Assignment III: Graphing Calculator

Objective

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

By doing this, you will gain experience creating your own custom view, building another `UIViewController`, understanding more about the lifecycle of an application and the application delegate's role in that, using a protocol to delegate responsibility from one object to another, and creating a `UINavigationController`.

Be sure to check out the Hints section below!

Also, check out the latest additions to 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'll be creating a new project and copying code from last week's assignment.

Required Tasks

1. Create an application that, when launched, presents the user-interface of your calculator from Assignment 2 inside a `UINavigationController`.
2. The only variable button your calculator user-interface should present is `x`.
3. Add a button to your calculator's user-interface that, when pressed, pushes a `UIViewController` subclass onto the `UINavigationController`'s stack which brings up a `view` that contains a custom view which is a graph of whatever `expression` was in the calculator when the button was pressed. The y axis should be the evaluation of the `expression` at multiple points along the x axis (each of which is substituted for the x variable). Pick a reasonable `scale` for your graph to start with.
4. In addition to the custom view to draw the graph, the view that appears should also have two buttons, "Zoom In" and "Zoom Out" which increase or decrease the `scale` of the graph by a reasonable amount.
5. Your graphing view must display the axes of the graph in addition to the plot of the `expression`. 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 drawing the axes, only for the graphing of the `expression` itself. You probably will want to check this out to understand how the scaling works before you do #3 and #4 above.
6. Your graphing view must be generic and reusable. For example, there's no reason for your graphing view class to know anything about a `CalculatorBrain` or an `expression`. Use a protocol to get the graphing view's data because Views should not own their data.
7. Make your user-interface as clean as possible. For example (only an example), use some colors on the button titles to group them together visually. Set the `title`s of both `UIViewController`s that appear on the `UINavigationController`' s stack to something reasonable. Make sure your layout is balanced and aesthetically pleasing. Consider the user's experience when touching buttons (e.g. when the user touches the Graph button, you should probably automatically confirm any partially entered number as the operand and even do a `performOperation:@"="` on behalf of the user). The calculator UI this week now matters: it's not just a testing UI for the `CalculatorBrain` like it was last week.

Hints

1. When you create your new project in Xcode, make sure you choose Window-based Application (not View-based Application). You will have to manage the allocation and initialization of all the view controllers in this application yourself (in your application's delegate "did launch" method and elsewhere).
2. When you drag your `CalculatorBrain.[mh]`, `CalculatorViewController.[mh]` and `CalculatorViewController.xib` files from Assignment 2 into your new project, **make sure you click the box that says "Copy items into destination group's folder (if needed)."**
3. After you've created the project and dragged in your files from assignment 2, you might want to test that you've brought your reused code in properly by going into your application delegate's `applicationDidFinishLaunching:` and creating a `UINavigationController` and pushing a `CalculatorViewController` (created using `alloc` & `init`) onto it and then call `addSubview:` on your application's `window` with the `UINavigationController`'s `view`. Your calculator should show up. If it's all smashed or buttons are cut off, you'll need to go back into Interface Builder and re-lay out your UI (this is a good opportunity to get experience with the autosizing mechanism in the Inspector in Interface Builder). If your UI doesn't show up at all, this might be time to contact a TA.
4. Don't forget to click on the button "With XIB for user interface" when you create your new `UIViewController` for your "graph and zoom buttons" `view` (via New File ... in the File menu) or you won't have a `.xib` for it.
5. The "graph and zoom buttons" `view`'s `UIViewController` is just like any other MVC Controller: it's going to want to have a Model instance variable (what is the Model for this new Controller, do you think?) and outlets into its View.
6. There are 3 view controllers in this application. A `UINavigationController`, the view controller you wrote in assignment 2 (modified slightly), and a new view controller you're going to write for this assignment which controls the "graph and zoom buttons" `view`.
7. After you've created your custom `UIView` subclass (the one that is going to draw the actual graph itself) (also using New File ... in the File menu), it would probably be a good idea to go right into IB and lay out and wire up your "graph and zoom buttons" `view`. Don't forget to set the class of your custom `UIView` subclass in the Identity Inspector in IB after you drag a generic `UIView` out of the Library and position it.
8. This might also be a good time to add the button to your existing calculator view that pushes the "graph and zoom buttons" `view`'s `UIViewController` onto the `UINavigationController` stack. This pushing code lives in your view controller from assignment 2. That way you can easily test your new custom view as you are

developing it. You can dump the Solve button you had there for testing from last week.

9. It is okay to be calling your generic graphing view's delegate's data provision method repeatedly inside a drawing loop inside your custom graphing view's `drawRect:`. In fact, this is the preferred solution. Remember that the UIKit is only going to ask you to draw your custom view when some geometry has changed or someone calls `setNeedsDisplay` on your custom view.
10. When you go to implement your custom `drawRect:`, you'll want to use the helper code provided to draw the axes. 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:`.
11. The implementation of your custom `drawRect:` is deceptively simple. You just need to iterate over every pixel across the width of your view and convert (`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 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 the next point.
12. 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 since it's simple to implement and it's right a lot of the time. ;-) Check out the Extra Credit on this front below.
13. To take advantage of the very high resolution display 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. We talked about how to do this in lecture. Recall `UIView`'s `contentScaleFactor` method.
14. Zooming in and out (via the buttons) is just a matter of changing the `scale` you are using to convert to/from view coordinates from/to your graph's coordinates.
15. 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!
16. To test your application, try entering the `expression` `x sin =` or a simple line of the form `m * x + 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.
17. Pay attention to your memory management.

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.
- Code leaks memory!

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 adding a `UISwitch` to your user-interface which lets the user switch back and forth between "dot mode" and "line to" mode drawing.
3. Clean up the `descriptionOfExpression:` output. This is an exercise in anticipating all the possible `expression` combinations and also an exercise in using the `NSString` class. Think about not only converting the `expression` `x sin` to `sin(x)`, but even tricky memory operations, e.g., turn `3 * x * x = Mem+ 4 * x = Mem+ 5 Mem+` into `3*x*x + 4*x + 5`.
4. See if you can make one or both of your `UIViewController`s work when the device is rotated. You need to implement `shouldAutorotateToInterfaceOrientation:` to return `YES` more often than it does by default and you will need to get your springs and struts in Interface Builder set up properly so that views stretch in the right directions and stick to the edges of the view they are supposed to. This is pretty straightforward for your "graph and zoom buttons" view, but quite a bit more of a challenge (nigh on impossible!) for your calculator keypad view.