Saturday, March 21, 2020

Connect4 - IOS Development

Ga Jun Young - 16440714

Architecture: To begin my development approach I devised a basic architecture. The architecture consisted of core components that were considered easy enough to implement. The architecture is as follows:

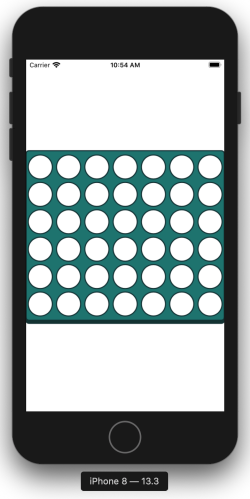
1. Game Board & Wedge (draw)
2. Discs (subviews)
3. Behavior (Colliders, Gravity, ItemBehavior)
4. Gestures + Game Mechanics- tap, swipe (IBActions)

GameBoard

I began with designing the game board. The calculations involved finding the center-point and the point in which the game board should be drawn from. Using the knowledge of the game board being a 6 x 7 dimension. I devised a rounded rectangle calculation. The grid width was calculated through dividing the width of the minimum bounds (height / width) by the number of columns.

```
// Rounded board
let boardRect = CGRect(
    x: delegate.startPoint.x,
    y: delegate.startPoint.y - delegate.gridWidth * CGFloat(BoardConstants.rows),
    width: delegate.gridWidth * CGFloat(BoardConstants.columns),
    height: delegate.gridWidth * CGFloat(BoardConstants.rows))
let roundedBoard = UIBezierPath(roundedRect: boardRect, cornerRadius: delegate.gridWidth *
    BoardConstants.boardCornerRatio)
path.append(roundedBoard)
```

Having drawn the rectangular board, I started drawing circles which were incircles of my grid cells. This was done with UIBezierPath. In order to cut circles from the board I had to rely on CAShapeLayer. The “fillRule” using .evenOdd enabled the circle shapes to be cut out of the rectangle.



The image on the left hand side is the result of using CAShapeLayer with various variable settings. The wedge is also drawn on top of the board. This wedge can be easily removed or added by appending its calculation to a specific public UIBezierPath variable in the Connect4View.

For the underlying calculations I designed a delegate model called Connect4DataSource. The Connect4Model was used to handle the necessary calculations which were then initialized and passed on from the Connect4VC. As a result, the IBDesignable was rather useless as it did not enable development to visualize the game board on the storyboard.

The MVC compactly reduced code in the view and later on I had to use a lot of the model functions in the VC.

```
init(frame: CGRect, move: (action: Int, color: Color, index: Int)?) {
    self.move = move ?? (action: 1, color: Color.yellow, index: 1)

    label = UILabel(frame: frame)
    label.center = CGPoint(x: frame.width / 2, y: frame.height / 2)
    label.textAlignment = .center
    super.init(frame: frame)

    // Create the layer for displaying the disc
    if self.move.color == Color.red {
        backgroundColor = ColorConstants.redInnerDisc
        layer.borderColor = ColorConstants.redOuterDisc.cgColor
    } else {
        backgroundColor = ColorConstants.yellowInnerDisc
        layer.borderColor = ColorConstants.yellowOuterDisc.cgColor
    }

    layer.cornerRadius = (frame.size.width / 2.0)
    layer.borderWidth = layer.cornerRadius * BoardConstants.outerDiscRatio
}
```

Discs

Following on from the gravity bubble tutorial, I was able to create disc shaped bubbles by making a DiscView.

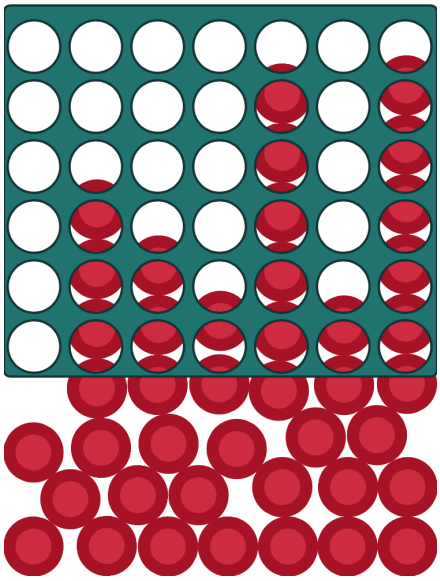
Instances of DiscView became subviews to my Connect4View. My DiscView took several iterations of changes.

Before, it would only drop a disc with a color but later on, it took in the “move” as parameter to label the discs with its index, color, and action.

The default parameter is just the frame of the square which is then rounded off into a disc. The collider type is also “.ellipse” as it is round.

Behavior

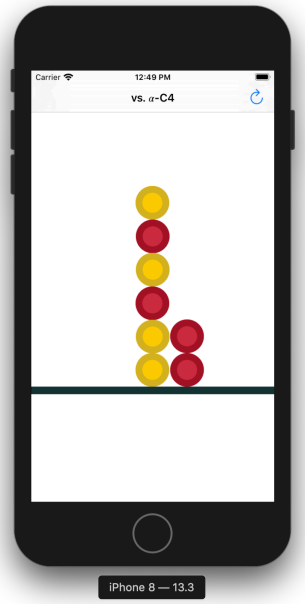
The picture on the right shows the behavior of the discs. They collide, fall due to gravity, and have relative resistance and elasticity.



This is done through DiscBehavior, similar to GravityBubble. However, later on I remove the bounds as colliders and added resistance and increased elasticity to the item behavior. This enabled discs to fall at a rate that it wouldn’t squish the other discs entirely. Originally, the disc views would become distorted.

A unique aspect to this behavior is that I added a gravity.action. This action would then check if the subview is out of bounds while it is falling due to gravity. If it is true, then the discView is removed.

Saturday, March 21, 2020



In Connect4VC, I added the UIDynamicAnimatorDelegate, along with animator to coincide with this delegation between DiscBehavior and UIDynamicAnimator.

The wedge and barriers are calculated in the VC and passed onto the Connect4View to draw the wedge and pass it onto the DiscBehavior to add colliding boundaries for the discs. As I didn't want to show the barriers, I did not draw them onto my Connect4View, but the colliders are present.

After I created the colliders, I moved onto several gestures. I added a refresh button on the top right of the Connect4VC. I embedded the VC onto a navigation controller, producing a title bar.

Gestures + Game Mechanics

I started with the tap gesture. This was the bread and butter to enabling the player versus AlphaC4. My idea of tap is as follows:

1. Tap only on **regions on or above the game board** to drop a disc.
2. If the column is full, tapping on that area is disabled.
3. A player tap would automatically call the AI to move next.
4. Any locking conditions are mainly handled using gameSession object methods.

Note: I reused a lot of the default code provided in the ViewController at the start so that I can print out the board on the console to see if the discs matches with the printout.

```
// MARK: - IBActions
@IBAction func tap(_ sender: UITapGestureRecognizer) {
    if !gameSession.done {
        let point = sender.location(in: self.connect4View)
        let width = gridWidth
        let yLine = startPoint.y

        DispatchQueue.global(qos: .userInitiated).async {
            for column in 1...BoardConstants.columns{
                if point.x < width * CGFloat(column), point.y < yLine {
                    // tappable area

                    if self.gameSession.userPlay(at: column - 1) {
                        Thread.sleep(forTimeInterval: BoardConstants.sleepTime)
                        if let move = self.gameSession.move {
                            DispatchQueue.main.async {
                                self.dropDisc(move)
                                print("\(self.printBoard("Player", [move.action], move.color == Color.red ? "X" : "O"))")
                            }
                        }
                        self.playAI() // AI's turn
                    }
                    break // not a valid drop location
                }
            }
        }
    }
}
```

For this tap function, I used threads to separate and delay inputs. That way both the AI and player would have a small waiting period before they can drop a disc. The gameSession.userPlay checks if a play is possible and only then can a disc be dropped. I did some basic calculation using modulo + 1 to figure out where the move.action would go in the column.

The AI would then take its turn and a gameSession.move would be produced. The code for the AI is very similar. But at the end of the AI's method, it would check if the game is done. If so, display outcome message which is sent to the UILabel.

The swipe gesture and the refresh gesture uses the same IBAction method called resetGame. Which calls a more general method called resetGameSession. This second method checks if it should delete the wedge or not to let discs to fall out. "resetGameSession" is also called at the start without deleting the wedge.

```
private func resetGameSession(_ isDelete: Bool) {
    if isDelete {
        deleteWedge()

        DispatchQueue.main.asyncAfter(deadline: .now() + 0.2, execute: {
            self.present(self.showAlert(), animated: true, completion: nil)
        })
    }

    if connect4View.subviews.count < 2 {
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.2, execute: {
            self.drawWedge()
            self.gameStart()
        })
        outcomeLabel.text = " "
        self.title = "vs. α-C4"
    }
}
```

Alert Controller

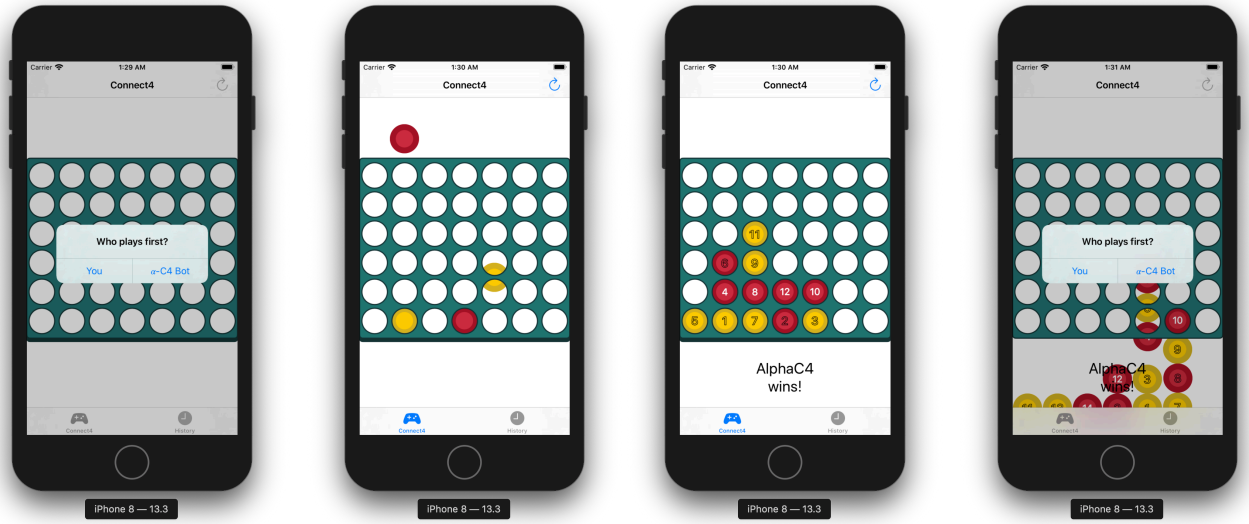
After this set up, I created an alert controller which would ask the player if they would like to play first or second. Depending on the player selection, a specific

```
// Alert controller to decide who plays first
private func showAlert() -> UIAlertController {
    let alertController = UIAlertController(title: nil, message: "Who plays first?", preferredStyle: .alert)
    alertController.addAction(UIAlertAction(title: "You", style: .default, handler: { action in self.gameSession.botStarts =
        false
    }))
    alertController.addAction(UIAlertAction(title: "α-C4 Bot", style: .default, handler: {
        action in self.gameSession.botStarts = true
        DispatchQueue.global(qos: .userInitiated).async {
            self.playAI()
        }
    }))

    return alertController
}
```

boolean value would be set onto the botStarts variable. If the AI starts first, the AI would call the gameSession.move to carry out the first move.

Game States

The following images show the startup screen, tap gesture disc creations, game outcome, and refresh (left to right).



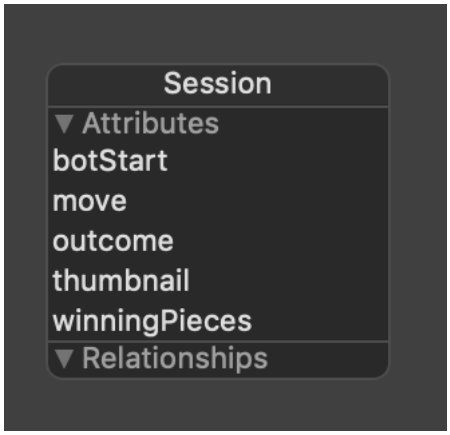
In coding this first section to this application, I had to make vital decisions on how the UI is displayed.

1. The moment the winning action is made, the discs show their indices. This means that the disc that is dropping will show their index. I have done this because it may take too long for the disc to drop and halt. Meaning that the user will have to wait several milliseconds. This is undesirable especially if the user wants to play the next game.
2. Notice in the second image that the AI (red) drops immediately after the player (yellow). Again, this is to speed up the gameplay for the user. I feel like it is undesirable in making a user wait.
3. In the last picture, the user can either swipe or refresh to restart the game. After a thread wait, the user can reselect the player that will go first. Meanwhile, the discs behind are falling out of bounds and being removed permanently. Note: while the wedge is gone, the player cannot tap to drop discs.

Core Data and Tab Bar Controller

Persistence in this game only occurs when a game has an outcome. The game cannot be continued on as we cannot access the internal API of GameSession and as a result, we cannot save it into core data. It would not be possible to reuse GameSession to carry out the AI's saved moves. **Instead, there are several variables that are public that will help with replays.** These variables are:

1. GameSession's move
2. GameSession's botStarts
3. GameSession's outcome message
4. GameSession's winning actions
5. Additionally, a screenshot of the game board will be necessary to show the game that was played in the HistoryCVC (UICollectionViewController).



The overview: of the core data is as follows. A PersistenceService class contains the core data container. I moved the AppDelegate core data methods specifically here and made them static so I can easily save and retrieve. In the Connect4VC, I have a method that will call and save the session, once the "gameSession.done" is true. Each individual values will be stored in their respective attributes (shown on the left). Furthermore, a screenshot of the game board will be taken at the same time to store under thumbnail as "Data".

Retrieval: Core Data "Session" is retrieved in the HistoryCVC (UICollectionViewController). Like a table view but instead the view has grid cells.

```
func fetchData() { // Core Data fetch sessions
    sessions.removeAll()

    let fetchRequest: NSFetchRequest<Session> = Session.fetchRequest()

    do {
        let sessions = try PersistenceService.context.fetch(fetchRequest)
        self.sessions = sessions
    } catch {}

    self.collectionView.reloadData()
}
```

The array of sessions is retrieved in the fetch method and stored in a private variable. Using UICollectionViewDataSource I can then display my session data onto small collection cells. Each cell will display a screenshot of one of the last moments in the game, the outcome message, and the total amount of turns required to finish the game.

Note: the screenshot is a custom method I created in extension to the UIView. It takes precise coordinates to capture the image of the board which is possible since, we know where the board starts and end.

These cells are clickable and will segue to a new Connect4VC, where only the replay of the session will be carried out.

Replay Functionality

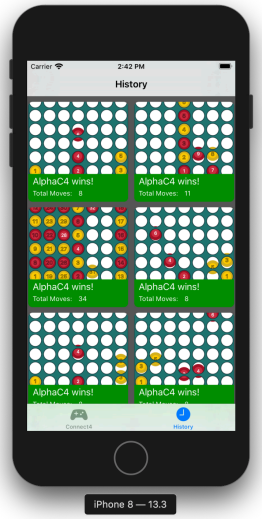
In the beginning, I created a ReplayVC class but because a lot of the methods would be the same as Connect4VC, I decided to not have this redundancy and just reuse the Connect4VC. I refactored multiple methods to conform to the new retrieved session data which is optional.

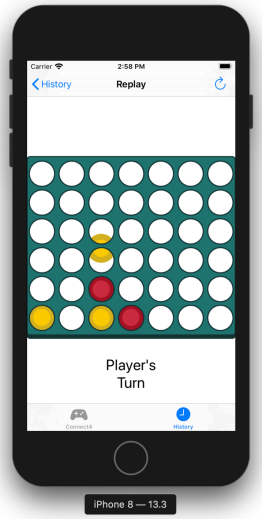
That is if the session data is nil, then we are in a playable game session else only replay versions of the methods are enabled.

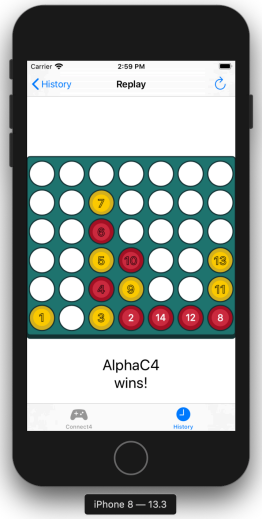
```
private func replay() {
    if !moveHistory.isEmpty { // Drop colored disc
        let singleMove: Move = moveHistory.remove(at: 0)
        var myMove: (action: Int, color: Color, index: Int)

        if singleMove.color == 0 { // Yellow disc
            myMove = (action: singleMove.action, color: Color.yellow, index:
                singleMove.index)
        } else { // Red Disc
            myMove = (action: singleMove.action, color: Color.red, index:
                singleMove.index)
        }

        dropDisc(myMove)
        displayTurnLabel(singleMove.index)
    } else {
        outcomeLabel.text = replaySession!.outcome
        let intWinningPieces = replaySession?.winningPieces?.convertToInt()

        displayDiscLabel(intWinningPieces)
    }
}
```

When a replaySession is not nil, the replay method gets called until all moves are played. Each move turn will show which player is putting in the disc. Furthermore, after the replay is done, the user can replay the session again by swiping or tapping the refresh button.

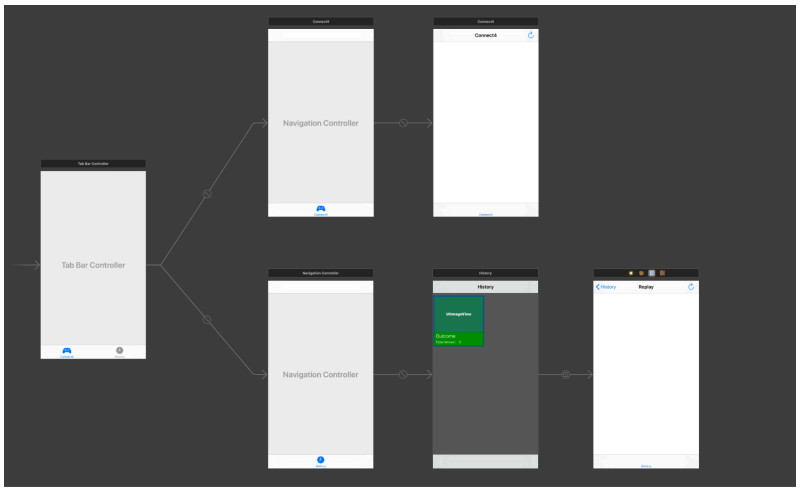
Note: I created Array extensions to convert Int arrays to Int16 array and vice versa for core data storage.







Phone Display Results: The following images are the HistoryCVC display, and Replay (Connect4VC) display, and finally the last finished state of replay.



StoryBoard

The image bellow is the main storyboard. No UIImageViews are present in both Connect4VC. However, there is a cell in the HistoryCVC which represents the cell of the past sessions. A tab view controller is used to move between the playable game session and the collection of cell views.

Threading: Throughout this whole project there are a series of threads being used. A main thing i've noticed is that the screenshot may freeze the frame a bit. But I have fixed this issue by delaying it's thread and setting the drawHierarchy to false. So far from what I have noticed, there is no frame freezing.