FIT2102 Assignment 1 Report

Name: Isaac Lee Kian Min

Student ID: 31167462

Summarize the workings of the code and highlight the interesting parts.

To put it simply, my code seeks to maintain and update the game state by calling other functions to perform the required computations. It consists a main merged **observable** to observe for certain events while calling pure (and impure) functions to **update** (reduce) the game state based on these observed events. A few global **constants** are defined to keep track of the initial values of certain attributes e.g. the canvas width and height.

An interesting part of my code is the changeVelocity function. The changeVelocity function works by calculating the **reflective vector** of a ball upon collision with a paddle. The main idea is to get a proportion and determine the x and y components of the reflected vector through the multiplication of this proportion with a maximum angle of reflection, along with the resultant components of the incident vector. (Note: I drew inspiration/idea through the website: https://gamedev.stackexchange.com/questions/4253/in-pong-how-do-you-calculate-the-balls-direction-when-it-bounces-off-the-paddl/4255#4255)

Give a high level overview of your design decisions and justification.

Firstly, I decided to have 4 observables: keyDown$, keyUp$, ballMove$ and leftPaddle$ to help trigger the movements of the paddles and ball. After which, we **merge** these streams using the merge creation operator to funnel all these observables into one single observable, which makes it easier to **maintain** the game state as we only need to subscribe to one observable rather than maintain four separate observable streams.

On a similar note, I chose to control the ball and left paddle movement with its own individual observables to provide some degree of **smoothness** and time flow to the gameplay as each fires an event within a fixed interval, causing some sort of movement in response.

Next, I created a few curried styled functions such as the binaryOperation function to allow **reusability** of it for other purposes. This higher order function takes in an operation function and invokes it only after receiving its arguments through the currying process.

Having said that, the use of **recursion** may not have been the most elegant of ways to restart the game given that it might cause a stackoverflow issue. But this issue would not be much of a problem as the functions composed in this program are relatively **pure** and that there are no hidden forms of mutation, making them transparent.

Explain how you tried to follow the FRP style.

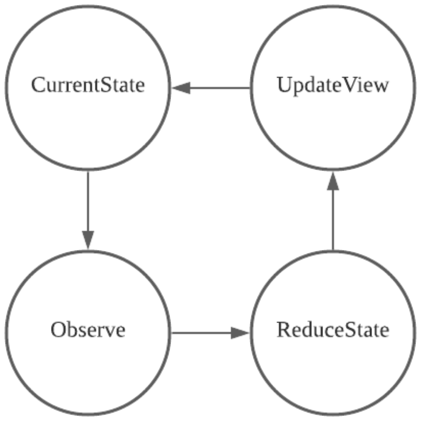
Firstly, I tried to follow the FRP style of programming through the use of **Observables**, in particular Observable Streams. As mentioned in the previous section, merging all observables into one centralized observable provided me the options of using **operators** such as map and scan to transform the observed events into some useful action or information about state changes.

Next, I tried to limit the impact of **side-effects** by creating the updateView and determineWinner functions since it is required to update the attributes of the HTML elements to produce a visual representation of the game. This allows me to better **manage** mutable data by encapsulating it within a function rather than making it global data.

Besides that, all variables were declared as **constants** to prevent the values of these variables from mutating, avoiding any nasty side-effects as a result of accidental mutations. Doing so also allowed me to define **referentially transparent** functions, such as the invertY function where we could replace such function calls with its evaluated value but still reproduce the same result for the same inputs (i.e. there are no side-effects occurring within function scope).

Finally, I decided to make a few generic functions such as the keyObservable and overlappingIntervals functions. Making these functions **generic** allows me to take advantage of **parametric polymorphism**, making it **reusable** not only in terms of avoiding duplicated code but also reusable on other data types.

How you manage state throughout your game.



(Note: This state management system is heavily influenced by Tim's Observable Asteroids example: https://tgdwyer.github.io/asteroids/)

I chose to create data types Ball and Paddle to help keep track of the individual states of both the ball and paddles respectively for every cycle while also a GameState data type that allows me to keep track of the whole entire game state. These types are also declared to be Readonly to prevent any forms of **mutability** of internal states. In the event one of these data types requires mutation (updating), a new copy of the original type is created instead with the updated values.

The process in which I managed the game states can be summarized into 4 distinct sections. Firstly, we are presented with the **current state**, which holds the states of all objects in the game (e.g. paddles, balls, scores). Next, we **observe** from our observable streams for observable events that acts as a signal for an update to the current state, e.g. when we press down a specific key, we observe that pressing this key indicates a change of a paddle's position on the canvas. After this, we **reduce** the current state based on the observable events being pushed, e.g. updating the paddle's position. During each reduce stage, a new updated copy of the state is created to prevent any side-effects. Finally, we **update** the view of the canvas, transforming our states into a graphical representable interface through the web.