# CSCI E65g: Mobile Application Development Using Swift and iOS

## Fall 2018

## Assignment 6

### Issued: 10/09/2018 Due: 10/16/2018

### Scoring

Undergraduate: 140 points; Graduate: 200 points

### Repository

Find through [Github Invitation](https://classroom.github.com/a/eD-7Yd5B).

### Overview

You will creating a subclass of UIView called GameGridView. Unlike the very complex multi-object view UICollectionView which is also a grid-oriented view, the job of GameGridView is to represent the state of a simple two-player grid-based game. The rules of the game are intentionally abstract for this assignment and will only have a trivial stub implementation. Assume a two player game where each player has a unique color. Players alternate placing their colored marker on any unused square. Each player chooses a square simply by tapping on it. When the game ends, the App is able to detect it by querying the game logic, and updates the user interface with a message and by preventing further play.

Undergraduate: Ignore the game state logic and related UI outside the grid view itself; only implement a grid view that accommodates tapping to toggle the state of a cell between on and off.

The game grid view we need for this does not exist in iOS so we must write it. To have a well-defined approach that our peers and the iOS developer community can quickly lock onto, we will closely follow the design pattern of other complex views like tables and pickers. It will have its own:

* Rendering logic ([draw method](https://developer.apple.com/documentation/uikit/uiview/1622529-draw)): geometric shapes determined by its size and information given by its data source.
* Data source protocol: define a series of questions that it will use to query the current state of its data
* Behavioral delegate methods: define a series of callbacks that indicate when a meaningful touch action has occurred.
* Appearance delegate methods: define some callbacks that customize its superficial appearance.
* Mutation notification method: define a method that allows the controller to tell the view that (some or all of) the data source values have changed.

The intent is to learn all of these concepts as simply as possible, while understanding this is the very design used by Cocoa Touch and by industrial-strength commercial Applications.

### New APIs

There isn’t much, but the power gained from these is immense.

* [UIColor](https://developer.apple.com/documentation/uikit/uicolor): Represents a displayable color for drawing geometrical objects. In particular, [using pre-defined system colors](https://developer.apple.com/documentation/uikit/uicolor#2603616) and setting the stroke and fill color with [setStroke:](https://developer.apple.com/documentation/uikit/uicolor/1621948-setstroke) and setFill:, respectively..
* [UIBezierPath](https://developer.apple.com/documentation/uikit/uibezierpath): Represents an arbitrary polygon or curve that can be stroked, filled, or both with a given color. In particular the [shortcut constructor](https://developer.apple.com/documentation/uikit/uibezierpath/1624359-init) for rectangles.
* [UIView](https://developer.apple.com/documentation/uikit/uiview) properties and methods: isUserInteractionEnabled, bounds, setNeedsDisplay:, and overriding: didMoveToSuperview:, draw:rect:
* [UITapGestureRecognizer](https://developer.apple.com/documentation/uikit/uitapgesturerecognizer): detecting finger-tap events

### Reading

* [Ray Wenderlich tutorial on making a custom UIView](https://www.raywenderlich.com/411-core-graphics-tutorial-part-1-getting-started)
* [Quick run-down of using UIBezierPath](https://www.hackingwithswift.com/example-code/uikit/how-to-draw-shapes-using-uibezierpath) to design simple shapes.
* [CodePath Tutorial on Gesture Recognizers](https://guides.codepath.com/ios/Using-Gesture-Recognizers) Focus on the Tap variant, and doing it **programmatically** just to see how things work outside the world of Storyboard, in particular learning [#selector](https://developer.apple.com/documentation/swift/using_objective-c_runtime_features_in_swift) syntax (see example listing and just follow pattern). If you have a fancy project idea involving more than just taps, see this elaborate [Ray Wenderlich tutorial](https://www.raywenderlich.com/433-uigesturerecognizer-tutorial-getting-started).
* [My own draft on handling a 2D grid model expressively](https://harvard-ios.github.io/csci-e65g-2018/supplemental/ThinkingIn2D.html) using custom types and operators.

#### Logical design

The view will consist of a two-dimensional array of locations. The values are configurable from 3x3 up through 20x20. The horizontal and vertical dimensions need not be the same so 3x20 or 20x3 is permissible.

Each location is extremely simple and tracks only 3 states: Empty, Player1, or Player2. *Implementation note: It must be said early: each location will be rendered as a simple 2D rendering action and****not****a separate UIView subview!*

#### Object design

GameGridView will have dataSource and delegate properties which must be protocol-typed (your own new protocols). As in Assignment 3, the view controller will serve as both so must adopt these newly defined protocols. The specification will only be more in plain English to give you more freedom (and responsibility) in naming and designing the function signatures.

The datasource protocol must:

 Provide a method to answer how many rows are in the grid

 Provide a method to answer how many columns are in the grid

 Provide a method to answer in what state is a particular grid location (see Logical design above, only 3 states)

The delegate protocol must:

 Provide a method to answer, given a grid state, in what color should the grid be drawn

 Provide a callback method that informs the delegate when a user has tapped on the grid, and at what location using **logical** coordinates, that is, in terms of a zero-based row and column number. (The other coordinate system is the **visual** or **screen** coordinates, which are measured in points.)

The protocols must follow the pattern of delegate and dataSource method signatures in UITableViewDatasource and UITableViewDelegate, so they must make heavy use of function overloading, and always pass in a reference to the view class itself as the first argument. (This is part of the overriding philosophy of this problem set, to emulate Apple’s own design patterns, even if occasionally tedious.)

The view implementation itself must:

1. Provide stored delegate and dataSource properties. In keeping with the UITableView pattern, these must be declared as weak, be writable, be protocol-typed, and be Optional. It is not an error for a view controller managing this view to leave these nil.
2. Crash if the aforementioned limits on grid size are violated by the data source.
3. Not crash, throw, or do anything else disruptive if either or both the delegate and datasource are left empty. The default grid size, lacking a datasource, will be 2x2. In this case tap events should still work. There should be muted default colors if the delegate is missing.
4. Override the standard custom-rendering method [draw:rect](https://developer.apple.com/documentation/uikit/uiview/1622529-draw). Using the delegate and dataSource properties, this method should render the grid as a series of equally sized rectangles in the appropriate color. The sample code will show how to draw an arbitrary rectangle in an arbitrary color. The logical grid coordinates follow Core graphics rendering conventions, meaning y values (row numbers) increase as you move down the screen.
5. Implement a method reloadAllSquares (that takes no parameters) analagous to [UITableView.reloadData:](https://developer.apple.com/documentation/uikit/uitableview/1614862-reloaddata) that causes the view to re-render itself assuming all of its current appearance is out-of-date.
6. Implement a method analagous to [reloadRows](https://developer.apple.com/documentation/uikit/uitableview/1614935-reloadrows) that requests the view to re-render only ONE square (don’t have to be quite as fancy and accept an array of individual squares). Because of how rendering works, the implementation will ignore its input and is otherwise identical to reloadAllSquares. The iOS rendering engine actually clears the entire UIView viewable rectangle anyway once any part of it is marked invalid. True optimizing involves manipulating an off-screen graphics context to save as an image, then copying the entire image to the screen when refreshing. This is out of scope for this assignment.
7. Implement a [tap gesture recognizer](https://developer.apple.com/documentation/uikit/uitapgesturerecognizer) that translates the relatively low-level touch coordinates (which are measured in screen "points") to logical coordinates (measured in values returned by the datasource). These logical coordinates be forwarded to the delegate callback method detailed above. There is no implication about what the touch will actually do.

### The Data Model

To allow for the full flow of a real game, we’ll need more information. The model protocol must be totally distinct from the data source protocol (why?) even though some methods are conceptually very similar. It must give the Controller enough power to fully manage a game UI. The model protocol must:

 Provide methods or properties to reveal the extent (dimensions) of the game grid.

 Provide methods (or subscript operator functions) to reveal the state of a location given its coordinates.

 Provide a method to accept a player’s turn, which consists of the player ID and coordinates of the chosen move. This method must throw an exception if the move is illegal (including occupied square or out of bounds) or if that player was not allowed to move at that point (other player’s turn or game over).

 Provide a method or property to answer whose turn it is.

 Provide a method or property to reveal the game state: ongoing or completed, and if completed, the result.

 Notify the dataListener, if any, when any significant mutation occurs.

Although it’s not necessary for the implementation to reflect this, there are conceptually two layers to the model.

* The *board state*: the state of every square in the board: empty, player1, or player2. This layer informs the appearance of the grid itself
* The *game state*: the progress of the game: How many turns have been taken, whose turn it is, and what stage it is at. This layer informs the rest of the UI outside the grid.

All information about the game state must be encapsulated here. Nothing abstract about the game itself may be stored anywhere else. In the future, this would include the player’s names, scores, game history, etc.

Since we’re not interested in writing game logic yet, implement a trivial game logic that always starts with "Player 1", forces players to alternate, is automatically over after 5 turns each, and should be reported as a draw. When a player taps a square, that grid location should change state to that player. A player may not move in a location that has already been used by either player, so you will have an opportunity to throw exceptions.

### View implementation notes

The view must be implemented as a sub-class (inheriting from) UIView. Naturally it goes in its own source file. Unlike data models, it relies on UIKit so don’t forget to import that. Through the magic of inheritance, all the fields and methods of UIView (see references) are available, even though you don’t see them in your file.

Designate the custom view as @IBDesignable. Now you’ll see the power of having defaults when the delegates are nil! Xcode will actually execute the default constructor (which includes static initialization and any instance initialization within property declarations) and show the result right in Storyboard.

A lot of tutorials on custom UIViews deal with externalities to our focus. If you explore, you’ll need to filter the material carefully. The following are interesting but **are not**part of this assignment:

* Designing a custom UI view in Xcode visually (creating a XIB file). We are doing it entirely *programatically*, that is, in code only.
* Adding subviews to your custom view.
* Using the lower-level drawing functions like UIGraphicsGetCurrentContext, CGContextSetFillColorWithColor and CGContextAddLineToPoint.
* Overriding the default constructors (required when a XIB file might be involved)
* Overriding layout functions (required when adding subviews)
* Causing setNeedsDisplay to be called automatically when certain properties are mutated by overriding UIView.needsDisplay(forKey:)
* Breaking down the rendering into CALayer objects (core animation layers) used for visual effects, high-performance rendering, and animation. (But see [a tutorial](https://www.raywenderlich.com/402-calayer-tutorial-for-ios-getting-started) if curious).

All drawing is dimensioned in CGFloat. This is a tissue-thin wrapper around Double but gives some generality in case a different type is ever used for screen coordinates. When performing floating point arithmetic, especially when converting from Int values, use CGFloat not Double. Use CGPoint to represent a point on the screen and CGRect to represent a rectangular area.

You will need a UITapGestureRecognizer. Do it in code only not Storyboard. Delay its initialization until it’s needed by using the didMoveToSuperview: method as the trigger.

You will need to use the UIColor class to specify the drawing colors.

You will need to use the UIBezierPath class to render rectangles easily without using UIViews. The stroke method draws the outline of a shape, which is a very easy (albeit a little inefficient) way to get gridlines.

Since you cannot call the draw:rect: method directly, you must trigger a call by using setNeedsDisplay: when implementing the reload methods.

The size of each rectangle depends on the currently laid out dimensions of the view as well as the number of rows and columns. Properly calculate this every time. Use the bounds property for the current size measured in points.

### Model implementation notes

Remember to put each data model class its own .swift file. Because these files are unrelated to the appearance, layout, or physical dimensions of the view, no UIViews may be mentioned.

You’ll probably want to use Swift’s built-in [2-dimensional array](https://www.dotnetperls.com/2d-swift) capability, which is really just an Array of Arrays. The [repeating-element initializer](https://developer.apple.com/documentation/swift/array/1641692-init) saves a lot of manual looping code. Within those docs, the confusing notation Array<Element>.Element refers to the type of each element.

You may be tempted to use a Swift built-in type for the inner element type of the Array; why is this frowned on and what’s the superior alternative?

When broadcasting a change notification, some level of detail is helpful so that the Controller can minimize the updating work it requests of the View. To be more specific, the simplest zero-parameter somethingChanged: callback pattern is a little too simple.

Don’t forget the game logic goes here too so the model needs to know whose turn it is, and how to detect the end of the game (see above; it’s deliberately trivial.) It also needs to ensure no llegal moves and enforce alternation.

### View Controller implementation notes

You’ll have to deal with errors thrown by the model. Propagate them to the UI using a [UIAlertController](https://developer.apple.com/documentation/uikit/uialertcontroller). The simple example shown in the reference is adequate but make it specific to the situation and in plain English. Many tutorials abound; popping up an alert with a **Dismiss** button is very simple.

Follow the new MVC pattern strictly, according to the sample lecture code. The cardinal rules (following each one has a certain point value):

1. All action handlers, data source methods, and delegate methods must go in the controller.
2. All model data goes into well named, separate classes in separate .swift files, never in the controller or view. The controller is not even allowed to temporarily cache model data in a private stored property.
3. All model data access and mutation must be defined by a protocol.
4. All model references must be typed by the above protocol, never a concrete type.
5. In the implementation of an action handler (and the conceptually identical delegate “something-happened” notification methods), the only job is to translate this into a model mutation. No more and no less!
6. In the implementation of a data source method, the data must come from the model and get translated into something that makes sense for that particular view. There can never be model mutation in these methods.
7. When anything is mutated in the model, it must notify interested parties via a callback mechanism. For now, it will be via a dataListener property, which must be protocol-typed. This protocol will contain a simple set of methods, and maybe just one. Consider the most important event: a player has moved. (Strictly speaking, the controller does not need to hear a separate “game ended” event; it can simply check after every move.)
8. The controller must implement the data listening protocol and ensure the correct views update as a result. When a player moves, *several* things in the UI need to update. This is the burden of the controller! But it (you) can take comfort that all state is just a phone call away in the model.
9. For all the single-value views, the view update logic must be consolidated into a single method updateUI in the Controller, which consults the model for the necessary values. Naturally, the various methods that implement the data listening protocol need to call this method.
10. The view must rely entirely on the datasource and delegate methods to configure its particular appearance for that App. The only internal assumptions it can make are its fundamental nature: in this case, equally sized rectangles packed in a grid, each of which is filled with one color, and that tap events should be detected and translated into the logical coordinates of the tapped rectangle.
11. The view must take low-level events (in our case, gesture recognition events) and translate them into delegate callbacks. It must not be aware of the model at all, or of the concrete type of its datasource or delegate.
12. The view should have reasonable default behavior if either or both the datasource or delegate properties are empty.
13. The view must store the datasource and delegate properties via a weak reference. Just use the keyword weak for now; explanations will come later.

### Overall implementation notes

Using a [typealias](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html" \l "ID327) for a labeled tuple of row and column is very useful and is perfectly fine to declare globally to be used by all classes. If you prefer a struct that’s fine too. Again for dealing with 2 dimensions with clean syntax and value safety, see [my tutorial and 2D representation discussion](https://harvard-ios.github.io/csci-e65g-2018/supplemental/ThinkingIn2D.html) – this is *guidance* not a requirement!

### Requirements of the Main interface

Regarding layout and keeping track of views, this should be as simple as possible. There will be nothing more than 3 labels and the custom grid view.

1. Lay out the grid, giving it as much space as possible (edge-to-edge). Assume 7 columns and 6 rows.
2. There should be two labels outside of the grid labeled Player 1 and Player 2. Each time a touch event within the grid is received, that player’s color should fill the tapped square.
3. The player labels should update their background color to indicate whose turn it is: white background when not a player’s turn, and change to the player’s square color when it is.
4. Disable user interaction with the grid view when the game is over but enable it when it is underway. See the isEnabled property of UIView.
5. Have a status label that indicates an ongoing game, won (by whom), or drawn.

### Grading breakdown

Each particular crash modality will be a deduction of 5 points in the relevant section.

1. (40) View rendering, relying on info from its data source and delegate, and filling the grid view with the correct colors. Not assuming a fixed rectangle size but querying the current layout
2. (25) View tap detecting, forwarding correct grid-based events
3. (20) Controller management in handling view-based events
4. (30) Controller management in handling model-based events, including showing game state and whose turn, and stopping game play once game over
5. (15) Protocol design following Apple delegate/dataSource patterns
6. (15) Model prevents invalid game states
7. (10) Model forwards events at the right times
8. (15) Correct logic of trivial game
9. (30) Code style: the usual naming and formatting, readability, Swifty idioms