BREAKOUT

COMP41550

James Dorrian
13369451

Application Overview:

The first step was setting up the MVC. The model in my game is called BreakoutBehaviour and it is where I will define the dynamic behaviors for my game including: gravity, collisions and specific object behaviours. The ViewController is where I place my gameView and controlled the subviews such as the ball, paddle and bricks – each of which had their own class defining unique characteristics for each. Finally I created the settings view controller which is a tableView offering users the ability to customize the game in terms of bounciness, number of bricks, colour of ball etc. *GlobalAttributes* is where the global variables were defined.

This report outlines the logical and methodical way in which I built this application and provides screenshots and code snippets to aid in this description as suggested in class.

Step 1: The Paddle

As the paddle needs specific and unique operations in terms of movement and visual aesthetics, it required a class of its own, called *Paddle.swift*. In this class the width, height, colour and distance from ground were defined. This allowed me to create the first visual element of the breakout game and I added it to the gameView in my ViewController class. At first the paddle couldn't move so I set about making this my next task. I added the UIPanGesture to the storyboard and connected it to the ViewController via an *@IBAction* called *PaddleMover()*. This function allowed me to interact with the paddle directly and using the move () function in *Paddle.swift* (which takes the *UIPanGesture* velocity as input) I was able to move the paddle across the X-axis of the game view screen. UIView.animate was used to reposition the paddle based on the velocity and direction of the PanGesture as shown below:

```
UIView.animate(withDuration: 0.0, delay: 0.0,
    options: [UIViewAnimationOptions.allowUserInteraction, UIViewAnimationOptions.curveEaseIn],
    animations: {
    self.center = newCenter
},
    completion: nil
)
```

One early problem encountered was that the paddle was not forced to reside within the default screen boundaries. This was not a problem yet but it would make the game unpleasant later on when the ball and blocks were introduced. To stop the paddle moving off-screen the left edge of the paddle was checked against the left edge of the screen and if it was less than the gameView edge then the paddle was repositioned to be inside the screen. The same was applied to the right edge of the screen - once again if the right edge was outside the gameView right side then the paddle was repositioned. This algorithm is depicted in the below code snippet.

```
func setNewCenter(_ proposedCenter: CGPoint) -> CGPoint {
    var currCenter = proposedCenter //proposed center is actual center
    //paddle co-ords
    let centerLeftPaddle = floor(currCenter.x - (paddleWidth/2))
    let centerRightPaddle = ceil(currCenter.x + (paddleWidth/2))
    //screen co-ords
    let screenLeftEdge = floor(referenceView.bounds.origin.x) //0
    let screenRightEdge = ceil(referenceView.bounds.width)

    //left edge check
    if centerLeftPaddle <= screenLeftEdge {
        currCenter.x = floor(CGFloat(paddleWidth/2))
    }
    //right edge check
    else if centerRightPaddle >= screenRightEdge {
        currCenter.x = ceil(referenceView.bounds.width - (paddleWidth/2))
    }
    return currCenter //return the modified center to await animation
}
```

All that remains to be done to the paddle at this stage is to deal with object collisions. This will be discussed after I have touched on the different animation and behaviours of the ball.

Step 2: The Ball

I started by creating a *Ball.swift* class just like we did with the paddle. Both of these classes have init() initializers signaling that they are both *NSObject* subclasses and instead of the default *CGRect.zero* we pass in CGRect and edit the aesthetics of the object using *layer.MODIFIERS* such as *layer.cornerRadius, layer.borderColor*, *layer.backgroundColor*, etc. The frame size and location relative to the gameView are set in the *ViewController* class before instantiating the ball. I found this aspect of the assignment poorly documented because apple do not document inherited classes, they require users to look through the master classes to find the relevant method information.

Next I added behavior to the ball. This involved invoking the breakoutBehaviour class and sending the instance of the ball to this class. Here it gains gravitational, elasticity, resistance, friction and rotation behaviours. These behaviours are defined in the below code snippet:

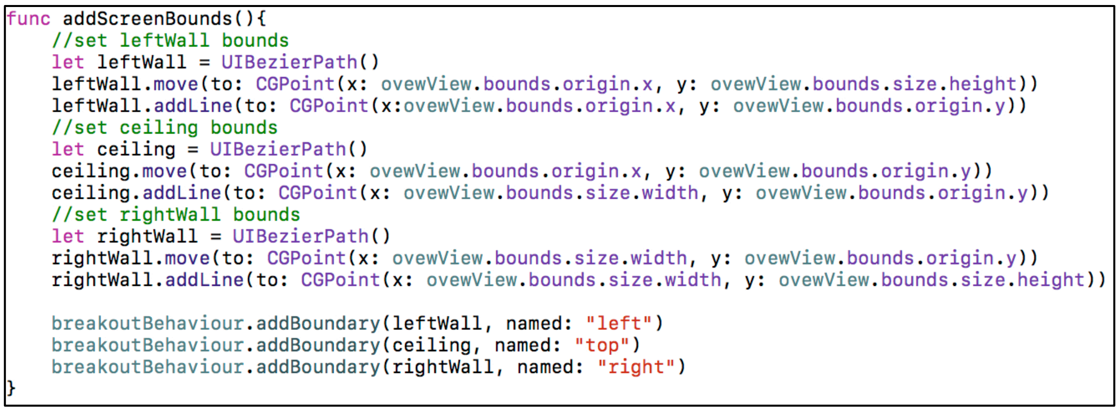
```
lazy var ballSpecificBehaviour: UIDynamicItemBehavior = {
   let behaviour = UIDynamicItemBehavior()
    behaviour.elasticity = GlobalAttributes.sharedInstance.bounce
    behaviour.resistance = 0
    behaviour.friction = 0
    behaviour.allowsRotation = true
   return behaviour
}()
```

The next task undertaken was to add collisions. The collision between the ball and the paddle was easily configured; I simply added boundaries for the paddle in ViewController, ensuring that *UICollisionbehaviour()* objects cannot pass through. This is the *createBall()* method found in ViewController class and we can see if references the original ball class giving the frame as input as well as adding the ball to the gameView and adding the behaviours using *breakoutBehaviour.addBall()*.

```
func createBall(){
    var frame = CGRect()
    frame.size = CGSize.init(width:15, height:15)
    frame.origin.x = paddle!.pos.x+(paddle!.paddleWidth/2)-(15/2)
    frame.origin.y = paddle!.pos.y-(15)
    let ball = Ball(frame: frame)
    balls.append(ball)
    breakoutBehaviour.addBall(ball)
    ovewView.addSubview(ball)
}
```

Inside BreakoutBehaviour (the model of our MVC) we add the collision and gravity behavior to the ball making it interact with the walls and fall downwards according to real world physics. The application is beginning to resemble a game at this point.

Similar to how we added boundaries to the paddle, I added boundaries to the left, right and top bounds of the screen to stop the ball from disappearing off and instead make it bounce in back to the screen. This is depicted in the below code snippet where I define Bezier curves whose boundaries will enclose the ball in the screen to the top left and right but will leave the bottom unbounded. I will discuss what happens when the ball exits the gameView when on the topic of endgame circumstances.

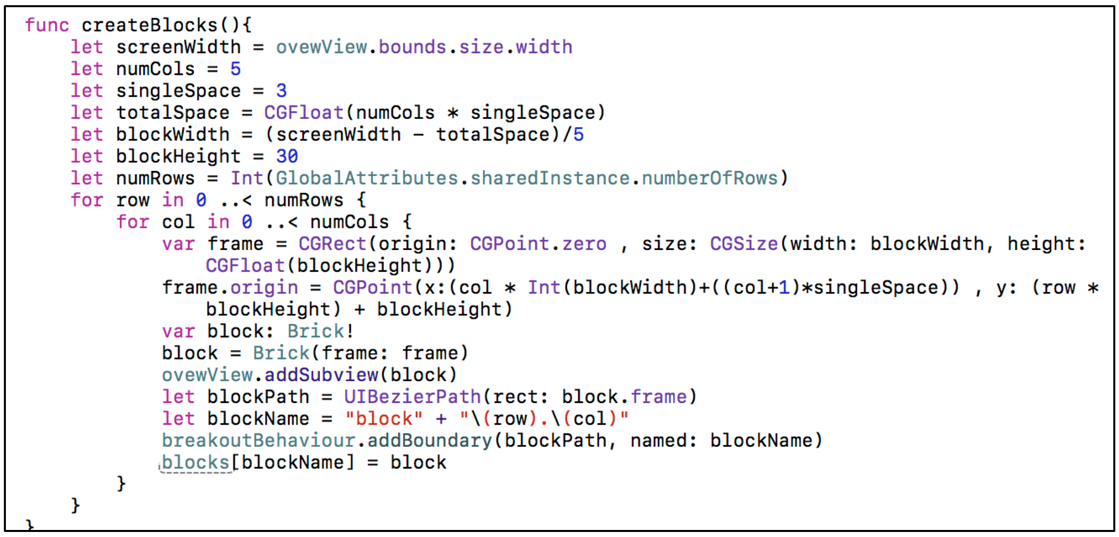


```
func addScreenBounds(){
    //set leftWall bounds
    let leftWall = UIBezierPath()
    leftWall.move(to: CGPoint(x: ovewView.bounds.origin.x, y: ovewView.bounds.size.height))
    leftWall.addLine(to: CGPoint(x:ovewView.bounds.origin.x, y: ovewView.bounds.origin.y))
    //set ceiling bounds
    let ceiling = UIBezierPath()
    ceiling.move(to: CGPoint(x: ovewView.bounds.origin.x, y: ovewView.bounds.origin.y))
    ceiling.addLine(to: CGPoint(x: ovewView.bounds.size.width, y: ovewView.bounds.origin.y))
    //set rightWall bounds
    let rightWall = UIBezierPath()
    rightWall.move(to: CGPoint(x: ovewView.bounds.size.width, y: ovewView.bounds.origin.y))
    rightWall.addLine(to: CGPoint(x: ovewView.bounds.size.width, y: ovewView.bounds.size.height))

    breakoutBehaviour.addBoundary(leftWall, named: "left")
    breakoutBehaviour.addBoundary(ceiling, named: "top")
    breakoutBehaviour.addBoundary(rightWall, named: "right")
}
```

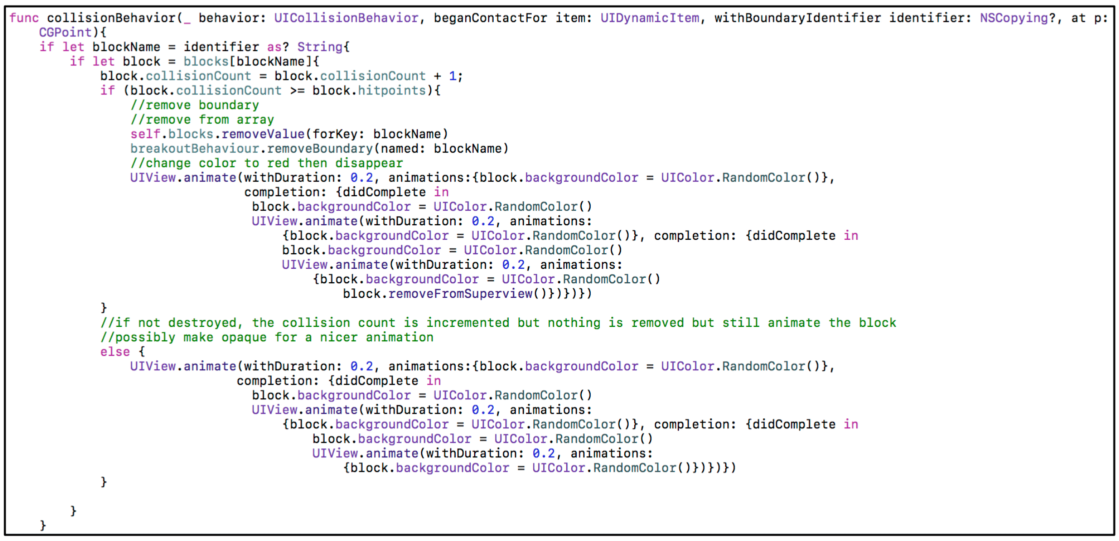
Step 3: The Blocks

Now that the ball and the paddle are interacting properly and the ball is not constantly flying off-screen thanks to our gameView boundaries, it is time to create the blocks. The first step was to create a Brick class, again this will be a UIView and will have an init() method for drawing the rectangle. Drawing the bricks require considerations to be made. First of all the width of the screen must be taken into consideration, the number of columns of bricks and the spacing between the bricks. I set the number of columns to 5 and the user defines in the settings tab how many of rows of bricks they would like. Once we know the number of rows and columns we can cycle through and draw the required number of bricks. This is done by setting the frame width, height and origin and passing the frame into the *Brick.swift* class. The brick is added to the gameView and finally boundaries are added. The name associated with the boundary is a culmination of the column and row number, this is need later to determine precise collision mechanics. This process is documented in the following code snippet:



```
func createBlocks(){
    let screenWidth = ovewView.bounds.size.width
    let numCols = 5
    let singleSpace = 3
    let totalSpace = CGFloat(numCols * singleSpace)
    let blockWidth = (screenWidth - totalSpace)/5
    let blockHeight = 30
    let numRows = Int(GlobalAttributes.sharedInstance.numberOfRows)
    for row in 0 ..< numRows {
        for col in 0 ..< numCols {
            var frame = CGRect(origin: CGPoint.zero , size: CGSize(width: blockWidth, height:
                CGFloat(blockHeight)))
            frame.origin = CGPoint(x:(col * Int(blockWidth)+((col+1)*singleSpace)) , y: (row *
                blockHeight) + blockHeight)
            var block: Brick!
            block = Brick(frame: frame)
            ovewView.addSubview(block)
            let blockPath = UIBezierPath(rect: block.frame)
            let blockName = "block" + "\(row).\(col)"
            breakoutBehaviour.addBoundary(blockPath, named: blockName)
            blocks[blockName] = block
        }
    }
}
```

The next aspect of the game is determining which brick the ball has hit when a collision is detected. This is done using the collisionBehaviour method it takes the object with which the ball has collided and uses it's identifier to determine whether or not it is contained in our array of blocks. If it is in our array then it is a brick, if it is not in our array then it is either the paddle or a wall boundary. Then I check how many hits are allowed on a block (configurable in settings) the default is one but raising this number will increase the high score. If a brick is hit and it is not destroyed it becomes animated similar to the destruction of a brick except the block is not removed from the superview nor is it removed from the array of blocks this is done using *UIView.animate*. This is illustrated in the below code snippet:



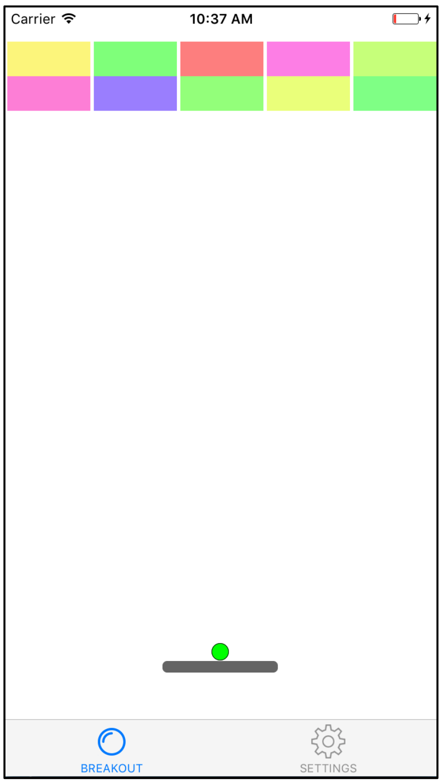
```swift
func collisionBehavior(_ behavior: UICollisionBehavior, beganContactFor item: UIDynamicItem, withBoundaryIdentifier identifier: NSCopying?, at p:
    CGPoint){
    if let blockName = identifier as? String{
        if let block = blocks[blockName]{
            block.collisionCount = block.collisionCount + 1;
            if (block.collisionCount >= block.hitpoints){
                //remove boundary
                //remove from array
                self.blocks.removeValue(forKey: blockName)
                breakoutBehaviour.removeBoundary(named: blockName)
                //change color to red then disappear
                UIView.animate(withDuration: 0.2, animations:{block.backgroundColor = UIColor.RandomColor()},
                               completion: {didComplete in
                                block.backgroundColor = UIColor.RandomColor()
                                UIView.animate(withDuration: 0.2, animations:
                                    {block.backgroundColor = UIColor.RandomColor()}, completion: {didComplete in
                                    block.backgroundColor = UIColor.RandomColor()
                                    UIView.animate(withDuration: 0.2, animations:
                                        {block.backgroundColor = UIColor.RandomColor()
                                            block.removeFromSuperview()})})})
            }
            //if not destroyed, the collision count is incremented but nothing is removed but still animate the block
            //possibly make opaque for a nicer animation
            else {
                UIView.animate(withDuration: 0.2, animations:{block.backgroundColor = UIColor.RandomColor()},
                              completion: {didComplete in
                                block.backgroundColor = UIColor.RandomColor()
                                UIView.animate(withDuration: 0.2, animations:
                                    {block.backgroundColor = UIColor.RandomColor()}, completion: {didComplete in
                                        block.backgroundColor = UIColor.RandomColor()
                                        UIView.animate(withDuration: 0.2, animations:
                                            {block.backgroundColor = UIColor.RandomColor()})})})
            }

        }
    }
```

Like in the GravityBubbles example I extended the UIColor class to facilitate random coloring, this will be especially useful when animating block destruction.

Step 4: Starting the game

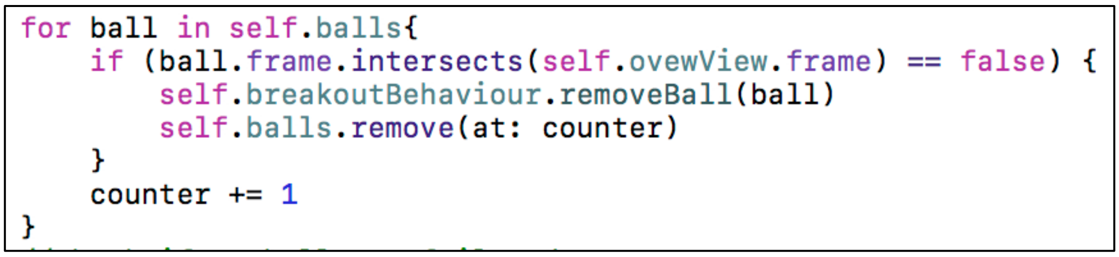
When the app is opened the user sees a Label saying ‘click to play’ which when clicked creates the ball, paddle and blocks as shown below:



Nothing moves yet but the game has started, the number of blocks, colour of the ball and elasticity of the ball are all configurable in the settings tab. The next time the user click the ball move upwards through the air at a random angle between 70 and 110 degrees. This is done using the UIPushBehaviour. So the first click starts the game, the second launches the ball and each subsequent click is used to change the direction of the ball, this helps keep the game interesting and means that the ball can always be set back in motion should it come to a halt for some reason (e.g. low elasticity). Finding the movement angle and reversing it serves to accelerate the ball back into the air, to keep some randomness in the game ± 0.1 radians are added. Linear velocity and UIPushbehaviour.angle were used to set the speed and angle of the ball after the user clicks the screen.

Step 5: The End Game

There are 2 ways the game can end. The first is that the ball drops out of the *gameView*. Using unowned self I check that the ball is contained intersects the gameView. If it does not intersect the *gameView* then the game is over and the player has lost:

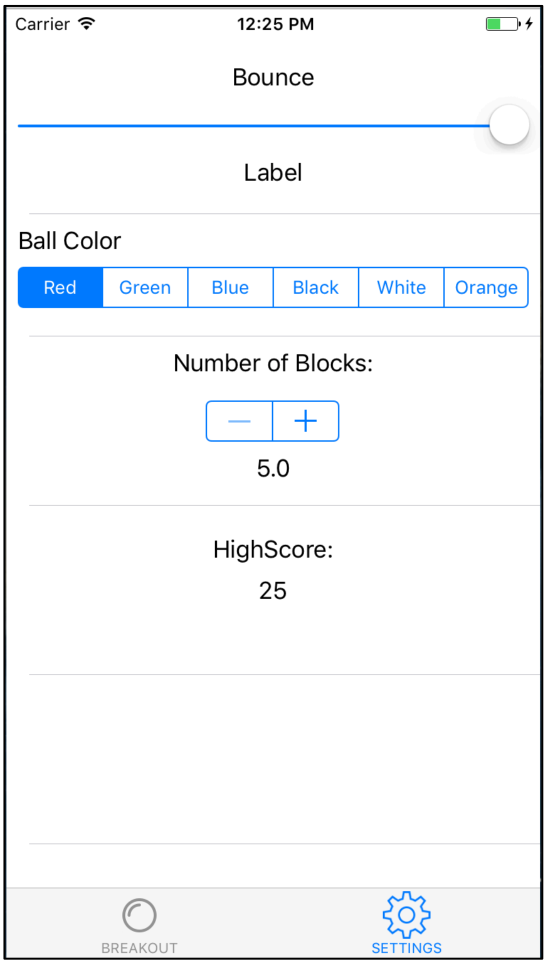


```
for ball in self.balls{
    if (ball.frame.intersects(self.ovewView.frame) == false) {
        self.breakoutBehaviour.removeBall(ball)
        self.balls.remove(at: counter)
    }
    counter += 1
}
```

If this is not the case then I check the block array to see if the user has successfully destroyed all the bricks. If they have managed this feat then the game is over and they have been successful. The next check is to see if they have beaten their previous high score – the high score is a persistent global variable and involves a simple inequality check. Either way once the game is ended, the paddle and ball are removed and the endgame *UIView* is displayed with a *textLabel* telling the user how they did. The *gameClick* variable is returned to 0 and the user can begin the game again as described in step 4.

Step 6: Settings

The settings tab was the second tab in the game as shown in the below image:


Carrier 12:25 PM

Bounce

Label

Ball Color

Red | Green | Blue | Black | White | Orange

Number of Blocks:

5.0

HighScore:

25

All of the functions in the settingsViewController rely on global variables. I created a *GlobalAttributes.swift* class that is responsible for defining these attributes. The attributes I chose to control were ball elasticity, ball colour and the number of bricks. To demonstrate my knowledge I decided to use 3 different types of user input controls: a slider, a segmented control and a stepper. To control the high score display I used a label. Once I connected the user controls to the SettingsVC I simply replaced the definitions in the other areas of the application with declarations like:

```
behaviour.elasticity = GlobalAttributes.sharedInstance.bounce
```

In keeping with the assignment description I implemented 3 user controls but the exact same method can be used to control all aspects of the application. The important thing is that in the gameStart() method found in the game ViewController the values are reset, this means that user changes are instantly applied when changed in the settings tab.