Assignment 1

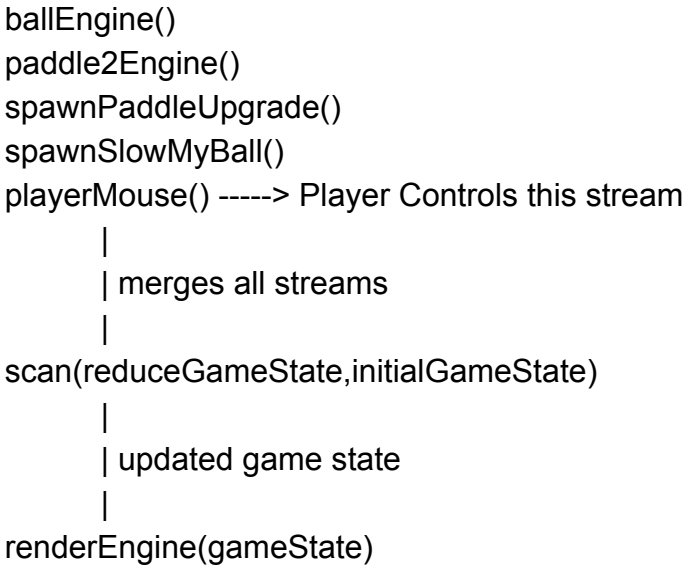
Pong Game Report

Name: Blaise Tiong Zhao Xiong
ID: 31112005

Architecture:

The architecture used in this project is Model View Controller Model. The user will manipulate the program's state data using the mouse and the view will be updated based on the changes. I chose this a i find it is the most easy way and structured to code a reactive game like this

I have five streams of observables, they will be merged and piped through scan. The changes from the stream are intercepted to return an updated 'GameState'. The game state will be updated, and rendered by my renderEngine. The whole system is tightly controlled with no side effects apart from the renderEngine. So to recap, the scan function takes in 'changes' from different streams of data, applies the changes, to produce a new game state to replace the old one before being rendered on screen via the render engine.



I have followed the FRP style by using the concept of streams and how to manipulate them

GameState:

The game state is the heart of the game. We have all the data for the game here including the scores as well as ‘uiStates’. ‘uiStates’ is the data where the renderEngine will need to make ui elements animate in the game. And also this is how the state is managed with in the game

```
interface GameState {
 playerScore: number,
 computerScore: number,
 uiStates: {
   ballState: BallState,
   paddle1State: PaddleState,
   paddle2State: PaddleState,
   scoreBoardState: ScoreBoardState,
   paddleSizeIncreaseState: UpgradeStates,
   slowMyBallState: UpgradeStates,
 },
}
```

First of all, I have ‘ballState’ which holds the center coordinates of the ball on the x-axis and y-axis, the velocity of the ball traveling along the x-axis and y-axis. The radius of the ball, the fill and the svg type will remain the same throughout the whole game

```
interface BallState { svgType: String, cx: number, cy: number, r: number, fill: String, vx: number, vy: number }
```

Next, I will have the paddle state which contains the x and y coordinates of the rectangle, the fill, the width and height of the paddle. Only y and the height is subject to updates in paddleState.

```
interface PaddleState { svgType: String, x: number, y: number, width: number, height: number, fill: String }
```

ScoreboardState will just hold the position of the scoreboard on the svg canvas

```
interface ScoreBoardState { svgType: String, x: number, y: number, 'font-size': String, fill: String }
```

Finally we have the UpgradeStates, the state will hold the coordinates of the updates spots on the svg canvas. Initially when the game starts, the will be at the coordinate (0,0) with their width and height set to 0.

```
interface UpgradeStates { svgType: String, x: number, y: number, fill: String,
width: number, height: number }
```

From my code I have a function called initialGameState() which will return when calling my initial game state. We will be using this gameState and return a new one in the program.

Direction System:

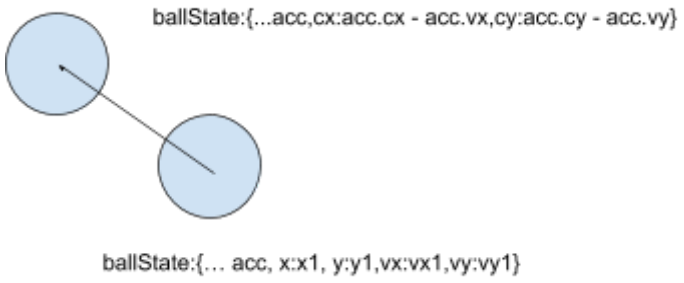
vy: Going up is negative down is positive
vx: Going left is negative right is positive

The Ball:

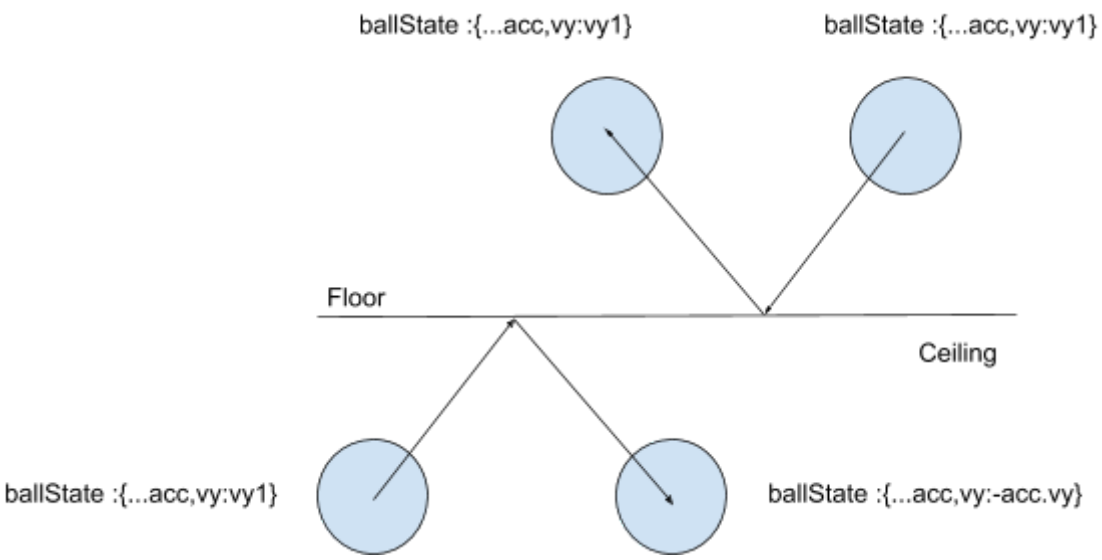
1. **Traveling in straight line**
   In order to make the ball move, we have to make it add the velocity*time to its existing location on both x and y axes. Since the ball will be updated every interval we just need to minus the velocity on x to the ball's x coordinate and velocity on y on the y coordinate so it will be in the next position in 1 interval.

   We take a ball state 'acc' and we will return

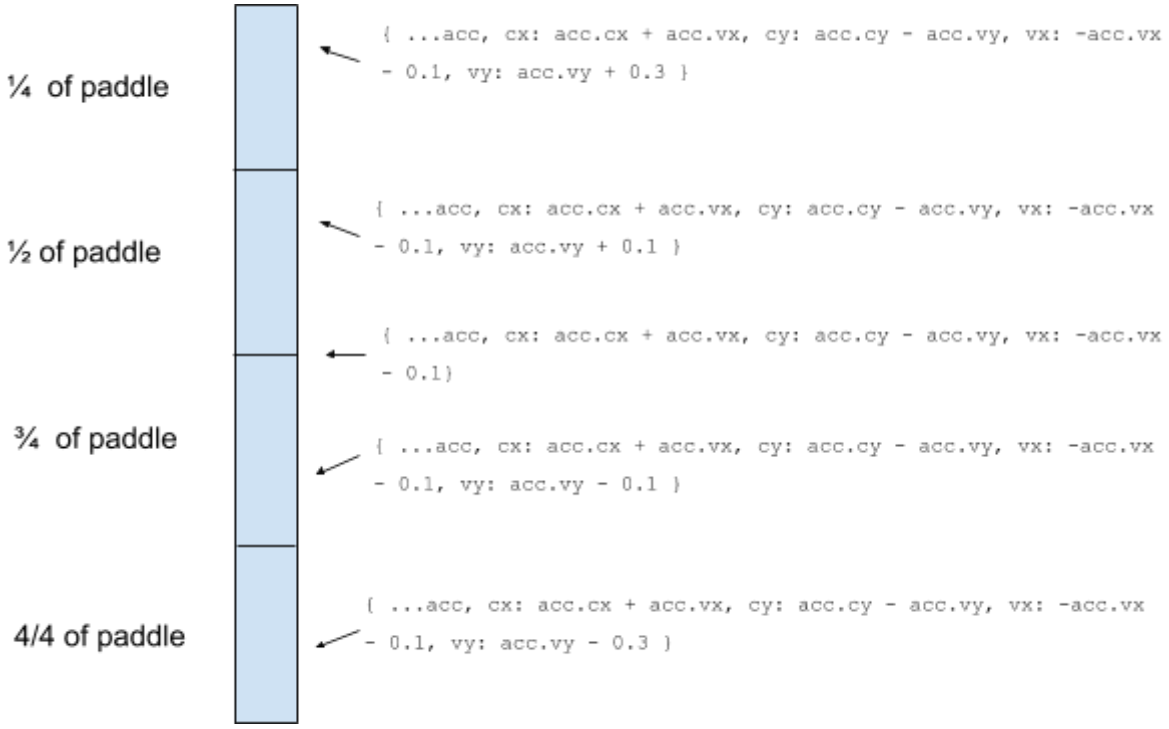
   ```
   { ...acc, cx: acc.cx - acc.vx, cy: acc.cy - acc.vy }
   ```

2. **Hitting the ceiling or floor**
   According to the laws of physics when a travelling object collides with something the velocity on the axis of collision will be negated. This goes the same with our ball if the ball hits the ceiling or floor. The function that is in charge of this is onYCollision1 where it takes in the initial ball state, and returns a ballState with a negated vy



3. **onPaddleCollision**



My paddle has been divided into 4 parts. When the ball collides with ¼ part, a new ball state with a negated vx and a vy with an increase of 0.3 is produced. This ensures the new ball will travel in the opposite direction and upwards. If the ball collides with the ½ part it will travel in the opposite direction and a slightly upwards since the vy is updated with a minimum increase of 0.1. The same goes to ¾ paddle and 4/4 just that the ball will deflect downwards. If the ball is hit directly in the middle the new ball state produced will have a negated vx only.

The function which handles all these cases is the function onTravelX1() it takes in a ballState and a paddleState and resolve which part of the paddle the ball hits and returns a new ballState

4. **Someone scores**
   When someone scores(when the center of the ball < 0 or ball > 700), the gameState will be taken in and has the value updated in the returned new gameState. This is done in the function someoneScores(). If either the computer or the player has won (getting a score of 7), an alert will pop up displaying the winner and returns initialGameState() so that a new game will be run.

5. **Upgrades**
   The upgrades are shown as red squares and blue squares on the canvas, if the cx and cy of the ball happens to be within the the square, an updated game state will be returned by the function onBallHitsPaddleUpgrade() and onBallHItsSlowDown(). The former will return the gameState with the paddle elongated while the later returns a ball will a vx of 1.

   Finally the function updateBallPos() will resolve the conditions of the balls and returns a corresponding updatedGameState()

```
function updateBallPos(gameState){
        If (the ball is in red square)
                return onBallHitsPaddleUpgrade(gameState)
        If (the ball is in blue square)
                return onBallHItsSlowDown(gameState)
        If (the ball is beyond the left edge or right edge)
                return someoneWins
        If (the hits the ceiling of the floor)
        return {...gameState,uiStates:{gameState.uiStates,ballState:onYCollision}}
        If nothing else
        return {...gameState,uiStates:{gameState.uiStates,ballState:onTravelY}}
}
```

Player's paddle

To update the player's paddle I built a function updatePaddlePos1 which takes in a gameState and number (player's mouse) and returns a new gameState with the y coordinate of the paddle same as the player's mouse on the Y coordinates.

Computer's paddle

The computer's paddle is made to follow the ball in the function computerAI which takes a gameState and analyses if the top of paddle is below the ball and it will reduce the paddle's y coordinate in the gameState which we will be returning. Else if the ball is below the paddle , y coordinates of the paddle is made to increase in the new state that will be returned.

Upgrades
PaddleUpgradeSpots and SlowBallSpots are functions which will return a new gameState with the updated positions of the blue squares and red squares

Reducing the streams into gameStates

```
function reduceGameState(acc: GameState, changes: Paddle1YChanges | Paddle2YChanges
| RunBall | PaddleUpgrade | SlowMyBall): GameState {
 if (changes instanceof Paddle1YChanges)
   return updatePaddle1Pos(acc, changes.newY);
 if (changes instanceof Paddle2YChanges)
   return computerAI(acc);
 if (changes instanceof RunBall)
   return updateBallPos(acc);
 if (changes instanceof PaddleUpgrade)
   return paddleUpgradeSpots(acc, changes.x, changes.y)
 if (changes instanceof SlowMyBall)
   return slowBallSpots(acc, changes.x, changes.y)
}
```

This function is used in scan. it runs the corresponding codes based on the instance the changes are of. As mentioned earlier I have 5 observables and each of their streams will be passed into reduceGameState and reduceGameState will call corresponding functions to the type of stream to return the corresponding gameStates. For example, playerMouse will return an Observable that produces a stream of Paddle1YChanges that holds the y coordinate of the player's mouse on the canvas and it will be passed into reduceGameState where updatePaddle1Pos will be ran and return a corresponding new Game State, same goes with the others.

```
function ballEngine(): Observable<RunBall> {
 return interval(1).pipe(map((_) => new RunBall()))
```

```
}
function paddle2Engine(): Observable<Paddle2YChanges> {
 return interval(1).pipe(map((_) => new Paddle2YChanges()))
}
function spawnPaddleUpgrade(): Observable<PaddleUpgrade> {
 return interval(30000).pipe(map((_) => new PaddleUpgrade(300 + Math.random() *
100, Math.random() * 500)))
}
function spawnSlowMyBall(): Observable<SlowMyBall> {
 return interval(32000).pipe(map((_) => new SlowMyBall(300 + Math.random() * 100,
Math.random() * 500)))
}
function playerMouse(): Observable<Paddle1YChanges> {
 return fromEvent<MouseEvent>(document.getElementById('canvas'), 'mousemove')
   .pipe(map(({ clientY }) => new Paddle1YChanges(clientY - 105)))
}Finally the renderEngine(),
function renderEngine(gs: GameState, canvas: String): Array<Element> {
 /* This function renders all UI component states, the canvas will be cleared
before rendering. */
 document.getElementById('canvas').innerHTML = '';
 return Object.entries(gs.uiStates).map(
   ([key, UIAttr]) => {
     return Object.entries(UIAttr)
       .reduce(
         (acc: Element, [key, value], index: number) => {
           acc.setAttribute(key, String(value));
           acc.innerHTML = String(gs.playerScore + ':' + gs.computerScore)
           return acc
         }, document.createElementNS(document.getElementById(String(canvas))
           .namespaceURI, String(UIAttr.svgType))
       )
   }
 )
}
```

This function clears the canvas before iterating through the the uiStates and renders the UI elements on the canvas.