Assignment IV: Graphical Set

Objective

You will use your knowledge of the card matching games from the past few weeks combined with your newfound skills at creating custom `UIView` subclasses and using `UIDynamicAnimator` 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.

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 3 (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.
- A completely optional Grid class is available (see Hints).
- You are welcome to use any code that was written during demonstrations in lecture.

Required Tasks

1. Your application this week is still required to play both the Set and Playing Card matching games (in separate tabs) and must show the score and allow re-deals, but you can remove the UI for showing the result of the last card choice as well as the History MVC added last week.
2. Cards must have a "standard" look and feel (i.e. for Set, 1, 2 or 3 squiggles, diamonds or ovals that are solid, striped or unfilled, and are either green, red or purple; for Playing Cards, pips and faces). You must draw these using `UIBezierPath` and Core Graphics functions. You may not use images or attributed strings for Set cards. The drawings on the card must scale appropriately to the card's `bounds`. You can use the `PlayingCardView` from the in-class demo to draw your Playing Card game cards.
3. When a Set match is successfully chosen, the matching cards should now be removed from the game (not just blanked out or grayed out, but removed from the UI entirely).
4. As in a real game of Set, the user should start with 12 cards and then have the option of requesting 3 more cards to be dealt at any time if he or she is unable to locate a Set. Do something sensible when there are no more cards in the deck.
5. Use a flip transition to animate choosing cards in the Playing Card game (so that the cards look like they are being flipped over when chosen).
6. The arrival and departure of cards must be animated and, as they come and go, you must automatically adjust the layout of the cards in your user-interface (i.e. their size and position) to efficiently use the real estate on screen (i.e. don't waste space) and make them all fit.
7. Animate re-deals.
8. 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 as much as possible to make this work. Positioning the cards themselves will require some additional work (although you are probably already doing this work for some of the Required Tasks above). None of your code should be specific to any given screen width or height or orientation (i.e. no "if landscape then" or "if width/height ... then" code). The bottom line of this task is that your MVCs' Views should look good in any reasonably-sized `bounds` rectangle.
9. The movement of the elements inside your UI that occur in response to device rotation must be animated (Autolayout already animates any changes it makes for you, but any other layout changes that you do in code must be animated by you).
10. Use a `UIDynamicAnimator` to allow the cards in either game to be gathered up into a pile via a pinch gesture. Once gathered, the stack of gathered cards must be able to be moved around (via a pan gesture). Tapping on the stack will return the cards to their normal positions (animated, of course) unharmed.

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. Share as much code and mechanism as possible through this base 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. Obviously you will not being using `UIButton`s to display your cards this week. So you can remove your `cardButtons` outlet collection. Likely you'll want to replace it with a mutable array (populated in code rather than ctrl-dragged) of your new card views.
4. Since cards will be coming and going in this version, you can't just drag them out and place them like last week's card buttons. You must calculate where to put them as they come and go. A `Grid` class has been provided to help you calculate where to put cards given a certain amount of space, minimum number of cards and specified card aspect ratio. You can modify it to suit your needs or not use it at all if you wish.
5. Also, you'll very likely want to drag a generic `UIView` into your storyboard(s) simply to serve as the boundary area for your cards (unless you plan to have your cards fill up the entire View of your MVC, but then where will your other UI elements go?). This `UIView` does not need to be subclassed because it does nothing except specify an area on screen and serve as the `superview` for your card views.

6. You will need to enhance your `CardMatchingGame`, but not that much. 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. But that's it. It's logic should not have to change.
7. Matches now result in the removal of cards. This is much simpler to implement if you make the architectural decision that this is merely a visual thing rather than part of the "game play." In other words, it is strongly recommended that you not have API in your Model to delete cards. "Remove" them on screen simply by not showing `isMatched` cards (but still obey Required Task 6: you cannot just leave "holes" in the layout of cards).
8. Your concrete card game view controller subclasses can be implement with very little code (less than two dozen lines each is quite possible). If you are finding yourself writing tons of code in these concrete subclasses, then you may not be designing the API of your abstract super class effectively enough.
9. Note that the third kind of "filling" in a Set card is striping, not shading.
10. It is not a Required Task to remove unused code (e.g. attributed string code, history-displaying code, etc.), but it is generally recommended to do so.
11. There are numerous ways to animate the coming and goings of cards. Creativity will be rewarded! On the other hand, be sure to check the updated Evaluation criteria this week (i.e. animation quality is part of it).
12. The animation of re-deal should make it clear to the user that the current cards are being discarded and new ones being dealt. For example, just flipping the cards that are face up back down would be insufficient (that might just look like you are letting them start the same game again with the same set of cards).
13. If you have game-specific animation (i.e. specific to the Playing Card game or to the Set game), you can still put most of your code in your abstract superclass. For example, if your subclass will be doing some animation, it can either let the superclass know how long it is going to take or call the superclass back when the animation is done or the superclass can simply wait for dynamic animation to settle back down (if the subclass is only using dynamic animation).
14. The "animation in response to device rotation" Required Task might well take exactly zero extra code. If you've implemented the rest of the Required Tasks in a generic way (i.e. your UI is always appropriate to its `bounds` and any time your UI changes anything, it animates that change) this should come for free.
15. When you are implementing the pinching of the cards, remember that an attachment behavior's `length` and `anchorPoint` can be modified at any time and the animation will instantly adapt.

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. More experience subclassing with Controllers.
2. Making code generic and reusable using OOP versus copying/pasting it.
3. More about using methods in the View Controller lifecycle.
4. Creating custom `UIView` subclasses and implementing `drawRect:`.
5. Understanding the view hierarchy.
6. Using a generic `UIView` simply as a "container" of other views.
7. Creating views in code (as opposed to dragging them out in Xcode).
8. Drawing with Core Graphics and `UIBezierPath`.
9. Attaching gesture handlers to views (in code and via ctrl-dragging in Xcode).
10. Handling gestures.
11. Using Autolayout to react to `bounds` changes (e.g. on device rotation).
12. Animating with `UIView` class methods.
13. Understanding the delay between setting properties and appearance in animation.
14. Animating with `UIDynamicAnimator`.

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.
- Animation is abrupt or otherwise not pleasing to the eye. Animation's purpose is to provide visual appeal and communicate change to the user. It should not be a distraction, but rather an integral part of the aesthetic design of your application.

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.

1. 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).

2. 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?

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

4. Think of some other way(s) to use animation in your application and implement it.