Assignment III: Graphing Calculator

Objective

You will enhance your Calculator to create a graph of the "program" the user has entered which can be zoomed in on and panned around. Your app will now work not only on iPhones, but on iPads as well.

Materials

- You will need to have successfully completed Assignment 2. This assignment builds on that. You can try to modify your existing program or create a new project (and reuse the classes you wrote by dragging them into the new project). In any case, be sure to save a copy of last week's work before you start.
- This AxesDrawer class will likely be very useful!

Required Tasks

1. You must begin this assignment with your Assignment 2 code, not with any in-class demo code that has been posted. Learning to create new MVCs and segues requires experiencing it, not copy/pasting it or editing an existing storyboard that already has segues in it.
2. Rename the `ViewController` class you've been working on in Assignments 1 and 2 to be `CalculatorViewController`.
3. Add a new button to your calculator's user-interface which segues to a new MVC which graphs the `program` that is in the `CalculatorBrain` at the time the button was touched using the memory location `M` as the independent variable. For example, if the `CalculatorBrain` contains `sin(M)`, you'd draw a sine wave. Subsequent input to the Calculator must have no effect on the graphing MVC (until the graphing button is touched again). Ignore user attempts to graph if `isPartialResult` is true at the time.
4. Neither of your MVCs in this assignment is allowed to have `CalculatorBrain` appear anywhere in its non-`private` API (the I in API stands for *interface*, not *implementation*).
5. On iPad and in landscape on iPhone 6+ devices, the graph must be (or be able to be) on screen at the same time as your existing Calculator's user-interface (i.e. in a split view). On other iPhones the graph should "push" onto the screen via a navigation controller.
6. Anytime a graph is on screen, a description of what it is being drawn should also be shown on screen somewhere sensible, e.g., if `sin(M)` is what is being graphed, then the string "`sin(M)`" should be on screen somewhere.
7. As part of your implementation, you are required to write a *generic* `x` vs. `y` graphing `UIView`. In other words, the `UIView` that does the graphing should be designed in such a way that it is independent of the Calculator (and could be reused in some other completely different application that wanted to draw an `x` vs. `y` graph).
8. The graphing view must not own (i.e. store) the data it is graphing, even temporarily. It must ask for the data as it needs it. Your graphing view graphs an `x` vs. `y` *function*, it does not graph an array of points, so don't pass it an array of points.
9. Your graphing calculator should do something sensible when graphing discontinuous functions (for example. it should only try to draw lines to or from points whose `y` value `.isNormal` or `.isZero`). To make things simpler on this front, it's okay if your graphing view improperly graphs a function that rapidly goes through a huge swing in value across a single pixel by drawing an almost vertical line between those two points even if the function is actually probably discontinuous there (e.g. `tan(x)`). It would be cool to try to detect this case within some tolerance, though, and not draw that vertical line (up to you).

10. Your graphing view must be `@IBDesignable` and its scale must be `@IBInspectable`. The graphing view's axes should appear in the storyboard at the inspected scale.
11. Your graphing view must support the following three gestures:
    a. Pinching (zooms the entire graph, including the axes, in or out on the graph)
    b. Panning (moves the entire graph, including the axes, to follow the touch around)
    c. Double-tapping (moves the origin of the graph to the point of the double tap)

Hints

1. Forgetting to set the class of a `UIViewController` or a custom `UIView` in the Identity Inspector in Xcode is a common error. You'll need to do this when you rename `ViewController` to `CalculatorViewController` and for both the new `UIViewController` and the new `UIView` that you are creating in this assignment.
2. To make the drawing of the graph much easier, a class which can draw a graph's axes in the current drawing context is provided (`AxesDrawer`). Notice that this class's drawing method (`drawAxesInRect`) takes the `bounds` to draw in and two other arguments: `origin` and `pointsPerUnit` (this is essentially the "scale" of the graph). You will very likely want to mimic this (i.e. having `var`s for origin and scale) in your generic graphing view.
3. Your `CalculatorBrain` should not need to be touched for this assignment.
4. Here's a suggested order of attack … Get your existing `CalculatorViewController` working inside a split view controller and navigation controller structure with a graph button that segues to a new, blank MVC (at first) with an appropriate `UIViewController` subclass. Add your generic graphing view to this new MVC. Get the graphing view at least drawing the axes. Add gestures. Finally, get your new MVC to graph the `program` that is in your main MVC at the time the graph button is touched. You don't have to do it in this order by any means, but it might help you organize your work.
5. When your existing `CalculatorViewController` is embedded in a `UINavigationController`, the frames of all the buttons may not show correctly in the storyboard (and Xcode will complain with build warnings). It will be fine when you run. Sometimes restarting Xcode will clear these warnings. If not, then try using the button in the lower right corner of your storyboard: with the calculator scene selected, choose Update Frames under All Views in Calculator View Controller. It should move the buttons to where they should be according to your stack views and your "pin to the edges" layout constraints. If not, and if restarting Xcode doesn't fix it, ignore these warnings.
6. When you put your graphing view into your new MVC, you can use Reset to Suggested Constraints in that same menu to set its "pin to the edges" constraints as long as you have used the dashed blue lines to put your graphing view in the right spot in your new MVC's scene. If you mess up doing any of these constraints operations, remember that Xcode has undo.
7. The `UIViewController` subclass for your new MVC (the one that graphs what is in the Calculator) and the generic graphing `UIView` subclass are the only new classes you should have to write from scratch for this assignment. If you think you need to be writing other classes, you might be overdoing it.

8. As we've learned, a function is a first-class-citizen type in Swift. Thus, it is perfectly legal to have an `Optional` function if you want.
9. Make sure you think clearly about what your new MVC's Model should be.
10. Don't freak out when you drag out a Split View Controller and it brings along all kinds of other view controllers along with it. It's just Xcode trying to be helpful. You can safely delete those and use ctrl-drag to wire up your MVC's (inside navigation controllers) in their places.
11. It'd be nice for the origin of your graph to default to the center of the `UIView`. But be careful where/when you calculate this because your `UIView`'s `bounds` are not set until it is laid out for the device it is on. You can be certain your `bounds` are set in your `drawRect` of course, but be careful not to **re**-set the origin if it's already been set by someone.
12. A good place for your new MVC to set itself up to work with your generic graphing view is using a property observer on its outlet property to the graphing view.
13. Don't overcomplicate your `drawRect`. Simply iterate over every pixel (not point) across the width of your view and draw a line to (or just "move to" if the last datapoint was not valid) the next datapoint you get (if it is valid).
14. The coordinate system you are drawing in inside your `drawRect` is not the same as the coordinates your data is in (because, for example, your drawing coordinates have the origin in the upper left, but the data's origin is probably somewhere else in the view; not to mention scalability). Be clear in your mind as you write your code which of these two coordinate systems a `var` or an argument to a function is (and should be) in.
15. The `AxesDrawer` knows how to draw on pixel (not point) boundaries (like your `drawRect` should), but only if you tell it the `contentScaleFactor` of the drawing context you are drawing into.
16. Don't forget to use property observing (`didSet`) to cause your view to note that it needs to redisplay itself when a property that affects how it looks gets changed.
17. Make sure you set your `UIViewContentMode` properly (this can be done in the storyboard).
18. Your gestures will probably be *handled* by the graphing view, but will probably want to be "turned on" by your Controller. For this reason, the methods that handle the gestures shouldn't be `private` in your graphing view.
19. Remember that when specifying the action that is going to handle a gesture as part of creating a gesture recognizer, if that handler takes an argument it must properly be specified as such.
20. This assignment will probably require a bit more code than your first two assignments did, but it still can be done in well under 100 lines of code.

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. Understanding MVC boundaries
2. Creating a new subclass of `UIViewController`
3. Universal Application (i.e. different UIs on iPad and iPhone in the same application)
4. Split View Controller
5. Navigation Controller
6. Segues
7. Property List
8. Subclassing `UIView`
9. `UIViewContentMode.Redraw`
10. Drawing with `UIBezierPath` and/or Core Graphics
11. `CGFloat/CGPoint/CGSize/CGRect`
12. Gestures
13. `contentScaleFactor` (pixels vs. points)

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 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.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice."
- Public and private API is not properly delineated.

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

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 graphing button in your main Calculator scene reflect whether or not it is currently possible to graph what has been entered so far (i.e. whether there is a partial result or not). You could just disable it, but maybe a different graphic or something? This is a very easy task, so don't expect a lot of extra credit for it!
2. Preserve origin and scale between launchings of the application. Where should this be done to best respect MVC, do you think? There's no "right answer" to this one. It's subtle.
3. Upon rotation (or any `bounds` change), maintain the origin of your graph with respect to the center of your graphing view rather than with respect to the upper left corner.
4. Figure out how to use Instruments to analyze the performance of panning and pinching in your graphing view. What makes dragging the graph around so sluggish? Explain in comments in your code what you found and what you might do about it.
5. Use the information you found above to improve panning performance. Do NOT turn your code into a mess to do this. Your solution should be simple and elegant. There is a strong temptation when optimizing to sacrifice readability or to violate MVC boundaries, but you are NOT allowed to do that for this Extra Credit!
6. When your application first launches, have it show the last graph it was showing (rather than coming up blank). You could reset the Calculator upon relaunch to the last state it was in as well (which may or may not be the same thing as what the graph was showing). Be careful not to violate MVC in your solution, though (each MVC its its own independent world).