Cedric Liang (29674662) FIT2102 2020 Assignment 1 Report

This is my implementation of Pong using RxJS Observables. I'm 90% sure that my implementation is written in a very purely functional style, with minimal side effects.

Workings of the Code

The implementation draws heavy inspiration from Tim Dwyer's implementation of FRP Asteroids. Like FRP Asteroids, FRP Pong is fundamentally driven by interval-based ticks, with interspersed events which correspond to user interaction with the system.

```
interval(10)
  .pipe(
    map(elapsed=>new Tick(elapsed)),
    merge(
      startDown, startUp, stopDown, stopUp, restartUp),
    scan(reduceState, initialGameState)
  ).subscribe(updateView);
}
```

Furthermore, FRP similarly uses a readonly container called *State* that stores parameters corresponding to the current gamestate; a set of unique parameters uniquely describe a gamestate and a gamestate can be recreated with a known *State*. Here is an example *State*:

```
const initialGameState: State = {ballxvel: -1.5,
                                 ballyvel: (Math.random()*5)-2.5,
                                 ballx: 298,
                                 bally: 298,
                                 playerpaddley: 280,
                                 playerpaddleyvel: 0,
                                 aipaddley: 280,
                                 playerscore: 0,
                                 aiscore: 0,
                                 gamerunning: 1}
```

Every event in the main observable (henceforth *gamestream*) is an instruction that tells the game how to transition to the next gamestate, provided the previous gamestate is provided as a reference. In the purely functional spirit, the next gamestate is actually a newly constructed gamestate, which is then fed into and reduced in the proverbial woodchipper that is the gamestream's scan operator.

A *Tick* event evolves the system in time, such as moving objects based on their velocities in the previous state and checks for various collisions that trigger changes in gamestate.

A *movePlayerPaddle* event is created when a user presses the up and down arrow keys. This event creates a new gamestate with the corresponding alteration in the velocity of the player's paddle. Note that this doesn't directly move the paddle in the newly-created gamestate; the movement of the paddle must wait until the next *Tick*, where the velocity is applied to the displacement of the paddle.

Finally, a *restartGame* event is created when the user presses the Enter key. Note that this does not always restart the game! In the reduceState function upon reading a restartGame event, the previous state is checked to see if the game is currently running. Only if the game has stopped does the next state it creates correspond to an 'initial' gamestate; otherwise, it simply returns the current gamestate: no change! The effect is that the Enter key only restarts the game if it has ended already.

Speaking of *reduceState*, this contains the bulk of the logic of the game. Since reduceState is fed into scan in the gamestream, it is crucial to the correct running of the game. Technically speaking, the events themselves don't create the new states, they are just tokens or indicators that tell the reduceState function what new State to create. The following code is reduceStream creating new States based on the event and the previous state.

```
//this is where the magic happens!
const reduceState = (s:State, e:movePlayerPaddle|Tick|restartGame)=>
  e instanceof movePlayerPaddle ? {...s,
    playerpaddleyvel: e.translation}:
  //only permits restarting the game if the game is not currently running
  e instanceof restartGame ? ((s.gamerunning<1)? initialGameState : s):

  //EVERYTHING BELOW IS IN THE EVENT OF A TICK
  (s.playerscore === 7)?
  //playerwin
    {
      ballxvel: 0,
      ballyvel: 0,
      ballx: -298,
      bally: 298,
      playerpaddley: 280,
      playerpaddleyvel: 0,
      aipaddley: 280,
      playerscore:s.playerscore,
      aiscore:s.aiscore,
      gamerunning:0
    }:
```

These are all the components that are required to keep the game chugging along from state to state. I've defined some other functions such as *collisionBallX* or *shiftaipaddley*, but these are just simple pure functions that are reused code, and as such aren't so much a part of the *mechanism* of the game. They could be put into reduceState, but I've extracted some of them to reduce clutter.

All of this so far has been completely pure and devoid of side effects, but alas, it's not enough, because side effects are required to interact with the program. Indeed, we need to see what's going on in the game. We can do this by subscribing to the gamestream and feeding the output into an *updateView* function. This function quite simply edits the HTML document based on the latest gamestate that is in the scan function's accumulator and ensures that what we're seeing in our browser is the latest and most accurate representation of the gamestate. Luckily, performing all of these tasks within one function allows us to 'quarantine' and sequester all the code that creates side effects to this one section.

Design Decisions

The main advantage of this state-based design is that it can remain purely functional. Using scan and a function like reduceState allows new states to be created based on older states without any mutability: an older state is discarded and not used again. Of course, this has been contingent on the fact that Pong is a relatively simple game, and that the State can be contained within one container without becoming too complicated.

One design decision I made was to make the gamestream interval based rather than user-interaction based. This I decided to do because I was getting to the point where I had many moving parts (both paddles and balls). I need the ball to keep moving even when the user is not mot moving their paddle, and I also need any such movement to *stay consistent.* A velocity of 4 needs to visually appear consistent, and there cannot be any lag or stuttering in the game.

I had earlier implemented the game with a user-input driven gamestream and as a result, the movement of the ball and the paddle was uneven, fast at times, laggy during other times. An

interval-based stream acts as the backbone of the game and keeps things working all the time every time.

Using an interval-based stream also has another advantage, in that it allows me to represent the velocity components of the ball as simply a number. Since a change in position is simply velocity multiplied by some interval of time, I can easily obtain the change in position of the ball from one tick to another by adding some multiple or fraction of the velocity at every tick. This simplifies the gamestate greatly: I have my x and y coordinates for the ball, and the x and y velocities, and that's it.

FRP Adherence and State Management

I think I successfully followed the FRP style. I've only used observables, which ensures I am programming reactively, and I have explained in the previous paragraphs how I have kept my side effects minimal. I've also adequately explained my state management system.