" by one user gives the other some medium amount of time to choose a Set without penalty? You will need to figure out how to use `Timer` to do these time-limited things.
8. Can you think of a way to make your application work for people who are color-blind? If you tackle this Extra Credit, make it so that "color-blind mode" is on only if some `Bool` somewhere is set to `true` (and submit your application with it in the `false` state). In other words, you must still satisfy the Required Tasks and they specifically ask you to use 3 distinct colors. Some way to change the value of this `Bool` in the UI is not required, but you can include it if you want.

Assignment IV: Animated Set

Objective

The goal of this assignment is to understand the animation mechanisms described in lecture this week by adding animation to your Set assignment from last week.

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 before lecture 11.

Materials

- You will need your code from last week's Set assignment.
- You can use any of the code from lecture as well.

Required Tasks

1. Your assignment this week must still play a solo game of Set.
2. In this version, though, when there is a match showing and the user chooses a card, do not *replace* the matched cards, instead, *remove* (aka discard) them (leaving fewer cards showing in the game).
3. Add a "deck" and a "discard pile" to your UI. They can be any size you want and you can put them anywhere you want on screen, but they should not be part of your main grid of cards and they should each look like a stack of cards (for example, they should have the same aspect ratio as the cards that are in play).
4. The deck should contain all the not-yet-dealt cards in the game. They should be "face down" (i.e. you should not be able to see the symbols on them).
5. The discard pile should contain all the cards that have been discarded from the game (i.e. the cards that were discarded because they matched). These cards should be face up (i.e. you *should* be able to see the symbols on the last discarded card). Obviously the discard pile is empty when your game starts.
6. Any time matched cards are discarded, they should be animated to "fly" to the discard pile.
7. You don't need your "Deal 3 More Cards" button any more. Instead, *tapping on the deck* should deal 3 more cards.
8. Whenever more cards are dealt into the game for any reason (except when you first launch your application), their appearance should be animated by "flying them" from the deck into place. Once you get this working cleanly for dealing out 3 more cards, it should "just work" when you hit the "new game" button.
9. Note that dealing 3 more cards (by tapping on the deck) when a match is showing on the board still should replace those matched cards and that those matched cards would be flying to the discard pile at the same time as the 3 new cards are flying from the deck (see Extra Credit for even more on this). This Required Task might not take any extra effort if you implemented the above Required Tasks cleanly.
10. All the card repositioning and resizing that was required by Required Tasks 2 and 3 in last week's assignment must now be animated. If your cards from last week never changed their size or position as cards were dealt or discarded, then fix that this week so that they do.
11. When a match occurs, use some animation (your choice) to draw attention to the match.
12. When a mismatch occurs, use some animation (your choice) to draw attention to the mismatch. This animation must be very noticeably different from the animation used to show a match (obviously).

13. Add a shuffle button to your UI and animate the shuffling.

Hints

1. Your deck and discard pile have to look like a stack of cards. It's okay if they look like an extremely neatly stacked pile of cards (like Memorize's stack of undealt cards did).
2. You will start to see the importance of properly separating code out between View, ViewModel and Model when you do animation. Remember, changes to the game should be happening in the Model only and the View should just be reflecting those changes. Animation is just causing these changes to appear over a brief period of time instead of instantly (as was happening in A3, assuming you had no animation there).
3. You likely will want to shuffle only already-dealt and not-yet-discarded cards or you might get unexpected results (depending on how you've implemented the rest of your code).
4. Remember that `matchedGeometryEffect` doesn't really work properly in the Preview. Use the simulator.
5. The animations for match and mismatch do not necessarily have to repeat forever like we did in Memorize (though that could be cool depending on the animation) and it does not necessarily have to involve the symbols (though it's very easy to imagine the symbols being involved in these animations).
6. If you do use a `repeatForever` animation for your mismatches, be careful to have your implicit `.animation` `ViewModifier` go back to specifying the `.default` animation when the card returns to *not* being involved in a mismatch. Otherwise your `repeatForever` will, indeed, repeat forever which you obviously don't want in that case.
7. The "back" of a card can look like whatever you want (except that it can't give away what's on the front of the card, of course!).
8. There is no requirement for cards to "flip over". See Extra Credit.
9. None of the Required Tasks require you to create your own `ViewModifier`, but you'd probably learn a lot by doing so if you can fit it into your solution.
10. Since you'll be using `matchedGeometryEffect` to transition your cards in and out of existence (as they pass from deck to playing field to discard pile), you probably won't need to use `.transition` for that but depending on how you animate your matched and unmatched cards, it might be something you would want to use there.

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. Animation

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 not 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. Have your deck and/or discard pile be either "sloppy" (i.e. not a perfectly neat stack) or show the first few cards slightly offset (so that it looks more like a stack).
2. When a new game starts, animate the dealing out of the initial 12 cards.
3. When you deal 3 more cards and there is a match showing, start animating the matched cards flying to the discard pile *before* the animation of the 3 new cards flying in from the deck starts. In other words, give the user a better impression of "I just replaced these 3 cards for you" by *delaying* the dealing animation a short bit in this scenario. The animations can still overlap, but delaying the dealing one just a little bit can result in a pleasing effect.
4. Make the cards that you deal out flip from face down (as they are in the deck) to face up (as they are once they are in play). You can use/modify the `.cardify` `ViewModifier` from lecture if you want.
5. Add any other animation you can think of that would make sense.

Assignment V: Emoji Art

Objective

The goal of this assignment is to learn about how to deal with multi-touch gestures.

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 13.

Materials

- Start your work from the version of EmojiArt from Lecture 11.

Required Tasks

1. Do not break anything that is working in Emoji Art as of Lecture 11 as part of your solution to this assignment.
2. Support the selection of one or more of the emojis which have been dragged into your Emoji Art document (i.e. you're selecting the emojis in the document, *not* the ones in the palette at the bottom). You can show which emojis are selected in any way you'd like. The selection does not need to be persistent (in other words, restarting your application does not have to preserve the selection).
3. Tapping on an unselected emoji should select it.
4. Tapping on a selected emoji should unselect it.
5. Single-tapping on the background of your Emoji Art (i.e. single-tapping anywhere except on an emoji) should deselect *all* emoji.
6. Dragging a selected emoji should move the *entire selection* to follow the user's finger.
7. If the user makes a dragging gesture when there is no selection, pan the entire document (i.e., as it does in lecture). There is an exception to this if you do the Extra Credit.
8. If the user makes a pinching gesture anywhere in the Emoji Art document and there is a selection, all of the emojis in the selection should be scaled by the amount of the pinch.
9. If there is *no selection* at the time of a pinch, the entire document should be scaled (again, like it does in lecture).
10. Make it possible to delete emojis from the Emoji Art document. This Required Task is intentionally not saying what user-interface actions should cause this. Be creative and try to find a way to delete the emojis that feels comfortable and intuitive.

Hints

1. Get selection working first (by adding a tap gesture that adds/removes an emoji from the selection), then start in on the resizing and moving gestures.
2. Your selection is *not part of your Model*. It is purely a way of letting the user temporarily express which emoji they want to resize or move. Thus you will want to maintain your selection as temporary state in your UI code somewhere.
3. A `Set` might be a good data structure to store the set of selected emoji since there's no "order" to the selection (like an `Array` would imply) and a single emoji cannot be selected twice (again, as an `Array` would allow, but a `Set` does not).
4. If you do choose to use a `Set` for your selection, you might want to store `Emoji`'s `id`'s in there because, by definition, an `Identifiable`'s `id` is `Hashable` and anything put in a `Set` must also be `Hashable`.
5. Whatever view modifier(s) you are using on your `Text` to show whether it is selected need to be applied *before* you apply the `.position()` view modifier (because you want your selection-showing modifier to get positioned too).
6. Check out the view modifier called `.border` to see if it's something you want to use. Totally not required, but we didn't get a chance to mention its existence yet in lecture, so we're mentioning it here.
7. You might want to hide whatever UI you are using to show which emojis are selected when a gesture is in flight. It's unnecessary clutter in that circumstance.
8. When it comes to pinching, all pinches (whether zooming the document or resizing the selected emojis) are always recognized on the entire document (even though what happens on such a pinch depends on whether there's a selection at the time). So you can likely piggy-back your selection resizing on the existing `MagnificationGesture`. But you'll need to be sure that if there is something in the selection, your existing zoom view modifier is not paying any attention to `gestureZoom`. Also, you'll be doing something different in `.onEnded` depending on whether there is a selection or not.
9. We are using `scaleEffect` and `offset` to zoom and pan on our entire document. You can use the same strategy to resize and move your selected emojis (but only while gestures are in motion—the rest of the time your View is just drawing what it sees in the Model, as normal).
10. A general rule of thumb for gestures is that you are modifying `@GestureState` in `.updating` and you are modifying your Model (or some `@State`) in your `.onEnded`. Then you are using both that `@GestureState` and that Model/`@State` as arguments to your view modifiers throughout your View as appropriate.
11. You will not be able to piggy-back your emoji-moving onto the whole-document panning drag gesture as you can with zoom/resize. A drag gesture that starts on a selected emoji does a very different thing than a drag gesture that does not. So a new

`DragGesture` (attached to the view that is drawing each emoji) will be required to support moving the selection around.

12. You are allowed to pass `nil` to the `.gesture` view modifier. You might want to do this as part of a ternary expression which chooses whether to add a gesture to a view or not. For example, `Text().gesture(includeGesture ? theGesture : nil)`.
13. As always, we try to make "conditionality" in our SwiftUI views be embedded into the *arguments* to view modifiers rather than by using `if-else` statements (whether inside or outside `@ViewBuilder` constructs). This often takes the form of a ternary operator (`? :`) but may involve calling a `function` or getting the value of a computed `var` as part of the conditional decision-making.
14. This entire assignment can be done in fewer than 50 lines of code (possibly fewer than 30 lines of code in the View itself). So if you find yourself writing a lot more than that, maybe rethink your approach.

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. Gestures

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 lectures 1 and 2 works, but should not 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 one of these 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. Allow dragging *unselected* emoji separately. In other words, if the user drags an emoji that is part of the selection, move the entire selection (as required above). But if the user drags an emoji that is not part of the selection, then move only that emoji (and do not add it to the selection). You will find that this is a much more comfortable interface for placing `Emoji`s. Doing this will very likely require you to have a more sophisticated `@GestureState` for your drag gesture.

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.