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 `Views`.

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 `Shapes`). 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.