CSCI S-65 Summer 2015

Assignment 3

(FIXED) Due: July 15th just before Midnight

Total: 150 pts. Undergrad / 165 grad

READINGS

MVC:
https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

This is a Cocoa (OS X) centric discussion, so whenever you see NSController or NS*Xxx*Controller, imagine your subclass of UIViewController.
https://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14

UITouch:
https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITouch_Class/

Double-tap:
http://www.appcoda.com/uiscrollview-introduction/ (just the part about double-taps)

1. **Maturing Game of Life demo: Switch to using notifications for the model (25 pts)**

   Purpose:
   - Practice MVC as demonstrated in class and in the "UIBible", particularly steps 5, 6, 7, and 8 from the chalkboard diagram (Lecture 6-MVC-Block-Diagram.jpeg)

   a. (20 pts) *Estimated time: 1 hr*. Instead of directly calling `cellGridView.setNeedsDisplay()` in the `LiveCellGridViewController` class, use the notification techniques between the Model and the Controller developed in class for the `ColoredSquare` project. Make sure your shared String constants are updated to be meaningful.

   b. (5 pts) *Estimate time: 15 min.* There are two kinds notifications running around now: direct timer events, and model updates. Think of them as two separate radio stations that are broadcasting on two different frequencies. Is this cool, or confusing? How much so? In all seriousness, why not just send the "model changed"

For the game of life, you should be building on your assignment 2, since it assumes pausing functionality.
For the MVC class demo, you should 'git pull' and use Lecture 6-MVCClassDemo/MVCClassDemo.xcodeproj

`NSNotification` directly to the custom view class **`ModelBasedCellGridView`**, since that's where it really matters in this App?

2. **Cell Painting in the Game of Life (45 pts)** *Estimated time: 3-5 hours*

   Purpose:
   - Practice with the low-level touches*Xxx* UIViewController interface
   - Learn arbitrary path-dragging for drawing-oriented applications
   - Practice with model coordination: Converting from display (UIView) coordinates to logical (model) coordinates.
   - Making a more interactive, "touchier" interface

   While the simulation is not running, allow the user to "paint" cells. The TouchDemo project should be used as a template for capturing raw touch events. When the user first touches down, track whether the cell is "alive/born" or "died/empty". Change it to the opposite ("born" or "died"), and keep this "pen" active for the duration of the entire drag. **In other words, if the user touched down on a "live/born" cell, mark every live cell as "died" as the user continues to drag, and don't change any cells that are already "died/empty".** The end of the drag is indicated by a **`touchesEnded`** event. (You'll need a stored property at class scope to track this "pen" state, since your call to **`touchesBegan`** must return immediately.)

   This must be a one-finger drag only, so be sure to check that parameter. A two-finger drag should do nothing.

   It is fine if fast dragging skips over cells, in fact expected. This could be considered a feature. Interpolating between cell jumps is far too complicated to worry about.

   A cell must be logically AND visually updated the moment a touch passes over it.

   While the simulation is running, ignore touches.

   As a logical cell receives multiple touches during a drag (or if a cell already matches the pen's state) there is the danger of triggering needless calls to 'drawRect' – during a drag, you need at most one call per cell. Find a way to avoid needless calls to drawRect(). Hint: make the model smart!

   This should happen automatically if you use the model properly, however, to be explicit: When the simulation resumes, the changes made by any drags must be taken into account.

3. **ColoredSquareModel enhancements (40 pts Undergrad; 55 Grad)**

Purpose:

- Apply many of the other principles specified in the UIBible

a. (20 pts) *Estimated time: 1hr.* Add a new parameter to the model so that the shape is now a rectangle instead of a square: "height". Rename "sideLength" to "width". Use the parameter dictionary of the model just like the other parameters, along with setting the minimum, maximum, and a default value. Update the UI to allow this value to be edited, like the others. You must merge the action handling into the existing **`numericValueChanged`** method, and hence use the **`allFields`** dictionary.

b. (5 pts) *Estimated time: 15 min.* Is the **`allFields`** dictionary fully initialized before or after **`viewDidLoad`**? If I wrote a constructor for the ViewController (init method), could I correctly initialize **`allFields`** there? Explain.

c. (5 pts) *Estimated time: 20 min.* Make the maximum allowed width and height of the CG-drawn rectangle to be exactly half that of the containing view (the model's constructor will need to be updated).

d. (10 pts) *Estimated time: 30-45 min.* Add a new computed property to the model: the area of the square. What part of the UIBible says this area calculation belongs in the model? Use this computed property to keep a constantly up-to-date readout of the rectangle's area shown on screen.

e. (15 pts) *Estimated time: 1 hr.* **GRADUATE CREDIT** Use the doubleTap code shown in the middle of this tutorial: http://www.appcoda.com/uiscrollview-introduction/ If a double-tap occurs in the colored rectangle, both dimensions should immediately double. You will find the global function **`CGRectContaintsPoint`** useful. A double tap elsewhere in the custom view should do nothing. Be sure to defer to the model properly – don't let the controller do the model's job.

4. **PROJECT IDEA** (40 pts): *Estimated time: 3-5 hours, and some dreaming* We need to know now what you plan to do for your final project. To be "rich enough" it must use at least 4 of the following:

    - Storing data in the device filesystem (often via CoreData, but also directly via file URLs is possible)
    - Reading data from the network or Web, directly via URLs or via one of the Social Media libraries
    - Storing data back onto a web server or database
    - Capturing images, video, or audio from the device and allowing some kind of manipulation (saving, display, playback)
    - Using the CGImage toolkit to allow user edits

- Having multiple screens that you navigate through (via a NavigationController, a Master/Detail view, or a TabbedViewController)
- A user preferences screen that controls display preferences such as colors and font sizes
- Use of the Physics toolkit
- Use of the Accelerometer
- Use of the GPS
- Use of MapKit
- Use of the Social Media toolkit
- Use of the AirDrop toolkit

If you don't know what you want to do, research these facilities and see which one interests you. The App must have a single coherent purpose; it can't be a grab-bag of random demos from the above.

This problem will get full credit with any good faith effort of 100 words or more, as long as it explains *how* it will use 4 of the above facilities.

(15 pts) *Estimated time: 1-3 hr.* **EXTRA CREDIT** (If you're just zooming along, or want to make up some credit from a previous assignment) Use the above scroll view tutorial to make a custom Game of Life view in a much larger view that doesn't fit on screen, but is embedded in a scroll view. **UIScrollView likes to take over 'touches' however, in order to support all the scrolling gestures: panning and zooming. This is a "hostile" environment for the custom touches which are needed for the painting. So, for this, submit it separately (separate Scene or separate Project, and document what you did) and don't expect painting to work.**

Workarounds are quite complicated and won't be covered here. For example, you could re-conceive of scrolling entirely as a logical cell scrolling, and assign a two finger drag to it instead, but then you'd have to recreate, or fail to recreate, all of the physical effects (momentum, bouncing, physics) built into UIScrollView. There are other tricks involving two views layered on top of one another and switching which is the focus of the touch detection.