Project 2 - Sudoku

CS 458

1 Introduction

For this project you will create an iOS app that allows the user to fill in Sudoku puzzles. The app will start up in a view allowing the user to select whether they wish to start an easy or hard puzzle. If the user had a game in progress, they should also have the option to continue their existing game. Once a selection has been made, you'll switch to a different view controller presenting the puzzle interface.

I've provided some code which uses Core Graphics to render the puzzle itself, so you don't have to deal with that. The details of the interface are up to you, so long as it conforms to the guidelines in this document. I've also left the details of your game model pretty free-form.

Section 2 covers how to go about integrating the resources I've provided into your project. Section 3 lays out the requirements for the interface, and what the provided SudokuView expects on that front. The model functionality expected by the SudokuView is covered in Section 4, along with some general pointers. More details on game place are presented in Section 5 Grading details are in Section 6, along with some extra credit opportunities, while what you need to submit is in Section 7.

2 Getting Started

I have provided a number of files you may use for this project. They are:

- `simple.plist` – a plist containing an array of strings of simple sudoku puzzles
- `hard.plist` – like simple.plist, but difficulty puzzles
- `square.pdf` – a vector image of a white square with a black border
- `darksquare.pdf` – a vector image of a grey square with a black border
- `SudokuView.swift` – A subclass of UIView that will handle aspects of the sudoku view itself

The PDFs should be added to your app's assets. Mark them as single scale, and preserve their vector data. The remaining files should be added to the main project.

3 The Main Interface

Your application must contain at least two views managed by a navigation controller. The first view must, at minimum, provide a button to start a new easy game, one to start a new hard game, and one to continue an existing game. The continue button should be available if the user is in the process of completing a puzzle or an old puzzle was saved. If the user can continue an existing puzzle, the new game buttons should ask for confirmation using a `UIAlertController` alert before they actually create a new puzzle.

Starting or continuing a game triggers a show segue to a second view, which presents the sudoku grid along with a selection of buttons for filling in the puzzle. See Figure 1 for an example. Minimally, you need to provide 12 buttons, corresponding to the values 1-9, along with buttons for pencil mode, clear cell, and a menu. There also needs to be button in the nav bar which will prompt for confirmation and then abandon

the current puzzle. If the user returns to the first view via the back button, let them continue the puzzle. If they abandoned the puzzle, they are returned to the first view but cannot return to the puzzle in progress.

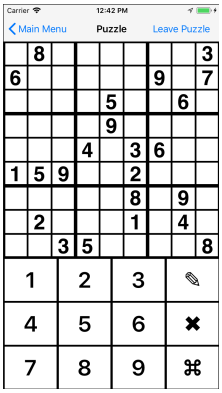

Figure 1: Example layout for puzzle view

3.1 Layout Constraints

The SudokuView class I provided needs a square area in which to work. To accomplish this, create a UIView (referred to here as PuzzleView), pin it to the top, leading, and trailing edges, and constrain it to a 1:1 aspect ratio. Set the view to be an instance of SudokuView, and it should be mostly good to go.

You are free to lay out the puzzle-completing buttons however you'd like. My implementation puts a UIView in the space under PuzzleView, and then lays out the buttons with nested stack layouts.

3.2 Letting the User Select a Cell

SudokuView is designed to handle tap gestures to allow the user to indicate where to fill in data. To facilitate this, in the main storyboard grab a "Tap Gesture Recognizer" and drop it onto PuzzleView. This should cause it to should up in the main view hierarchy and in the collection of icons above the view controller. Now, connect the recognizer's selector action to the PuzzleView's handleTap: method. (Either control-click the recognizer in the view hierarchy and drag from the selector to the PuzzleView, or control-drag the recognizer to the PuzzleView. In both cases, select handleTap:)

Once it's connected, PuzzleView should take care of managing which cell is selected on its own. Only cells which are not a fixed part of the puzzle can be selected. You can determine which cell is currently selected by checking the selected property of the PuzzleView, which is a tuple giving row and column. -1 in either field indicates that no tile is selected.

4 The Puzzle Model

4.1 Required Methods

SudokuView assumes your app delegate has a property called `sudoku`, which is the overall model for the game. Further, it expects the following methods to be present:

- `func numberAt(row : Int, column : Int) -> Int` – Number stored at given row and column, with 0 indicating an empty cell or cell with penciled in values
- `func numberIsFixedAt(row : Int, column : Int) -> Bool` – Number was provided as part of the puzzle, and so cannot be changed

- `func isConflictingEntryAt(row : Int, column: Int) -> Bool` – Number conflicts with any other number in the same row, column, or 3×3 square?
- `func anyPencilSetAt(row : Int, column : Int) -> Bool` – Are the any penciled in values at the given cell?
- `func isSetPencil(_ n : Int, row : Int, column : Int) -> Bool` – Is value `n` penciled in?

Obviously, you will need other methods. You may define them however you wish.

4.2 Loading Existing Puzzles

I have provided a pair of property lists which encode an array of strings. Each string is exactly 81 characters long, and contains the 9×9 puzzle in row-major order. Digits in the string correspond to numbers in the grid, while periods are used for empty spaces. The following method can be used to load the plists into arrays:

```
func getPuzzles(_ name : String) -> [String] {
    guard let url = Bundle.main.url(forResource: name, withExtension: "plist") else { return [] }
    guard let data = try? Data(contentsOf: url) else {return [] }
    guard let array = try? PropertyListDecoder().decode([String].self, from: data) else { return [] }
    return array
}

...

lazy var simplePuzzles = getPuzzles("simple")
lazy var hardPuzzles = getPuzzles("hard")
```

Once you have the arrays loaded, randomly choose a new puzzle as needed.

4.3 Data Persistence

Your app must save the current puzzle state – including penciled in values – when it goes to the background. When the app is reloaded, it must then give the option of continuing the saved game.

You are free to accomplish this however you like. One fairly straightforward approach is to make you sudoku model `Codable` and then rely on the `PropertyListEncoder` and `PropertyListDecoder` to format the model as a plist you can write out and deal with converting a saved plist to a model object. Note that this approach will involve replacing your existing model instance. Alternatively, you could have your model write its state into a `Codable` format and then be able to read such a format.

5 Playing Sudoku

The user enters a number into the puzzle cell by selecting the cell with a single tap (see Section 3.2) and then touching the appropriate digit button. If the user presses the same button again or hits the "clear cell" button (annotated in Figure 2) the number is cleared; entering other digits will have no effect until the cell is cleared or another empty cell is selected. The pencil button is a toggle switch as illustrated in Figure 12. Make sure the button looks different in its "selected state." (Note: IB will yet you configure how a button looks in various states. Look for "State Config" near the top of the attribute inspector.)

You'll need to keep track of the state of the pencil, and update it appropriately. Assuming you've got a boolean named `pencilEnabled` in your view controller, this will do the job:

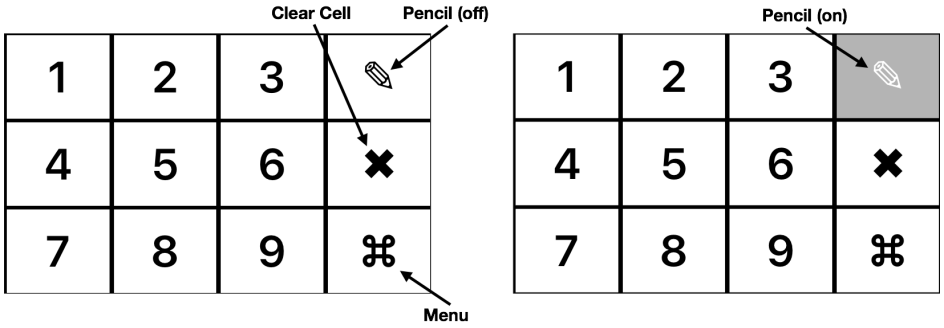

Figure 2: Annotated buttons. Note the pencil switching between off (left) and on (right) states.

```
var pencilEnabled : Bool = false  // controller property
...
@IBAction func pencilPressed(sender: UIButton) {
     pencilEnabled = !pencilEnabled   // toggle
     sender.selected = pencilEnabled
}
```

The user can "pencil in" multiple values in a cell when the pencil button is in its on state. The user can clear a pencil value by selecting the cell, setting the pencil state to "on," and pressing the digit of the pencil value to erase. All the pencil values can be cleared via the "clear" button. The "main" button brings up a `UIAlertController` action sheet that gives the user the option to clear all conflicting cells (that is, ones that cannot have the value they do, based on the state of the board), and to clear all cells.

6 Grading and Bonus Points

This assignment is worth a total of 50 points, broken up as follows:

10 pts	General Style / Compilation
15 pts	View transitions work as described
5 pts	App can start new games using plist or equivalent
10 pts	Puzzle interface works correctly
05 pts	Alerts and action sheets work correctly
05 pts	Game saves / loads state appropriately.

You can earn some bonus points for going above and beyond the requirements of the assignment, to a maximum of 5 bonus points. Here are a few examples of things you might do:

- Allow the app to work in landscape mode by adapting the constraints and moving buttons (2pts)
- Lay out buttons to keep them nearly square on any device (e.g. 3×4 for phones, 2×6 for iPads) (2pts)
- Detect that the user solved the puzzle and notify them (1pts)
- Complete rows / columns / 3×3 squares with exactly one remaining space (1 pts)

7 What to Submit

Zip up your project's directory, as usual. If you would like your project to be considered for extra credit, include a README file in the zip which indicates what else program does, and how to use the functionality.