Assignment III: Graphing Calculator

Objective

The goal of this assignment is to reuse your `CalculatorBrain` and `CalculatorViewController` objects to build a Graphing Calculator for iPhone and iPad.

By doing this, you will gain experience creating your own custom view, building another `UIViewController`, using a protocol to delegate responsibility from one object to another, creating a `UINavigationController` and `UISplitViewController`, creating a segue in a storyboard, handling gestures, and making a Universal application that runs on both iPad and iPhone/iPod Touch.

Be sure to check out the Hints section below in full before you start!

Also, check out the latest version of the Evaluation section to make sure you understand what you are going to be evaluated on with this (and future) assignments.

Materials

- If you successfully accomplished last week's assignment, then you have all the materials you need for this week's. You can try to modify your existing program or create a new project from scratch but, in any case, be sure to save a copy of last week's work before you start.

Required Tasks

1. When your application is run on the iPhone, it must present the user-interface of your calculator from Assignment 2 inside a `UINavigationController`.
2. The only variable button your calculator's user-interface should present is `x` (so remove any others you added in Assignment 2 and you can remove your Test buttons and the `UILabel` which shows the value of the variables being used in the display).
3. Add a Graph button to your calculator's user-interface that, when pressed, segues to a new MVC (that you will have to write) by pushing it onto the `UINavigationController`'s stack (on the iPhone). The new MVC graphs whatever `program` was in the calculator when the button was pressed. To draw the graph you will iterate over all the pixels in your view horizontally (x) and use `+runProgram:usingVariableValues:` to get the corresponding vertical (y) value. You will, of course, have to convert to/from your view's coordinate system from/to a reasonable graph coordinate system. You will need a `scale` and `origin` to do this coordinate system conversion. If the user has not already chosen a `scale` and `origin` for the graph (see Required Tasks 7 & 8 below), pick a reasonable starting `scale` and `origin`.
4. Anytime a graph is on screen, the description of the `program` used to draw it (e.g. the result of your `+descriptionOfProgram:` method) should also be shown on screen somewhere sensible. This might be a different place on the iPhone versus the iPad.
5. To implement your new MVC, you must write a custom graphing View which must display the axes of the graph in addition to the plot of the `program`. Code will be provided on the class website which will draw axes with an `origin` at a given point and with a given `scale`, so you will not have to write the Core Graphics code for the axes, only for the graphing of the `program` itself. You probably will want to examine the provided axes-drawing code to understand how the scaling works before you do this or any of the following Required Tasks.
6. Your graphing `UIView` must be generic and reusable (i.e. it should be a generic x-y graphing class and know nothing about a `CalculatorBrain`). Use a protocol to get the graphing view's data because Views should not own their data.
7. Support the following gestures inside your graphing view:
   - Pinching (adjusts your view's `scale`).
   - Panning (moves the entire graph, including axes, to follow the touch around).
   - Triple-tapping (moves the `origin` of the graph to the point of the triple-tap).
8. Make your Calculator work on the iPad too (in the same application, i.e., a Universal application) by having two storyboards, one for the iPhone (described above) and one for the iPad that uses a `UISplitViewController` to present your old Calculator MVC on the left of the screen (or in a popover) and your new graphing MVC on the right.

Since both MVCs will be on the screen at the same time, you will not need a segue; instead, you will have to get a pointer to the other MVC using `UISplitViewController` API. You should only update the right side of the split view when the Graph button is pressed (not continuously as the brain is modified).

9. You do not have to support rotation on all devices, but on the devices where you do support it, the user-interface should look good in all cases (so get your struts and springs right in your storyboard(s)).
10. Once the user picks a `scale` by pinching or a new `origin` by panning or tapping, the new value should be stored in `NSUserDefaults` so that the `scale` and `origin` persist as new `program`s are graphed and even through relaunchings of your program.

Hints

1. Get the first two Required Tasks working (i.e. your old calculator UI in a navigation controller on the iPhone with a few things removed) before you start working on your new MVC.
2. Whenever you drag files into a project (e.g. the `AxesDrawer` class's `.m` and `.h`), **make sure you click the box that says "Copy items into destination group's folder (if needed)."**
3. As a reminder, the way to create new things (including your new graphing MVC's `UIViewController` subclass and your new graph `UIView` subclass and your iPad storyboard) is to use the menu item `New File ...` in the `File` menu.
4. Your new `UIViewController` and your new `UIView` are the only new classes you should have to write from scratch for this assignment.
5. Your new Controller (the graphing one) is just like any other MVC Controller: it's going to want to have a Model (what is the Model for this new Controller, do you think?) and outlets into its View. Before you go on to do anything else in this assignment, add a `@property` to this Controller for its Model (if you proceed without really understanding what this Controller's Model is, you might have trouble implementing the rest of the assignment). Don't get this Controller's Model confused with your `CalculatorViewController`'s Model. THEY ARE DIFFERENT. And don't over-think this. Your new Controller's Model is near at hand!
6. Don't forget to set the class of your new `UIViewController` with the Identity Inspector after you drag out a generic one from the Object Library into your storyboard(s). Ditto the class of your graphing `UIView`.
7. It might be a good strategy to implement the `drawRect:` in your custom graphing `UIView` *after* you have your MVCs all wired up in your storyboard(s) so you can easily test it once you start working on it.
8. You'll need an `IBOutlet` in your new Controller to point to your graphing view for a variety of reasons. Your graphing view needs its data source delegate set, gesture recognizers added to it, and it needs to told that it needs to redraw itself when its data source delegate would provide different data if asked again.
9. You can convert an existing iPhone-only project to a Universal one in the Summary tab of your Calculator Target (click on the topmost icon in your Navigator, then click on the Calculator under Targets). This is also where you set which storyboard is used on which platform.
10. When you drag a split view controller out of the Object Library into a storyboard, it (tries to be helpful and) gives you a navigation controller with a table view in the left side of the split view controller by default. You don't need them, so you can just delete those controllers and then drag a new `UIViewController` from the Object Library (set

its class to your `CalculatorViewController`), then use ctrl-drag to connect it as your split view's master.

11. You will have to copy/paste your Calculator's UI from one storyboard to the other when you go to create your iPad version, and you may have to wire the buttons and labels back up (copying and pasting views between storyboards does NOT preserve the target/action or outlet connections, but copying and pasting entire view controllers does). The #1 reason your iPad storyboard will not work is because you forget to wire up one of its outlets (e.g. an outlet to your graph view or display or a target/action message from one of your many buttons).

12. When you go to implement your graphing view's `drawRect:`, the first thing you'll want to do is to use the helper code provided to draw the axes (just drag it into your project, but note Hint 2 above). To make it easy on yourself, make sure you use the same scaling approach as the helper class does (it's documented in that class's header file). All you need to do to use this helper class is set up the graphics state you want (colors, etc.), then call the lone class method in the helper class from your `drawRect:`. It will automatically use the current drawing context.

13. The implementation of your custom `drawRect:` is deceptively simple. You just need to iterate over every pixel across the width of the area in your view you want to draw and convert (considering `scale` and `origin`) the horizontal position from your custom `UIView`'s coordinate space to the coordinate space of the graph you are drawing (this converted position will be the x-axis of the data you are graphing), then ask your data source delegate to provide the y-axis position for that x value, then convert that y-axis value back to your `UIView`'s coordinates and then use Core Graphics to draw to the next point.

14. Your graphing `UIView`'s `scale` and `origin` are **not** its data. They are properties of the `UIView` that control *how its data is drawn*. Therefore they do **not** need to be delegated. It is not unreasonable in certain circumstances for a View to delegate some of its more complicated "how to draw me" properties (in addition to delegating the source of any of its data), but it is not necessary in this case.

15. It's not exactly the right thing to do to draw a line from point to point (especially if you're zoomed way out or have a discontinuous expression), but we'll accept it for this assignment since it's simple to implement and it's "the right thing" a lot of the time. ;-) Check out the Extra Credit on this front below.

16. To take advantage of the very high resolution Retina displays on the iPhone4 and new iPod Touch devices, be sure to iterate over pixels, not points, as you move along the x-axis in your graphing view. Remember `UIView`'s `contentScaleFactor` method.

17. You probably will want to implement setters for your drawing attributes in your custom graphing view so that, when someone changes these attributes, your view gets marked as needing to be redrawn.

18. Your graphing view's `contentMode` almost certainly will want to be `UIViewContentModeRedraw` since your `drawRect:` is (should be!) very high resolution.
19. Zooming in and out (pinching) is just a matter of changing the `scale` you are using to convert to/from view coordinates from/to your graph's coordinates. In other words, the pinching gesture handler should be very simple. Ditto panning and tapping with respect to your view's graph's `origin`.
20. It's fine if your graph's `origin` is maintained with respect to the upper left corner of the view when you rotate the device (as opposed to maintaining it relative to the center of your view, for example, which might be more what the user expects).
21. If all of this seems very similar to the Psychologist and Happiness demos we did this week in class, then you're on the right track!
22. As a guide, this entire assignment can be done in under 100 lines of code. Only a handful of lines of code need to be added to your existing `CalculatorViewController` and no changes should be required at all in your `CalculatorBrain`.
23. To test your application, try entering the `program` `sin(x)` or a simple line of the form `mx+b` or a quadratic equation. Try it without any `x` variable at all. Try it with a discontinuous function (e.g. `1/x` or `x * cos(1/x)`). Basically try anything that you think might break it.
24. The biggest "food for thought" on this assignment is the reusability and the scope of the keys you want to use in `NSUserDefaults` for your `scale` and `origin` preservation. Are you going to store these "by View" or "by Controller" or "by Application" or ??? For example, what if (in a future version of your application) you had **two** graph views on the screen at the same time? You wouldn't want their `scale` and `origin` to be tied to each other! Give this your best shot. There's no "right answer."
25. You'll almost certainly want your graphing view controller to be your split view's delegate on the iPad since it is the one who will have to put the button in a toolbar when your calculator view controller gets hidden by rotation. This is much simpler than what Psychologist had to do.

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.).
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Code is too lightly or too heavily commented.
- Code crashes.

Extra Credit

1. In the Hints section it is noted that you are allowed to draw your graph by drawing a line from each point to the next point. Clearly if your function were discontinuous (e.g. `1/x`) or if you had zoomed out so far that drawing a line between points would be jumping over a lot of changes in y, this would give misleading results to the user. The best thing would probably be to simply draw dots at each coordinate you calculate. This would not help much with the zoomed-out-too-far problem, but it would certainly be more accurate on discontinuous functions. It is up to you to figure out how to draw a dot at a point with Core Graphics.
2. If you do Extra Credit #1, you'll notice that some functions (like `sin(x)`) look so much nicer using the "line to" strategy (at least when zoomed in appropriately). Try dragging a `UISwitch` into your user-interface which lets the user switch back and forth between "dot mode" and "line to" mode drawing.
3. Improve the performance of panning. To do this, you need to try to understand where the CPU cycles are going when the graph is drawn. Is it our inefficient `runProgram:usingVariableValues:` method? Or is it all the Core Graphics calls we are making each time? Is there a simple way to reduce calls to both of these things in our `drawRect:`? Or is the performance issue something else entirely? If you are very brave, you can try to figure out how to use the Time Profiler (hold down Run in Xcode and pick Profile, then choose the Time Profiler from the dialog that appears). That's the way to really know where the time's going.