Assignment III:

Graphical Set

Objective

You will use your knowledge of the card matching games from the past two weeks combined with your newfound skills at creating custom `UIView` subclasses and using `UICollectionView` to build better versions of Set and the Playing Card matching games that look (and in the Set case, play) more like the real thing. Finally, you will also need to familiarize yourself enough with Autolayout to make your UI autorotate properly.

There's lots of Extra Credit this week, so if it fits in your schedule, this is a good week to bank some.

This assignment is due by the start of lecture next Tuesday.

Be sure to check out the **Hints** section below!

Materials

- You can modify your existing Matchismo or you can start fresh, but you will almost certainly want to use your Model from Assignment 2 (though it will require some minor modification). Your Controllers for this week will bear some resemblance to last week's, but there's a lot new here.
- By now, you should know how the Set game works. We will implement more of the rules (for a solo game anyway) this time.
- You are welcome to use any code that was written during demonstrations in lecture.

Required Tasks

1. Create an application that plays both the Set game (single player) in one tab and the Playing Card matching game in another.
2. You must use polymorphism to design your Controllers for the two games (i.e. you must have a (probably abstract) base card game Controller class and your Set game and Playing Card game Controllers must be subclasses thereof).
3. For Set, 12 cards should be initially dealt and Set cards should always show what's on them even if they are technically "face down". For the Playing Card game deal 22 cards, all face down.
4. The user must then be able to choose matches just like in last week's assignment.
5. When a Set match is successfully chosen, the cards should be removed from the game (not just blanked out or grayed out, but actually removed from the user-interface and the remaining cards rearranged to use up the space that was freed up in the user-interface).
6. Set cards must have the "standard" Set look and feel (i.e. 1, 2 or 3 squiggles, diamonds or ovals that are solid, striped or unfilled, and are either green, red or purple). You must draw these using Core Graphics and/or `UIBezierPath`. You may not use images or attributed strings. Use the `PlayingCardView` from the in-class demo to draw your Playing Card game cards.
7. In the Set game (only), the user must always have the option somewhere in the UI of requesting 3 more cards to be dealt at any time if he or she is unable to locate a Set.
8. Automatically scroll to show any new cards when you add some in the Set game.
9. Do something sensible when no more cards are in the deck and the user requests more.
10. If there are more cards than will fit on the screen, simply allow the user to scroll down to see the rest of the cards. Pick a fixed (and reasonable) size for your cards and keep them that size for the whole game.
11. It is very important that you continue to have a "last flip status" UI and that it show not only matches and mismatches, but also which cards are currently selected (because there can be so many cards now that you have to scroll to get to all the cards in a match). A `UILabel` may no longer be sufficient for this UI.
12. The flip counter can be removed from the game. You must still show the score, however.
13. The user should still be able to abandon the game and start over with a fresh group of cards at any time (i.e. re-deal).

14. The game must work properly (and look good) in both Landscape and Portrait orientations on both the iPhone 4 and the iPhone 5. Use Autolayout to make this work (not struts and springs).

Hints

These hints are not required tasks. They are completely optional. Following them may make the assignment a little easier (no guarantees though!).

1. If you used inheritance to design your Controllers last week, the polymorphism part of the assignment will be a breeze. If you did not, then this is a skill you must master. Hopefully your OOP skills are such that you know what polymorphism is, but suffice it to say, you must have a base Controller class which encapsulates as much as possible that is conceptually in common between the Set game and the Playing Card game and this base class must have public API for the subclasses to override to provide game-specific behavior. Each of the two tabs in your application will have a Controller which is a (different) subclass of this shared base Controller class.
2. If you designed your Model to be highly specific to last week's assignment (maybe you specified specific color or shape names in the Model's API, for example, instead of just saying colors 1 2 3 and shapes 1 2 3), you'll have to go back and fix it. Hopefully this will give you an appreciation for designing reusable, extensible API. A lot of good API design is taking time to think "how might someone want to use this in the future?" It would not have been a stretch to think that someone would want to use different colors or use shapes other than triangle, circle and square (the latter especially since you knew those weren't the normal shapes and that we were just doing that because that's all `NSAttributedString` could do for us). It is understandable if you hardwired these "appearances" into your Model, by the way, especially if you have really only been asked to program for "homework" in CS classes (which usually ask you to do a specific thing and often don't care if your API is extensible). You will not be docked for doing that this time either, but hopefully the experience will stay with you when you go out into the "real world."
3. You will need to enhance your `CardMatchingGame`, but not that much ...
4. Since cards can now be added to an existing game, your `CardMatchingGame` will have to hold onto the deck it was created with, have public API to cause more cards to be put in play from that deck, and have public API to let anyone who's interested know how many cards are currently in play.
5. Matches now result in the removal of cards. One could make the argument for the responsibility for this removal being either in the `CardMatchingGame` (automatically remove matched cards) or in the Controller (remove cards involved in matches on the last flip) depending on whether you think it's part of the "game play" or is just a "visual thing" to remove the cards. The latter (having an API in your Model for deleting cards from the game and letting the Controller decide when to do this) turns out to be easier to implement (especially if you are doing the deletion animation extra credit), so you are welcome to choose this architecture.
6. The Extra Credit to animate the removal of cards might also benefit from a reverse `indexOfCard:` method (not in all solutions, but in some).

7. As always, you will get a lot from the in-class demonstrations from the last two lectures (in this case, custom `UIView` and `UICollectionView`). You are allowed to use the code from those (but if you want to learn this stuff, be sure you're understanding it as you incorporate it into your homework solution).

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.
- Project does not build without warnings.
- One or more items in the **Required Tasks** section was not satisfied.
- A fundamental concept was not understood.
- Code is 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, etc.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice." Xcode gives you those dashed blue guidelines so there should be no excuse for things not being lined up, etc. Get in the habit of building aesthetically balanced UIs from the start of this course.
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Incorrect or poor use of object-oriented design principles. For example, code should not be duplicated if it can be reused via inheritance or other object-oriented design methodologies.

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 SDK, but should not assume that they already know the (or a) solution to the problem.

Extra Credit

Here are a few ideas for some things you could do to get some more experience with developing for iOS. You need only pick any two of these (except you cannot pick "the first two and only the first two" since they are so easy) to get the maximum available extra credit this week. If you want to do more of them, go for it, but you will not earn additional credit for doing so.

1. Animate the removal of matched cards. `UICollectionView` will do this for you if you call the `deleteItemsAtIndexPaths:` method. Just remember that at any time you call `deleteItemsAtIndexPaths:`, the `UICollectionView`'s `dataSource` (your Controller on behalf of your Model) must be in the state that will exist after the deletion happens. Otherwise you will crash with an assertion in the delete (because it will delete the items and try to reset everything and the new state of the `UICollectionView` will not match its `dataSource`'s idea of things). You can animate the dealing of 3 new cards as well using `insertItemsAtIndexPaths:` (though that animation is just "dissolving in" the new cards ... deletion is far more exciting).

2. Let the user choose how many cards to deal in the Playing Card game. Set is standardized to 12 cards to start a game, but the Playing Card game is flexible.

3. Add a section (or sections) to your `UICollectionView` to show "found matches." In other words, the user can scroll down in the `UICollectionView` (below the game) and see all of the matches they've found so far in the current game. The actual cards should appear (perhaps miniaturized, perhaps not, up to you). You will likely want to create a new `UICollectionViewCell` with 2 or 3 instances of a custom `UIView` subclass (that you've already written) as subviews and maybe some nice adornment.

4. You could add better score-keeping to the Set part of this application if you can figure out an algorithm for calculating whether a Set exists in the current cards in play. Then you can penalize the user not only for mismatches, but for clicking the "deal 3 more cards" button if he or she missed a Set. You'd also know when the game was "over" (because the user would click on "deal 3 more cards" and there would be no more cards in the deck and no more Sets to choose).

5. Knowing how to find Sets in the remaining cards also would allow you to let the user cheat. Have a button that will show them a Set (if available). It's up to you how you want to show it, but maybe some little indicator (a star or something) on each of the 3 cards?

6. Make this a two player game. There's no need to go overboard here. Think of a simple UI and a straightforward implementation that make sense.