Assignment IV: Universal Calculator

Objective

The goal of this assignment is to make your Calculator work on both the iPad and iPhone (and iPod Touch) platforms. You will build a single application that works on both using conditional code inside your application. Your application will be enhanced to handle touch gestures and preserve its state across launches as well.

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. It is **strongly** recommended that you make a copy of last week's assignment before you start modifying it for this week's.

Required Tasks

1. Your Calculator application must work properly on both the iPad and iPhone, using appropriate user-interface idioms on each. Your submitted application will be built using the iOS 4 SDK and then tested against both the 3.2 iPad Simulator and the iOS 4 iPhone Simulator.
2. Specifically, on the iPad, instead of having a calculator in a navigation controller which pushes the graph on-screen when the Graph button is pressed, a split view should be used instead. In landscape mode your UI should appear with your graph on the right and the calculator on the left, and in portrait mode, the graph should fill the screen and there should be a bar button which brings up a popover of the calculator. The calculator's Graph button should still update the graph, no matter where the graph is in the UI.
3. Remove the zoom in and zoom out buttons and, instead, modify your application to recognize the following 3 gestures in your custom graph view:
   a. It must translate it's origin when the user pans around with a single touch.
   b. It must zoom in and out when the user pinches in and out.
   c. When the user double-taps on your graph view, it should move the graph's origin back to the center of the view.
4. Do not use a `.xib` file for the user-interface of your graph view controller anymore.
5. Whenever your calculator appears in a popover, the popover should be sized appropriately to fit the calculator.
6. When your application exits and restarts, its graph view should be showing the same scale and origin. Extra credit for preserving even more UI state than that across launches.

Hints

1. Remember that you convert your application to be a Universal Application by selecting your Target in Xcode and choosing "Upgrade Current Target for iPad ..."
2. Change the target deployment environment of your application to be iOS 3.2 or later by right clicking on your target (i.e. your application under the Targets folder in the Groups & Files area of Xcode) and, in the Build tab, scroll down to the property "iOS Deployment Target." Change this to iOS 3.2. This will mean that your application (though it is being built under iOS 4's SDK) will be deployable on devices with iOS 3.2 (or later). The iPad currently only runs iOS 3.2. You might also need to change the Base SDK build property to iOS 4.1 (this property may be set to iOS 4.0 and thus say 'missing').
3. You test your application on the iPad Simulator 3.2 by temporarily changing the pull-down in the upper left of your main Xcode window to iPad Simulator 3.2 instead of iPhone Simulator 4.0. If iPad Simulator 3.2 does not appear in that pull-down, you probably did not do hint #2 above properly. You can try holding down the Option key while clicking on the pull-down (which shows older SDKs), but truly this should not be necessary. If you ever accidentally rebuild your application with an old SDK, you'll probably want to choose Clean All Targets from the Build menu and rebuild (this is not a bad idea to do every once in a while anyway).
4. Use `UIGestureRecognizer` concrete subclasses to implement the panning, pinching and double-tapping, not `touchesBegan:withEvent:` and its brethren.
5. Think about where the code for your `UIGestureRecognizer` target/action methods wants to live. If you put it in your View Controller, then your custom view would only be able to handle those gestures if it's associated with your View Controller. Your custom view wants to be more independent than that! But now think about where the `addGestureRecognizer:` code wants to go. What if someone wanted to use your graph view but didn't want to allow panning or zooming (because maybe they are controlling the scale or origin themselves)? Adding a `UIGestureRecognizer` or not would be a good way to control that.
6. You can test pinching in the Simulator by holding down the Option key while mousing down. It only provides limited (symmetrical about the middle) pinching ability, but enough to test your code.
7. You already probably have a property in your graph view which is the `scale`, so implementing pinching is going to be pretty straightforward.
8. However, you probably do not have a property in your graph view which sets the origin of the graph, so you'll have to add such a thing and update your `drawRect:` to use it (once you've done so, implementing panning and double-tap will be a snap). Luckily, the axes drawing helper class we gave you **does** let you specify its origin, so at least that part will be easy.

9. The point of Required Task #4 is for you to gain experience with a `UIViewController` that uses `loadView` instead of a `.xib` file to create its `view` property. Since you have removed the Zoom In/Out buttons (via Required Task #3), all that is left in your graph view controller's view (i.e. its `self.view` property) is your custom graphing view, so it should be extremely simple to implement `loadView`.
10. You can simulate device rotation from the Hardware menu in the simulator(s).
11. It is **very important** that **both** of the `UIViewController` objects in a `UISplitViewController` support rotation to all interface orientations or it will not work when the device is rotated. If even one of the two does not, the `UISplitViewController` will not support rotation either and thus when you rotate the iPad (in the Simulator by choosing "Rotate Left" from the Hardware menu) it will be stuck in portrait mode. You need to implement the appropriate method in both of your View Controllers (hint: the method starts with the word "should") to allow rotation to other interface orientations. (`UINavigationController`s are automatically rotatable as long as the controllers inside them are also rotatable.) Be careful about your iPhone UI though: it might be rotatable now too! You'll either have to prevent that case in your code ("if I'm on an iPhone" is not the best test for this, by the way) or do Extra Credit #2.
12. `UIView` objects can do different things to get themselves redrawn when their `self.bounds` property changes (for example, when they are rotated and their bounds go from a tall, skinny rectangle to a wide, short rectangle). This redrawing behavior is controlled by the `UIView` property `contentMode`. The default is to stretch the bits to fit the new `self.bounds` (i.e. `drawRect:` is **not** called by default when the view's bounds changes). Since your graph view is smarter than that, you probably want to set this `contentMode` to something else. Check the documentation to figure out which `contentMode` you want. Although this property can be set in Interface Builder, that's no good to you for this assignment because of Required Task #4. You could set it in your view controller when you create the graph view, but you don't really want to do that because the way this `UIView` redraws itself when its `self.bounds` changes has nothing to do with your view controller, it's a aspect of the way the view chooses to draw itself. Thus, as a good object-oriented programmer, you will probably want to override `UIView`'s designated initializer `initWithFrame:` in your custom view subclass to set this property--check the slides about creating objects from lecture to remind yourself about the goofy way we override initializers. Even though in this assignment this view does not come out of a `.xib` file, you might want to make sure that this property would get set properly if that were the case in the future (hint: `awakeFromNib`).
13. You are going to be calling at least one method (actually, it's a property getter) that is only supported in iOS 4's SDK. You must use introspection to make this call conditional. Once again, it's not really the right thing to make it conditional on iPad vs. iPhone. Make it conditional on what it really depends on.

14. The best place to implement the `UISplitViewController`'s delegate methods is in the right hand view controller. Remember to keep it generic. It should work as the right hand view controller for any left hand view controller (i.e. not just for a left hand view which is a `CalculatorViewController`).
15. One change your `CalculatorViewController` probably needs is to the code where it pushes a graph view controller onto the navigation stack in response to the Graph button being pressed. This push is now contingent upon whether the view of the view controller it is about to push is already on screen or not. Clearly if it is on screen, no push is necessary.
16. You will need to link up your two view controllers. Right now, your calculator view controller probably lazily creates a graph view controller when it needs to push it, but now you probably want that graph view controller to be a settable property in your calculator view controller now so that when your base user-interface is being constructed in `application:didFinishLaunchingWithOptions:` (especially on the iPad), the two view controllers can be linked.
17. Your `CalculatorViewController` is also going to need to know (and report when asked by the appropriate method) what size it should be when displayed in a popover. The best way to do this is to find the bounding box of all the views in your `CalculatorViewController`'s `self.view` and then add a little margin around the edges. To calculate the bounding box of the views, you're going to need to look at the `subviews` of the `CalculatorViewController`'s `self.view` and all of their `frame`s. You might find the function `CGRectUnion()` convenient for this task.
18. The obvious place to store and recall your graph view's `scale` and `origin` is by sending appropriate messages to `[NSUserDefaults standardUserDefaults]`. Don't forget to send `synchronize` to that object when you change it.
19. Make sure that the correct object is responsible for saving and restoring the graph's `scale` and `origin`. Is it the custom graphing view? the graphing view's controller? the calculator view controller? the application delegate? the split view's delegate? some other object? Give it your best shot and add a comment (in your code and/or in your submission `README`) if you want to try to justify your choice! There's not necessarily a right answer or a wrong answer here, but we're interested in your thought process because understanding the boundaries of responsibility between objects in an object-oriented programming system is very important. Spend some brain cells on this.

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!
- Application does not work properly on one or the other user-interface idiom.

Extra Credit

If you do any Extra Credit items, don't forget to note what you did in your submission `README`.

1. Get your application to run on a physical device. If you have not registered your UDID with your TA, now would be a good time to do so! Don't forget the Weak binding for UIKit (since your device is running 3.1). We can't really verify that you did this, so **you won't actually get any extra credit for doing this one**, but it'd be cool to do anyway.
2. Make your user-interface work in landscape mode on an iPhone. This will require either an alternate `.xib` file or some fancy coding of the frames of your views. This is not an easy extra credit item.
3. Apple's user-interface guidelines generally suggest that if a user quits an application and comes back, everything should be as it was. Required Task #6 makes this partially true (but only for the graph's axes). This extra credit item is to make it completely true (or as completely true as you can). You will probably want to use the property list conversion methods you wrote in assignment 2 in order to write/read the `expression` out to `NSUserDefaults` (since `NSUserDefaults` only knows how to read and write property lists). Don't forget about the calculator's display and graph contents.