Connect 4 Read Me

Eoin Ó'hAnnagáin - 20201662

Introduction

Welcome to the read me for the Connect 4 iOS application. In this document I will discuss some of the aspects of how various features were implemented into the app. Please note that throughout I will be referring to the MasterVC and SecondaryVC a lot due to the use of the splitViewController. The game area is in the latter of these and the tableViewController used for displaying historic games and options is in the former, however I have elected to call them these names for clarity. This document will mostly focus on the key features of the app as laid out in the brief, however a final section will discuss some of the extra, miscellaneous, features. The appendix also contains various screenshots of code and the app running in a simulator. These were mostly all taken from the iPad Pro (12.9 inch)(5th generation) simulator as this was the primary development and testing environment, due to the use of a splitViewController, although the iPhone 13 mini was also used, and works just as well.

Layout

The original plan for this app was to use a navigationController to divide the app between its game mode and historic mode. Users could use a tab on the bottom tab bar to go to the game's view and use a second tab to go to a splitViewController where the the MasterVC would contain the tableView of historic games and the SecondaryVC would play the selected game. After much experimenting this approach was abandoned as, while it can work inside a navigation controller, a splitViewController isn't designed to work as such and had unforeseen complications, usually actions not working as expected. This approach can be seen in appendix 1.1 and a second attempt in 1.2.

Taking my learning from this, I transitioned the app to being just the splitViewController. Now the MasterVC would manage both the new and current games in one section and the historic games in the other. Regardless of the row chosen you would segue to the SecondaryVC although your choice would effect how it acted as discussed in the next section. NavigationController elements were still embedded to control how some actions worked. This final layout's early form can be seen in appendix 1.3 and its final layout in 1.4.

Game View and Life Cycle

The SecondaryVC is responsible for managing the main game area. Inside it is the gameAreaView, a UIView which resizes itself based on the size of the screen, although it has a maximum size of 707:594. This UIView contains one subview on which the game board's bezier path is drawn, although many subviews are added for the pieces. This will be discussed further in the next section.

The SecondaryVC acts differently based on the mode it is in, chosen by the row in the MasterVC that the user used to segue to it. If the user chooses the New Game cell the new game mode will be entered and will begin by asking for the user's position, first or second, followed by the colour they wish to be. This will then launch into a new game session. In the process various variables are updated for data persistence. Launching the SecondaryVC in this manner will clear any current game stored.

Current will load the users current game through the string stored in user defaults. Here a boolean is used to indicate that the pieces should be placed all at once in their final positions, instead of dropping one after another, as the same method is used for spanning the pieces. The array of previous moves is then used to launch a new game session using these initial moves.

As a stylistic decision the board is completely black against a gradient background. This can create difficultly for users in identifying where their pieces can be places. To aid in this, as well as to indicate to users when they may begin placing pieces in the new game and current game modes, the gameAreaView's background will briefly flash the users colour when starting a new game session. This can also be triggered when playing by tapping outside of the game's board.

Swiping in this area will also clear the game board by removing all the pieces collision. Once enough time has passed for all pieces to be off screen they will also be removed from their superview and the pieces array. At the end of a game similar actions are taken although the gameAreaView will also flash the winners colour, as seen in appendix 2.1, among other actions. Both of these will also then trigger the position buttons to show, beginning a new game.

Once a game has concluded all the game pieces will be compared to the array of winning positions provided by the model. If their location matches a winner they are recoloured to white. All pieces will also have their subview containing the number label displayed. This can be be seen in appendix 2.2. After the pieces fall off screen a message will clarify the winner, shown in appendix 2.3.

Historic view works differently to the others. It begins much the same as current mode, although the data string is passed from the MasterVC to prevent a second retrieval from core data, in addition to the one made by the MasterVC. Unlike the current mode, pieces are dropped from the top of the screen using an interval of one second. The winning pieces are coloured white, and their number label is displayed in the players colour so their owner can be identified. This mode also disables all swipe and tap interaction from the user. After the game has concluded a repeat button fades in on the top right corner, allowing the game to be played again.

Paths, Animation, and Collision

A big challenge for this was to manage the drawing of the game board and barriers. The gameAreaView is given a maximum height and width of 707:594. These can be reduced if the screen size doesn't allow this size, but will always maintain this aspect ratio. This is achieved by giving the height and width constraints lower priority than the aspect ration constraint, as well as making then ≤ their max size. To design this I initially worked using the iPad only as it will always used these sizes, allowing me to hard code the values used. 707 was chosen as the width as it allows the columns to each have a width of 100 with a barrier of 1 dividing them. Pieces are set to have a diameter of 99, allowing them to fit perfectly in the columns. The hight is then six pieces high at 594 (99 * 6).

This resulted in a game board seen in appendix 3.1 when all of the barriers are drawn out as beaker paths. Unfortunately, collision refused to be added to these paths. Instead the UICollisionBehavior( )'s addBoundary( ) method was used, drawing invisible base and column boundaries over the lines which were removed in the future. These proved a challenge themselves since they could not be seen and, when not placed where expected could cause unusual behaviour, including teleporting pieces to the top left of the screen, both when they hit the bottom of a column, or just the top. Eventually, this bug was addressed and the result was collisions and the board layout working as expected, as seen in appendix 3.1.

Pieces are created using a circular frame, each with a subview containing their number, hidden until a game ends unless in historic mode. They are given collision and dropped from the top of the screen. To drop them the top of the screen the drop location calculated using the frame of the gameAreaView. By using -(gameAreaView.frame.minY) the distance to the top of the device is calculated in relation to the view the pieces will be added to as a subview.

For calculating the location on the x axis to place a piece an array is populated during viewWillAppear( ), used as the gameAreaView will have its final size and coordinates at this point. This array will contain the location of all points that the barriers are at +1, the point at which a piece can drop perfectly into a column. As this feels inorganic, a variance is also calculated based on the diameter of the piece. By dropping the piece in a random position within ± of this drop variance it can cause the piece to bounce briefly on the edges. This was original hardcoded to ±49, slightly less than the diameter of the piece. By putting in a one second delay before the bot's piece is dropped, or keeping the game area's tap disabled for one seconds after the bot's piece so the user doesn't drop too early, even if the next piece is dropped in the same location they should just miss one another. In all my testing they have come close to one another but have never bounced off one another into an incorrect column. Only once has a piece ended up in the wrong column, on a smaller iPhone in landscape. This was fixed by reducing the variance by one point on either side. The result of this can be seen in appendix 3.2.

With all the values hardcoded it was simple enough to figure out how to calculate them instead. This would allow everything to work on a device that would not allow the 707:594 frame.

None of these variables caused any particular issue to softcode though, bar the invisible barriers as, due to being invisible, if there was an issue it was not immediately apparent.

For the bezier path to draw the game board a view is placed on top of the gameAreaView with matching dimensions. Within this, the bezier path is drawn with subpaths used to remove circles at each of the board position. I will note at this point that the game view in the MasterVC does work slightly differently as it does not use pieces but this will be discussed in the next section. As part of this these subpaths are tracked for later. The subpaths are each given a background colour of .clear and the rest is coloured .black. By using boardView.layer.zPosition = CGFloat.greatestFiniteMagnitude this will always display on top of the pieces as they are added to the gameAreaView.

With all of these done the final game view displayed perfectly, as seen in appendix 3.3 and 3.4. Both of these screenshots precede colours being added to the pieces and the number label being hidden, the former using 7 for all numbers, the latter starting at 80 in order to test the limitations of the number sizes, 100 being a good test as it takes up a sizeable amount of space, which should limit themselves based on the size of the piece by reducing font size, not truncating text. Both screenshots also display pieces in the process of animating using gravity.

Historic TableView

The MasterVC in the splitViewController is a tableView containing two sections. The first of these concerns the play options: New Game and Current Game, the second is populated based on the number of games stored in core data.

The first section is simple enough. Both rows will segue to the SecondaryVC, passing the text from their label into the variable passedInput the didSet of which will place the view into the relevant mode. In the second section, the dataString for the historic game will be passed instead, the didSet acting accordingly. In the first section the only thing of note is the actions of the Current Game cell. This will be greyed out if there is no game stored in user defaults. If tapped, instead of segueing an alert will be displayed instead, informing the player that no current game exists. Should there be a game saved then the cell will display the same as the New Game cell, passing on the content of its text label when segueing. Both of these states can be seen in appendix 4.1 and 4.2, as well as the alert in 4.3

The second section uses a separate reusable sell with its own cocoa touch file called HistoryTVCell. This consists for two views given equal size. The right side states who played first and who won, both in the colours of the relevant player. The left side contains two views, one inside of the other, both with identical sizes and a ration of 707:594. These display the game board in much the same way as the SecondaryVC, even using the same methods, such as K.BoardAndPieces.calculateBoardVariables( ), to lay out the view. There is, however, a meaningful difference. Unlike the SecondaryVC, where pieces are their own view added to the base view, pieces here are not generated in the same manner. You may recall that as the subpaths are being drawn they are tracked. Here however, instead of giving them a background colour of .clear, they are instead given a background colour of the player who dropped them. Additionally, a check is made to see if the piece was a winning piece and they are coloured white if they are. This can be seen in appendix 4.4. This prevents potential issues from placing individual pieces and reduces memory use.

An issue did arise here that took a few days to fix, nearly resulting in a compromise having to be made. It became evident that calling tableView.reloadData( ), in order to update the cells when returning from a finished game, was not working correctly. While a new cell would be added, its game would not be displayed, instead displaying the game and text form a different cell. Worse, it became apparent that other cells were swapping around each time the MasterVC was returned to. Additionally, the game shown would not match when a row was selected.

The solution to this was simple enough. I just needed to override the prepareForReuse( ) method in HistoryTVCell and have it call setNeedsDisplay( ). This would force the cell to redraw when it was reused. This issue should have been resolved much faster. This method was my initial solution, but I mistakenly thought it was one of the tableView methods. Once I could not find it I had proceeded to try any number of methods including deleting all cells in viewWillDissappear( ) and reloading them each time. As nothing worked I was looking at a compromise between having the correct number of cells but they mix up, or not updating the number of cells until the app is relaunch. Mercifully, this compromise did not need to be made in the end.

Data Persistence

The app features two forms of data persistence, user defaults and core data, both for different reasons, although the former influenced how the latter works. User default is used to store a string representing how the game has proceeded to this point. An example of this string would be “13414241516”. A method stored in constants, K.Data.decodeSave( ) deciphers the content of this string, returning a tuple containing a Bool, two UIColours, and and array or Int arrays.

To do this the method first looks at the first three characters of the string. In the sample these are 134. The first character can only be a 0 or a 1, a 0 indicting the player went first, a 1 that the bot did. 3 and 4 can both be used with an array in Constants, each being the index of a UIColour, the first representing the players colour, the second the bots. From here, the remaining characters in the string represent locations on the game board. In the sample they are 14241516 which would be decoded into the following array of Int arrays: [[1,4],[2,4],[1,5],[1,6]]. The first digit in each of these is the row, the second the column.

Data is only saved to user defaults once the bot has played. Initially this it was saved using the didSet for dataString, saving every time the string was appended to, but, as the bot plays nearly straight after the user, there did not seem to be a reason to double the number of saves made. A sample of the old code before this optimisation can be found in appendix 5.1 and hardcoded sample string used for testing in 5.2. The dataString does still update during each players turn, adding the location of the dropped piece. This string, and user defaults, are both cleared whenever a game ends or starts, but not during a historic game, allowing the user to view one of these and return to their game.

Core data is only saved to once a game concludes. Initially the plan was to have two entities in core data, one for the games and another for the discs, connected with a one to many relationship. However, with a minor alteration, the same data string could be used for both. At the end of a game the data string is emptied, as is user default. Once this is done the data string is rebuilt, with slight alterations inspired by byte stuffing used in Networks andInternet Systems. By comparing the location of a piece against the winners array a flag can be placed into the string preceding the location if the piece was one of the wining pieces. As the numbers in the string never rise above seven an escape character was not needed to identify the flag. Instead the number eight was used as the flag, a number that will never be used in the string. Even if additional piece colours are used in the future they would be tracked in the first three characters of the string and not effect the decoding. Taking our sample string and adding an eight in as an example, “134142415816”, would now output the following tuple: (true, .green, .purple, [[1,4],[2,4],[1,5],[8,1,6]]). Of course an actual example would have far more positions and four or more eight flags, but this should demonstrate the concept. The viewController, when creating the pieces for a historic game, would then know to colour the piece white and its text the players colour (to distinguish the piece from the opponents) if the first number in an array is eight. A diagram showing the design of the dataString used for core data can be found in appendix 5.3.

This did lead to an unexpected bug that was somewhat annoying to track down, as well as to test the solutions to. Upon playing one of the games back an error in the stored dataString caused the issue that can be seen in appendix 5.4. Here white discs were being placed for both players, more pieces were being dropped than could be placed in a column, and the game crashed at the point shown. This issue only appeared in this particular game, all others playing as expected. The crash was easy to identify, its cause giving the largest hint as to what the issue was. As I will discuss in the miscellaneous section, a second string is stored when the game ends and is unpacked one character at a time as pieces are dropped. The crash was due to this string running out of characters, although it should have the same number of characters as there are pieces, meaning additional pieces were being stored when the dataString was being rebuilt for entering into core data.

Through testing and this clue it became apparent that the issue was in unexpected behaviour from Alpha0C4, specifically in the array it gives for winning locations. Once four in a row was achieved the winning locations were output as such:

[(row: 1, column: 1), (row: 1, column: 2), (row: 1, column: 3), (row: 1, column: 4)]

The expected behaviour would be that if five in a row was achieved then the output would be:

[(row: 1, column: 1), (row: 1, column: 2), (row: 1, column: 3),
(row: 1, column: 4), (row: 1, column: 5)]

Instead the model treated it as two for in a rows, doubling multiple locations:

[(row: 1, column: 1), (row: 1, column: 2), (row: 1, column: 3), (row: 1, column: 4),
(row: 1, column: 2), (row: 1, column: 3), (row: 1, column: 4), (row: 1, column: 5)]

This resulted in the doubled winning pieces being saved twice. A simple break was added to the method that compared the two and the issues was resolved. The problem was that to find the bug and test the solution I had to get the AI to achieve five in a row, something the model was surprisingly good at avoiding.

Ultimately, I did not leave the Games entity in core data as one attribute per game, the dataString. While this was enough for game playback, and is all that is passed to the historic mode, additional attributes were added in order to make generating the tables in the MasterVC's tableView easier. These included Int16s for the colour of the player who played first, who played second, and who won, as well as Booleans to identify who played first, if the user won, and if the bot won (both user and bot were tracked so a draw could be identified). While all of these could have been included in the string used for saving the game, it would have complicated the decoding process by adding additional prefixes to the string, or requiring a second flag to identify when the locations had all be processed. As such, the extra attributes made more sense. A Date attribute is also included to indicate when the game was played. This is used to sort the games when generating the tableView cells, ensuring that the most recent game is at the top. The final version of the Games entity can be seen in appendix 5.6, and the sorting code can be seen in 5.5.

After all this was coded an unexpected issue arose while attempting to tidy to code. Here, every time the MasterVC appeared an extra new game would appear. Relaunching the app cleared all of these, but during the life cycle they would build up. This ghost entry would even appear is there where no saves, all values were set to nil or 0, and they were all identical. As it turned out I had moved the core data context in the SecondaryVC to a point that was too early in the code resulting in the context shifting each time the SecondaryVC appeared, resulting in the context including this ghost entry. Moving the line of code back to its original place resolved the issue but a check I had left in to remove these ghost entries remains in the code to ensure that this does not happen again, see appendix 5.5.

The final area to tackle regarding data persistence is the deletion of entries. For individual deletion swiping on a row will being up the delete option. This uses two tableView methods, editingStyleForRowAt( ) which only returns the delete option here, and commit( ) which tells the context to delete the entry, updates core data, removes the game from the gameList array, and removes the row from the tableView. All of this is sandwiched between tableView.beginUpdates( ) and tableView.endUpdates( ) to prevent crashes in the window where the number of gameList array, and number of rows don't line up.

A button is also present on the top right of the tableView to delete all games. It is hidden if there are no games to display, but will appear as soon as there are any. Tapping it will show an alert, as seen in appendix 5.7, to warn the user that the action can not be undone. Proceeding will essentially perform the same actions as deleting an individual row, but for all games.

Rotation

Unfortunately, while the app is capable of functioning in both portrait and landscape, rotation between both is currently not supported in the SecondaryVC. To use the app in a specific orientation it is best to launch it in the preferred orientation, although rotating while viewing the masterVC largely works. This comes down to a simple decision on how to calculate the games variables made early in development that, with current knowledge, could have been easily avoided.

As it stands, so long as the device is rotated while viewing the MasterVC there are no major issues when rotating the app, so long as a tableView cell is selected to segue to the SecondaryVC. Unfortunately rotating during a game will cause issues as can be seen in appendix

6.1 and 6.2, although rotating back will fix these so long as no pieces are dropped while rotated. The reasoning for this is that all the game board coordinates, such as barriers, piece locations, and drop positions, are all calculated and placed using the x y coordinates in relation to (0,0). This works fine for generating these once all view etc have the final locations and sizes, but causes issues with reorientation when calling viewWillTransition( ).

This method is called when the view is about to rotate. On of its arguments is 'size', the size of the screen being transitioned to. Unfortunately, due to how everything is currently calculated, this does not help with rotating elements. Instead, had everything been laid out using the midX and midY points, instead of (0,0), then ending this function with a call to currentGame, with some features such as the music and flash deactivated, should work. A check would be needed to see if the gameAreaView is less than the standard 707:594, such as on an iPhone, in which case variables would need to be calculated using the new size provided, but they could be. Other than that, as the game board has a maximum size, some variable, such as height and width, would just need to be swapped, and some wouldn't need to be recalculated at all. Since everything would then be laid out in relation to the centre of the screen, using midX and midY, everything could be placed before the rotation happened. After that calling the current game mode would place all the pieces in their positions on the new board.

Instead, by the time this solution was apparent a large refactor and testing period would have been needed to implement it correctly. Finally, historic mode would also still need addressing as it would need to clear the board and stop dropping pieces. Some of the code in the historic mode would also need to be altered in order to be able to proceed from a certain point, placing the other pieces without animation, although musicString.count( ) could help in this as it would return the number of pieces left to drop.

It is disappointing to not see this feature realised, but it would have had to have been implemented from the beginning of the app to ensure no issues.

Misc

To achieve my vision of the app some extra features were added. The first of these is a demo mode. This cannot be triggered by the user but is present at start up. Pieces are dropped at random x locations over the game board's width, contain an emoji instead of a number, and use a random player colour. Each iteration of the app will set the timer for these to be placed at a different interval between 0.01 and 1 second inclusive. As leaving this running for a while can cause slow down due to memory, discovered by leaving the simulator running for a few hours, although the timer interval will effect this, once 1,000 pieces have been created the oldest will be removed. Regardless of how small the interval on the timer is the oldest piece should be off the screen by this point. This was tested on the largest iPad pro in portrait mode with no issues, although slowdown can still occur so this will need to be tested further to perfect this mode. Samples of some of the extremes can be seen in appendix 7.1 and 7.2.

A Constants.swift file was also created in order to store some static variables and functions that may be used in multiple locations. Variables include the array of colours, as well as segue identifiers, of which there was originally going to be more than one, reusable cell identifiers, and keys for user defaults. The functions also include the function to decode the the saved data string, as well the one to calculate all the game board variables, such as piece diameter and column width. These methods were placed in the constants file as they are called in multiple locations, such as the SecondaryVC and the HistoryTVCell.

A third additional feature was the ability to choose from more than two colours. This was a complicated feature to implement as the Alpha0C4 model only expects two colours to be in use. To implement extra colours the variables the model uses are largely ignored bar for the when the model needs them, such as checking who won. Instead extra variables are used to track these and code was created to ensure the correct one was used even though it won't line up with what the model sees.

An array was created in the Constants file in order to more easily track these colours. This allowed a number to be stored in core data giving the index of the colour used, as well as in the save string. All the colours can be seen in appendix 7.1 and 7.2, as can the buttons for the user to pick their colour in 7.3. The bot's colour is then randomly picked in a repeat while loop, repeating the loop if the bot picks the same colour as the user.

Finally, part of my motivation of making this app was to try out iOS features that I had not had to opportunity to use before. This was a large part of my motivation in using the splitViewController. It was also my motivation in wanting to make the app musical. As such the MusicModel was introduced. This uses the VAudioPlayer( ) from AVFoundation to play a note every time a piece is dropped. An array of strings store the file names of each note and a method called playMusic( ) takes in one of these names and plays the music stored in the file.

Five notes are used: C, D, E, G and A, the five notes of the pentatonic scale. This scale is being used as, in theory, any combination of notes will sound somewhat pleasing, due to the absence of complications caused by the mediant and leading note, as well as chromatic notes. When a new game is started the first note played will always be C, tracked using a bool, as this gives the listen a key. After that, dropping a piece will result in one of the five notes being played at random when a piece i dropped. A repeat while loop is used to check it is not the same note as the last piece, randomly picking a new note if it is. The index of these notes in the array of musicNotes is saved to a string that is stored in user defaults at the same time as the games dataString. This musicString is also saved to core data at the end of the game so that the historic game will play the same piece. Each of these notes is slightly less than a second so that they don't overlap, pieces only being allowed to drop once a second has elapsed as mentioned earlier.

Upon the game concluding one of three ending themes will play, the positive, "Good" ,ending if the player won, the negative, "Bad", ending, modulation to the parallel minor key, if the bot won, and the neutral, "Draw", ending, using an interrupted, I - V - vi, cadence to single that the game was inconclusive, in the case of a draw. These are stored in the musicString using their index from the musicNotes array. A final, tune is used in the case of a current game being played. As it is unreasonable to play all preceding notes, but the tonic C needs to be established, a small opening tune, called "Current", is played instead, with the flash being timed for the last note of the tune, indicating the user can continue the game. As this is the only time this tune is used it is not stored in the musicNotes array, or the musicString. All the notes and tunes were created in Garage Band although they are somewhat robotic, lacking dynamics or emotions. To remedy this multiple version of each note, at different dynamics and feels, could be created and picked from randomly, eg. C1 is found, C2 is soft, etc.

There is a known bug with this feature in that once a historic game is started all its notes need to be played. This can cause sounds to overlap if a new game is started, or a second historic game is started. It causes no issues beyond this so it has yet to be addressed.

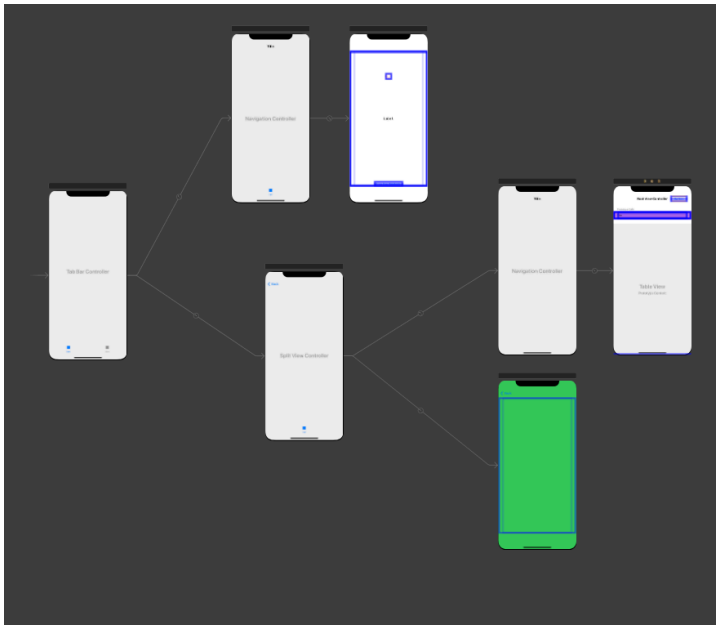
Conclusion

Overall I wanted this to be an application I could be proud of and put my own spin on. While it is unfortunate that I was not able to implement the screen rotation perfectly, I believe if I make a similar project in the future this would not be the issue it is here. Ultimately, I am happy with the final version, had the opportunity to experiment with various technologies I wanted to explore, and feel like I have done enough to make it my own.

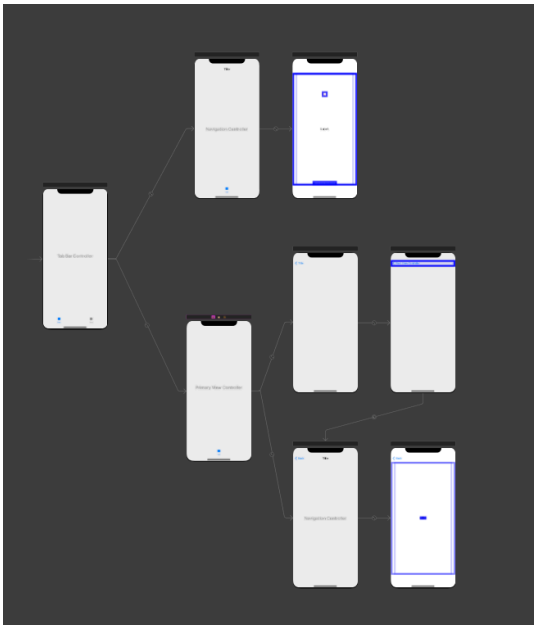
Appendix

Section 1

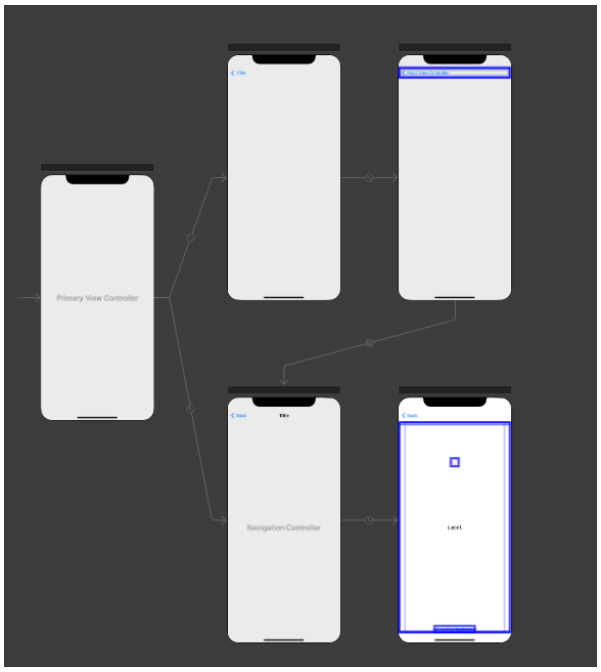
1.1



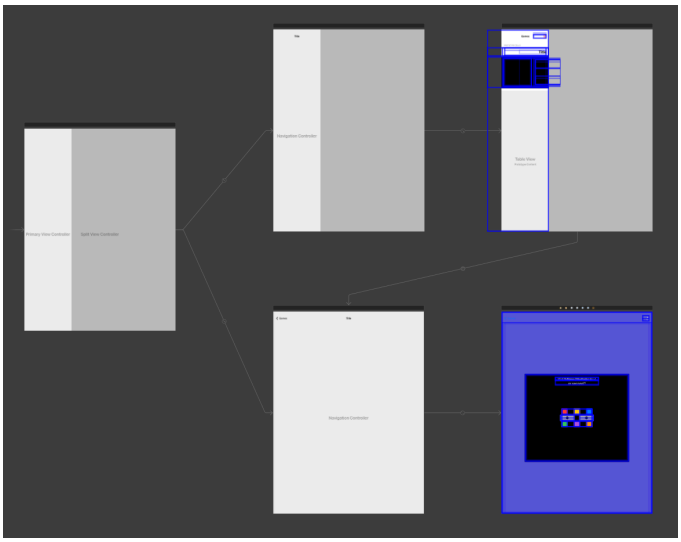
1.2



1.3

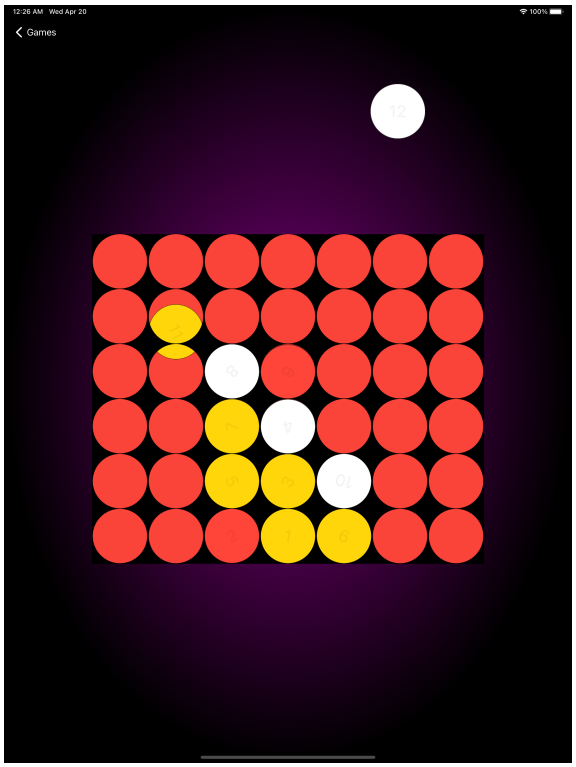


1.4

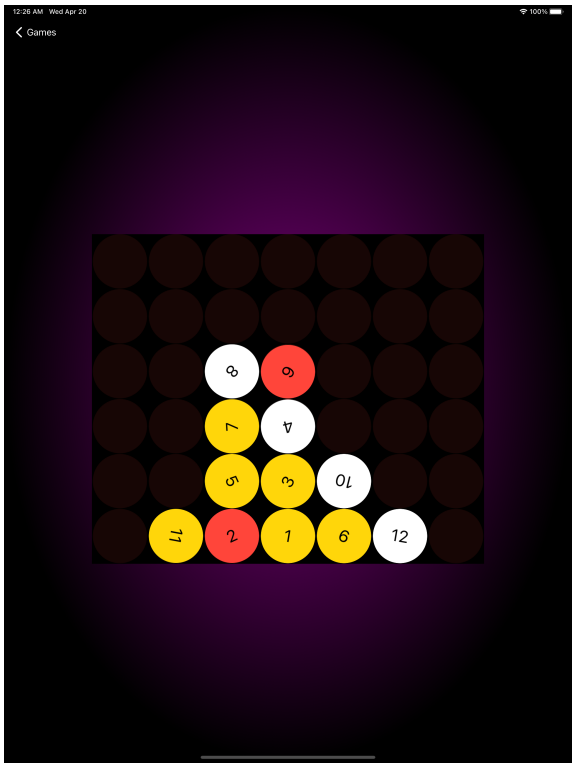


Section 2

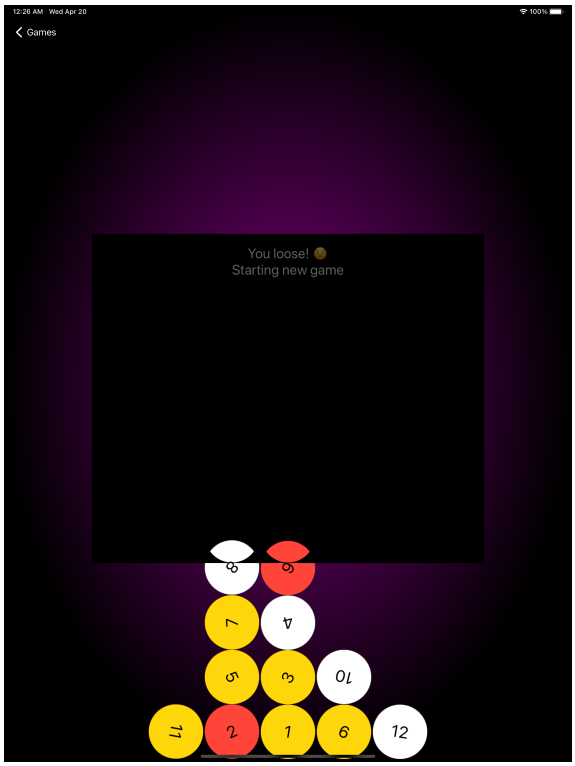
2.1



2.2

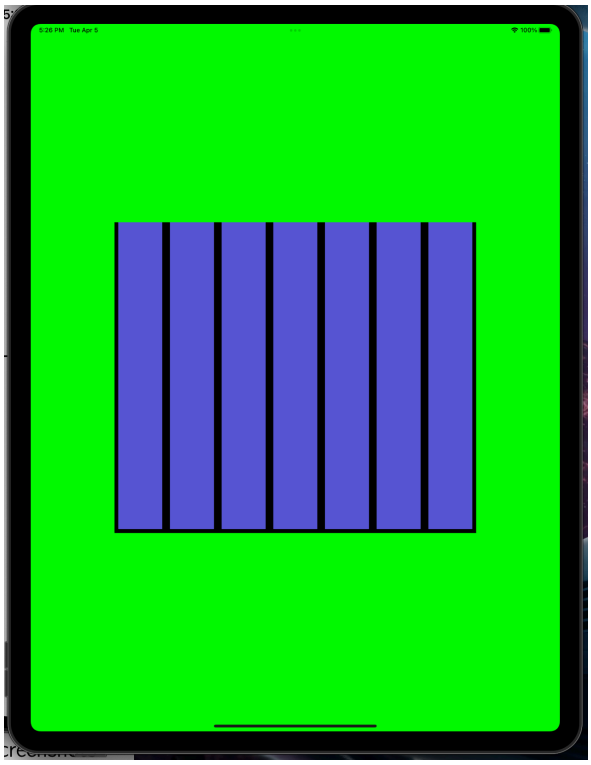


2.3

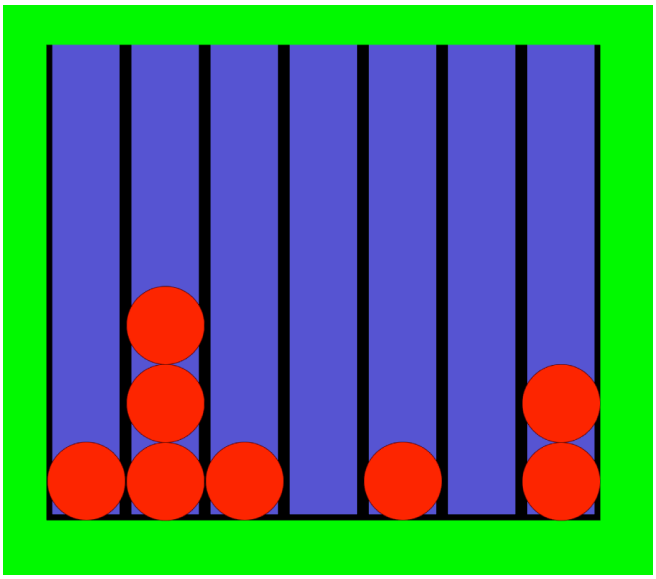


Section 3

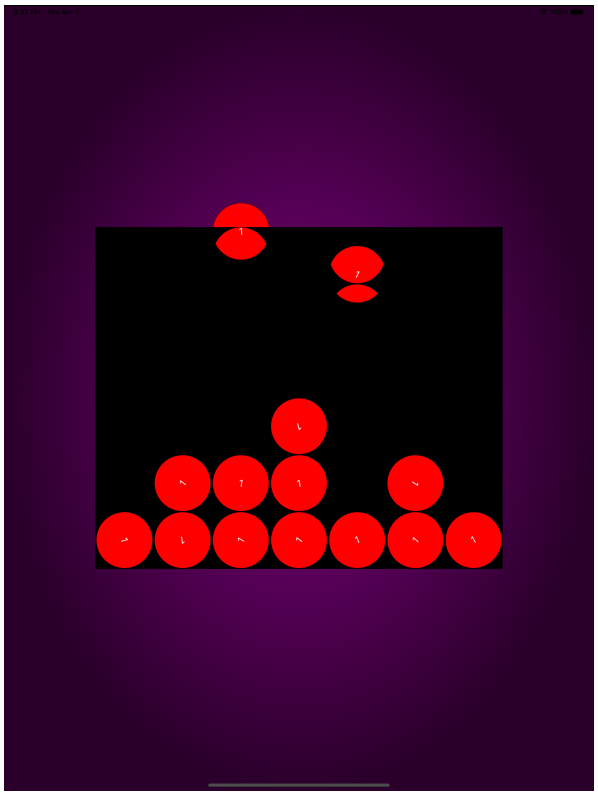
3.1



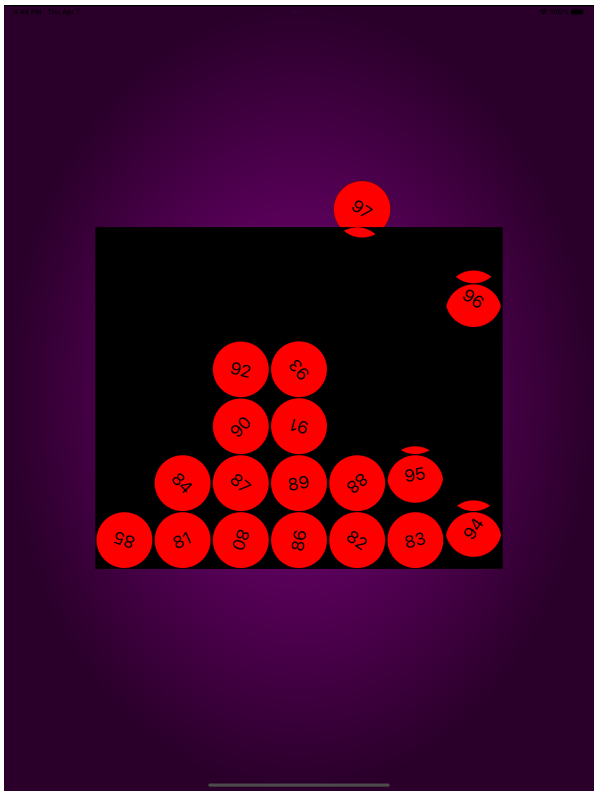
3.2



3.3

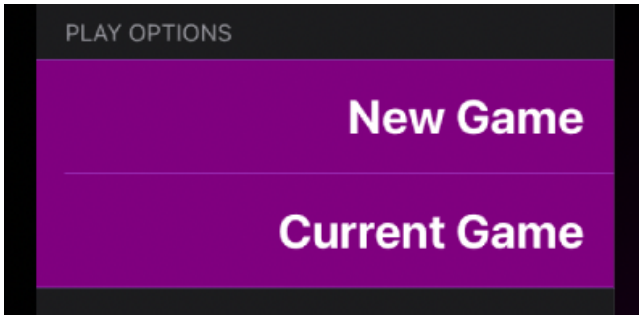


3.4

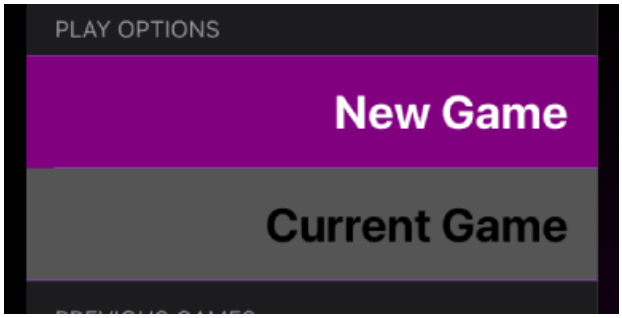


Section 4

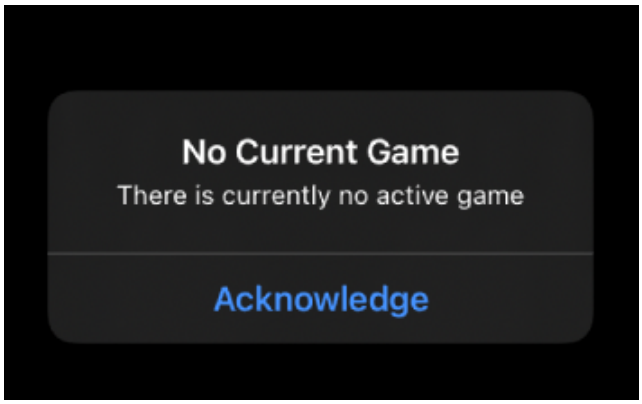
4.1



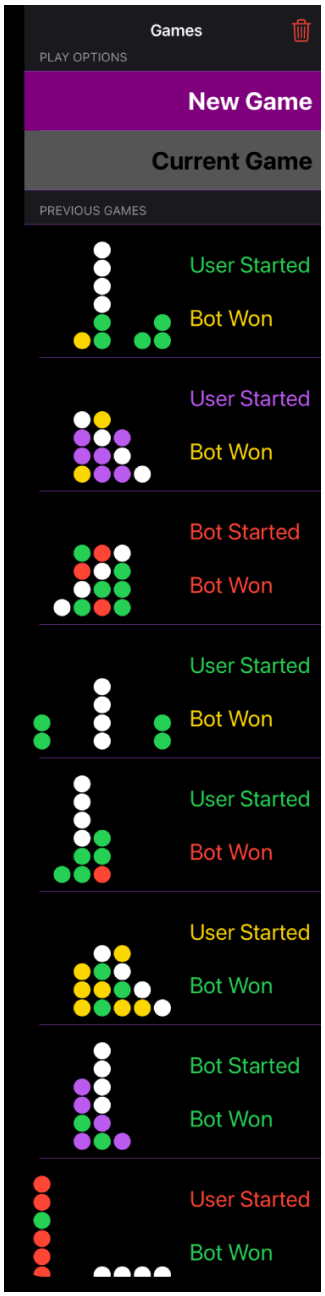
4.2



4.3



4.4



Section 5

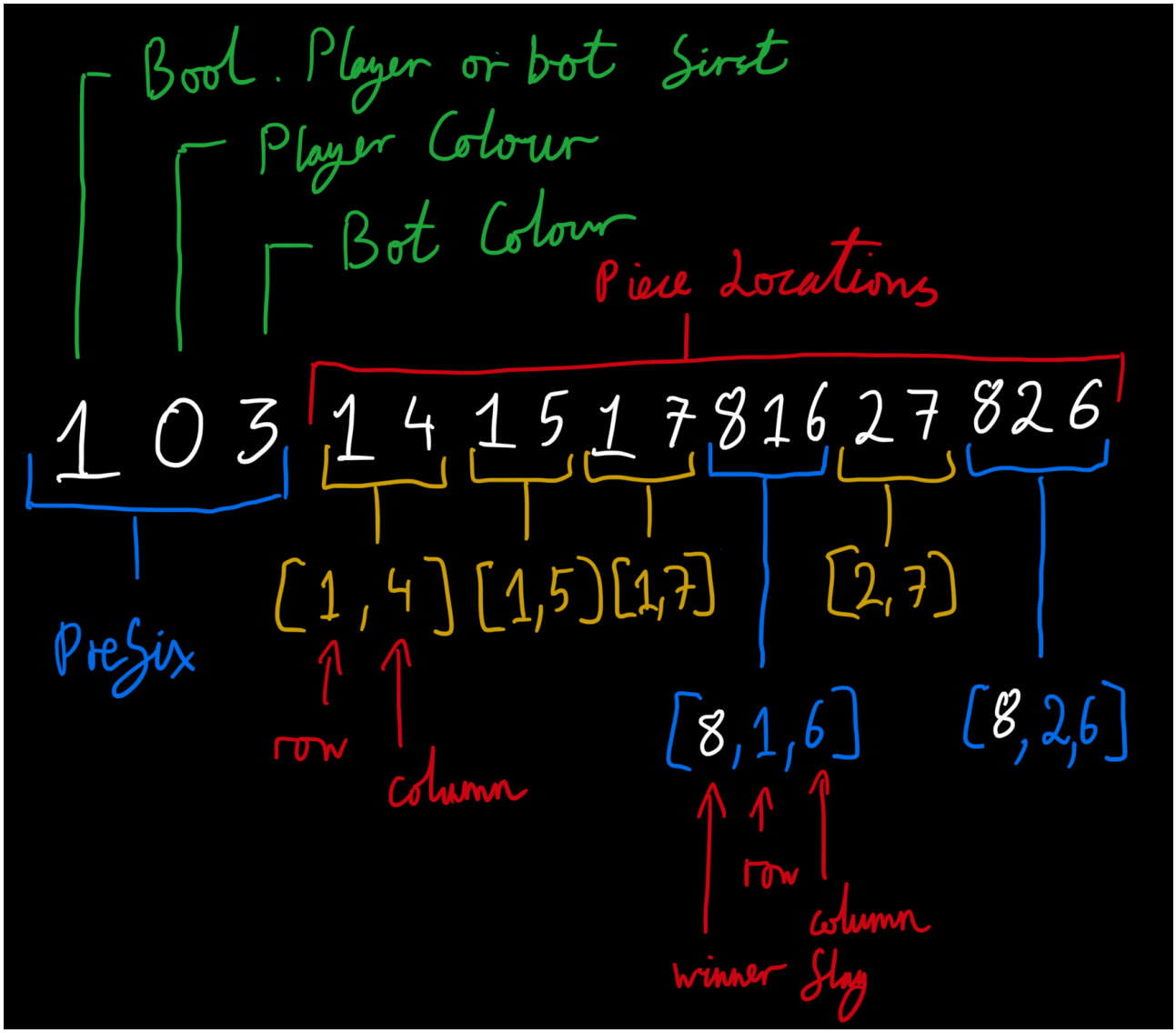
5.1

```
var dataString: String = "12114243444152535 45" {
    didSet {
        K.Data.defaults.set("\(dataString)", forKey: "dataString")
    }
}
```

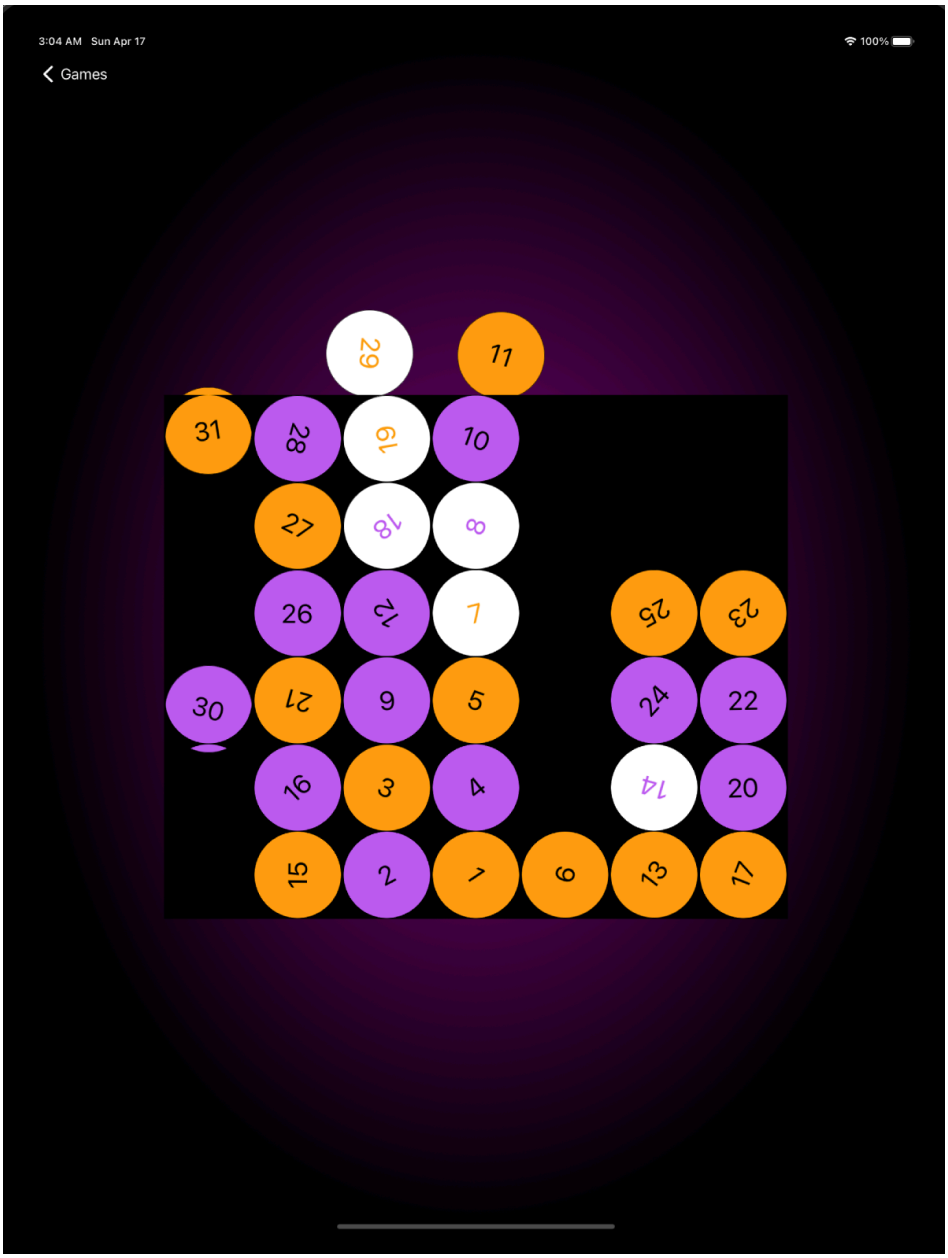
5.2

```
var dataString: String = "128112233448122324134285423148 2356"
```

5.3



5.4



5.5

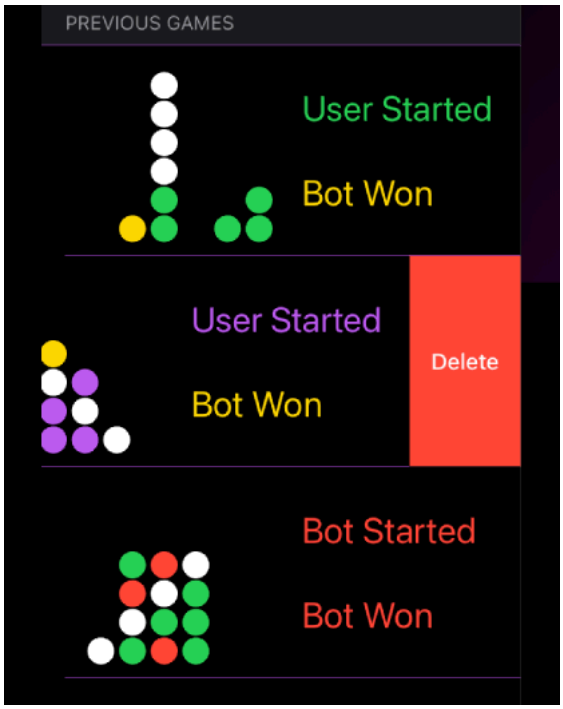
```
// Fetch Games data from core data and sort by date so the most recent is displayed at the top
gameList = try K.Data.context.fetch(Games.fetchRequest()).sorted { itemOne, itemTwo in
    itemOne.dateCreated! > itemTwo.dateCreated!
}
```

5.6

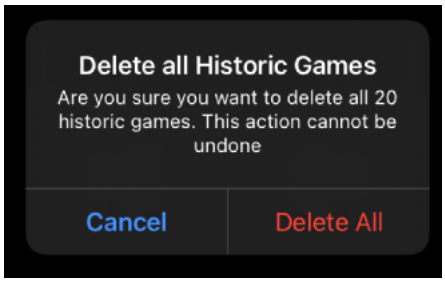
5.7

```
/* After adding Boolean values to the loading from core data would result in an extra entry being returned
    with its values set to false and nil.
 This next section checks for these and removes them.
 Issue was resolved as it was caused by the context being set up in the secondary view too early. Check
     remains as a precaution against a repeat of the error.
 */
for game in gameList {
    if game.gameString == nil {
        print("NILL FOUND")
        gameList.remove(at: gameList.firstIndex(of: game)!)
    }
}
```

5.8



5.9



Section 6

Fig 6.1

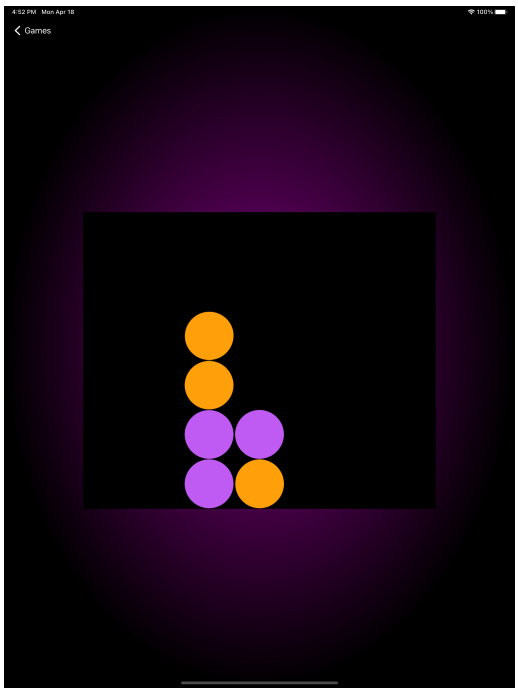
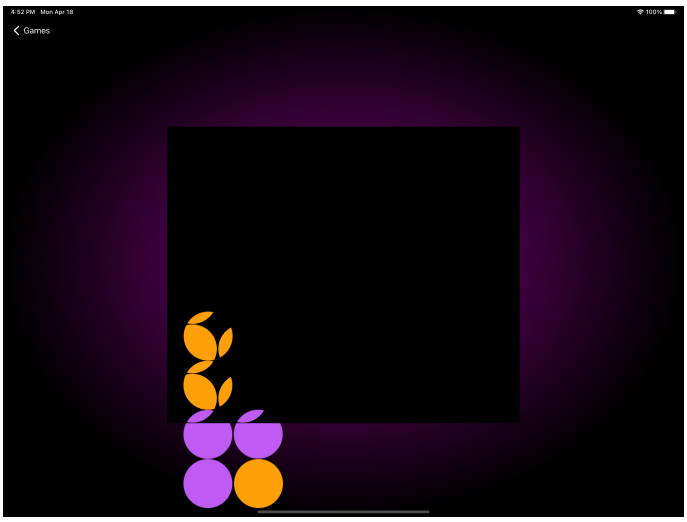
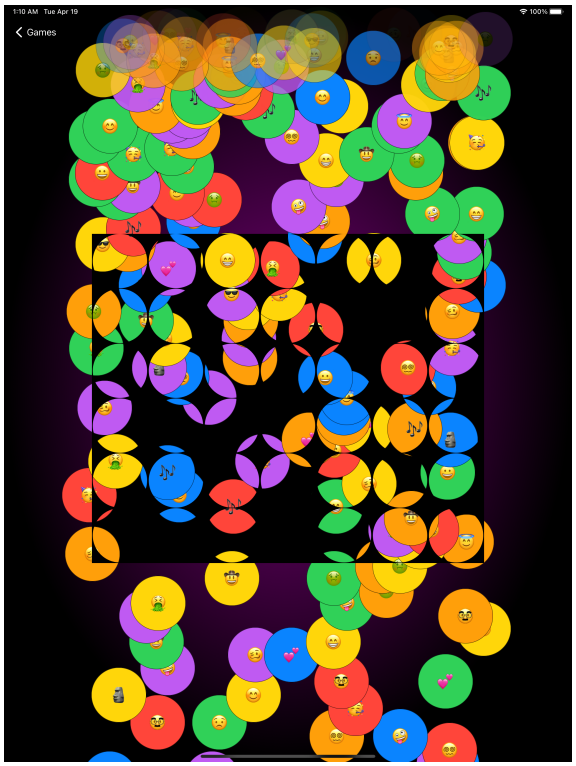


Fig 6.2

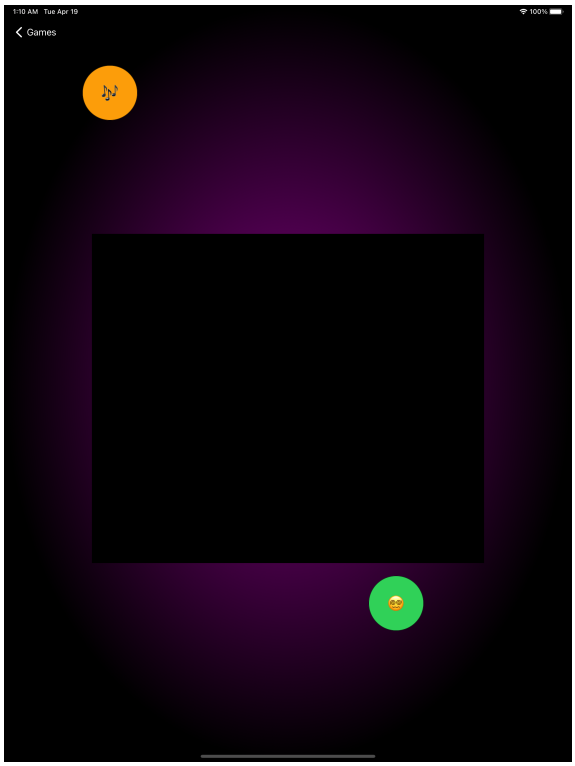


Section 7

7.1



7.2



7.3