Assignment I: Matchismo

Objective

This assignment extends the card matching game Matchismo we started last week to get experience understanding MVC, modifying an MVC's View in Xcode, creating your own actions and outlets, interacting with `UILabel`, `UIButton` and other iOS 6 SDK elements, and generally getting more experience with Xcode and Objective-C.

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

Materials

- You will need to have completed last week's assignment before starting this one (at least the walkthrough portion of it).
- You will also need the lecture slides from Tuesday's lecture.
- The slides for all lectures can be found in the same place you found this document.

Required Tasks

1. Follow the detailed instructions in the lecture slides (separate document) to reproduce the latest version of Matchismo we built in lecture (i.e. the one with multiple cards) and run it in the iPhone (normal, non-Retina, non-iPhone 5) Simulator. Do not proceed to the next steps unless your card matching game functions as expected and builds without warnings or errors.
2. Add 4 cards to the game (for a total of 16).
3. Add a text label somewhere which desribes the results of the last flip. Examples: "Matched J♥ & J♠ for 4 points" or "6♦ and J♣ don't match! 2 point penalty!" and simply "Flipped up 8♦" if there is no match or mismatch.
4. Add a button called "Deal" which will re-deal all of the cards (i.e. start a new game). It should reset the score (and anything else in the UI that makes sense). In a real game, we'd probably want to ask the user if he or she is "sure" before aborting the game in process to re-deal, but for this assignment, you can assume the user always knows what he or she is doing when they hit this button.
5. Drag out a switch (`UISwitch`) or a segmented control (`UISegmentedControl`) into your View somewhere which controls whether the game matches two cards at a time or three cards at a time (i.e. it sets "2-card-match mode" vs. "3-card-match mode"). Give the user appropriate points depending on how difficult the match is to accomplish.
6. Disable the game play mode control (i.e. the `UISwitch` or `UISegmentedControl` from Required Task #5) when flipping starts and re-enable it when a re-deal happens (i.e. the Deal button is pressed).
7. Make the back of the card be an image (`UIImage`) rather than an Apple logo.

Hints

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

1. `NSString`'s `stringWithFormat:` method will be very helpful.
2. Don't forget that `NSString` constants start with `@`. Constants without out the `@` (e.g. `"hello"`) are `const char *` and are rarely used in iOS.
3. Think carefully about where the code to generate the strings in Required Task #3 above should go. Is it part of your Model, your View, or your Controller? Some combination thereof? Justify your decision in comments in your code. A lot of what this homework assignment is about is your understanding of MVC.
4. You might be interested in the `NSArray` method `componentsJoinedByString:`. It turns an array into a string (with the given string separating each component in the array). If you don't have a use for this, don't worry.
5. You will have to read the documentation for `UISwitch` and/or `UISegmentedControl` to figure out how to use them. A switch is probably (a little bit) easier to understand, but a segmented control is probably more appropriate to the task. Being able to figure a class out solely from its documentation is crucial to being a good iOS developer. That's what that Required Task is about.
6. The logic in your Model to will have to be configurable for the two different game play modes. And your `PlayingCard` class will also have to know how to match itself against two other cards (it already knows how to match itself against one other card).
7. You might have to make your cards a bit shorter to fit all of the UI required in this assignment.
8. You can feel free to adjust the scoring of 2-card-match mode if you want it to be consistent with your 3-card-match mode's scoring. In other words, consider the difficulty of matching 2 out of 2 cards of the same suit (medium) versus 2 out of 3 (easy) or 3 out of 3 (hard).
9. Setting the background image of a selectable `UIButton` is tricky to do directly in Xcode (i.e. in the Inspector) since if you have not set an image for the `Selected` state, then the `Normal` image will **also** appear when the button `isSelected` (which is probably not what you want). Setting the image for the appropriate state conditionally *in code* is probably simpler (it can be done in 2 lines of code in your `updateUI`).
10. You'll need this method below which creates an image object suitable for setting as the `UIButton`'s image for a given state (i.e., using `UIButton`'s `setImage:forState:` method). All you need to do to make this line of code work is to drag a `cardback.png` (`jpeg` also works) file into your project's Navigator:

```
UIImage *cardBackImage = [UIImage imageNamed:@"cardback.png"];
```

12. Depending on what the image you are trying to use for your card back looks like, you might want to check out `UIButton`'s `imageEdgeInsets` property to make your image fit inside the rounded rect of the button better. The C function `UIEdgeInsetsMake()` will make things a snap when it comes to setting the value of `imageEdgeInsets`. This might also make you more comfortable with C structs if you don't have a lot of experience with them (since `UIEdgeInsets` is a C struct).
13. Economy is valuable in coding: the easiest way to ensure a bug-free line of code is not to write the line of code at all. This assignment requires more lines of code than last week, but still not an excessive amount. So if you find yourself writing many dozens of lines of code, you are on the wrong track.

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).

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 is an idea for something you could do to get some more experience with the SDK at this point in the game.

Add a `UISlider` to your UI which travels through the history of the currently-being-played game's flips and display it to the user (moving the slider will modify the contents of the text label you created for Required Task #3 to show its state over the course of the game). When you are displaying past flips, you probably want the text label to be grayed out (with alpha) or something so it's clear that it's "the past." Also, you probably don't want that text label from Required Task #3 to ever be blank (except at the very start of the game, of course). And every time a new flip happens, you probably want to "jump to the present" in the slider. Implementing this extra credit item will require you to familiarize yourself with `UISlider`'s API and to add a data structure to your Controller to keep track of the history. It can be implemented in fewer than a dozen lines of code.