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!

Do NOT name your app Set (or any other name that would be the same as the name of a `struct` or `class` in your application). For example, "Set Game" would probably be good.

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!

This assignment is a great opportunity to tackle a lot of the Extra Credit if you really want to challenge yourself.

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 on before you watch Lecture 7. You might find this assignment to be significantly more difficult than the first two assignments.

Materials

- You can use any of code from lecture (e.g. `Grid`).
- 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. When your game is run for the very first time, there should briefly be no cards showing, but as soon as it appears, it should immediately deal 12 cards by having them "fly in" from random off-screen locations.
3. As the game play progresses, use all the real estate on the screen in an efficient manner. Cards should get smaller (or larger) as more (or fewer) appear on-screen at the same time, while always using as much space as is available and still being "nicely arranged". `Grid` does all this and you are welcome to use it. All changes to locations and/or sizes of cards must be animated.
4. 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.
5. 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).
6. 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.
7. 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, animation, 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).
8. Support "deselection" by touching already-selected cards (but only if there are 1 or 2 cards (not 3) currently selected).

9. 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. the matched cards should fly away (animated) to random locations off-screen
   c. the replacement cards should fly in (animated) from other random off-screen locations (or from a "deck" view somewhere on screen, see Extra Credit)
   d. if the deck is empty then the space vacated by the matched cards (which cannot be replaced) should be made available to the remaining cards (i.e. they'll likely get bigger)
   e. if the touched card was *not* part of the matching Set, then select that card
10. 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).
11. 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 (with fly-in/fly-away as described above)
    b. or, if the selected cards do not make a Set (or if there are fewer than 3 cards selected, including none), fly in (i.e. animate the arrival of) 3 new cards to join the ones already on screen (and do not affect the selection)
    c. disable this button if the deck is empty
12. You also must have a "New Game" button that starts a new game (i.e. back to 12 randomly chosen cards). Cards should fly in and out when this happens as well.
13. To make your life a bit easier, you can replace the "squiggle" appearance in the Set game with a rectangle.
14. You must author your own `Shape` `struct` to do the diamond.
15. 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".
16. You can use any 3 colors as long as they are clearly distinguishable from each other.
17. You must use an `enum` as a meaningful part of your solution.
18. You must use a closure (i.e. a function as an argument) as a meaningful part of your solution.

19. 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 in Xcode to be sure.

Hints

1. Feel free to use `Grid` to lay out your cards if you'd like. You are not required to do so, however. You can also modify it if you want, but that's unlikely to be necessary.
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. Put some work into trying to design a coherent API for your Model.
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 efficiently 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. Along those lines, remember that any animation that is going on is just showing the user over time something *that has already happened*. Note also that you will not need to do any "animations of the things are are going to happen soon" like we had to do for our `Pie` in Memorize. All animations in this Set application are straightforwardly showing things that have already happened (like cards were dealt or cards were chosen or cards were matched and discarded).
8. It'd probably be good MVVM design not to hardwire "display-oriented" things like colors or even shape and shading names into 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?

9. Here's some help with the "flying" cards …

   9.a. All of the required tasks for "flying" cards are simply talking about the `Views` that represent a card "coming and going" from the UI.

   9.b. The "comings and goings" of `Views` in SwiftUI is animated using **transitions** (as explained in lecture).

   9.c. "Flying away (or in)" is just **moving**. Moving from where they are eventually going to be (or were) on screen to some random location off screen. So you'll want to have a simple function somewhere that calculates a random location off screen (where a card can fly to or from).

   9.d. There is a perfect transition for moving in SwiftUI called `AnyTransition.offset(CGSize)` (which animates the same `ViewModifier` that the `.offset` modifier in `View` uses).

   9.e. Make sure you are putting the `.transition` modifier on the `View` that is actually coming and going, not on some `Stack` inside that `View`, for example.

   9.f. Cards will not "come and go" when they should unless your Model clearly represents which cards are currently involved in the game. Has a card been dealt yet? Has a card already been matched and discard? Things like this must be in the Model somewhere or the UI won't know which cards are supposed to be on screen at a given moment.

   9.g. Your `View`, as always, just draws the state of the Model. So, for example, the array of `Identifiables` in the `Grid` (assuming you're using that to display your cards) is not going to include cards that haven't been dealt or that have been matched and discarded.

   9.h. Remember that transition animations don't happen unless you are explicitly animating. So you'll need any action the user takes that might cause flying cards to be explicitly animated using `withAnimation`.

   9.i. Don't forget about `.onAppear`. It can be very useful to get the game going when your UI first appears on-screen.

10. 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.

11. Don't forget to put proper access control on all your `var`s and `func`s.

12. This assignment does **not** require you add custom animation to a custom `ViewModifier` or `Shape` (i.e. the `Shape` you have to write for this assignment does not have any custom animation and you are not going to write your own `ViewModifier`).

13. When you use the type `some View` for a computed `var` or as the return type of a `function`, recall that you are asking the Swift compiler to look inside your code, figure out what type is *actually* being returned and effectively *replace* `some View` with that type (and, while it's at it, make sure that type conforms to `View`). This means that what is returned must be *determinable at compile-time* (not at run time). That's why the "conditionality" in building `Views` is done with **ViewBuilder** and why we can't do `if-else` or `switch` statements at the top-level of a function that returns `some View`. So remember to use ViewBuilder if you are returning `some View` and have `if-elses` inside there (ViewBuilder does not fully support `switch` (at least not as of this writing), so you will probably have to use `if-else`).
14. You might also be tempted to return `some Shape` from a function. That is pretty much never done because there's *no such thing* as a ShapeBuilder (i.e. something like ViewBuilder for `Shapes`). Instead, pass the information you need to `stroke` and/or `fill` your `Shape` and return `some View` from that function.
15. This is pretty much a "mid-term" in this course, so invest the time to be sure you really understand everything up until now. This application does not really require you to use anything that is completely new. If you are unable to get this app to work, you'll definitely have trouble with the final project.

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. Animation
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.

1. Have your cards fly in from a deck View located somewhere on screen. This is a little bit more difficult than flying to a random location because figuring out where some other View is will require some investigation on your part.
2. If you deal from a deck, have the cards be "face down" while on the deck and then flip over as they fly out to their locations when they are dealt.
3. Draw the actual squiggle instead of using a rectangle.
4. Draw the actual striped "shading" instead of using a semi-transparent color.
5. When cards match, provide some exciting animation. In other words, use animation to show that cards are matched in Required Task 7.
6. Make your fly away more exciting (maybe the cards spin as they fly away?).
7. Keep score somehow in your Set game. You can decide what sort of scoring would make the most sense.
8. Give higher scores to players who choose matching Sets faster (i.e. incorporate a time component into your scoring system).
9. Figure out how to penalize players who chose Deal 3 More Cards when a Set was actually available to be chosen.
10. Add a "cheat" button to your UI.
11. 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.