Pong with Functional Reactive Programming - Assignment 1 Report

Liow Gian Hao || 30666910 || glio0001@student.monash.edu

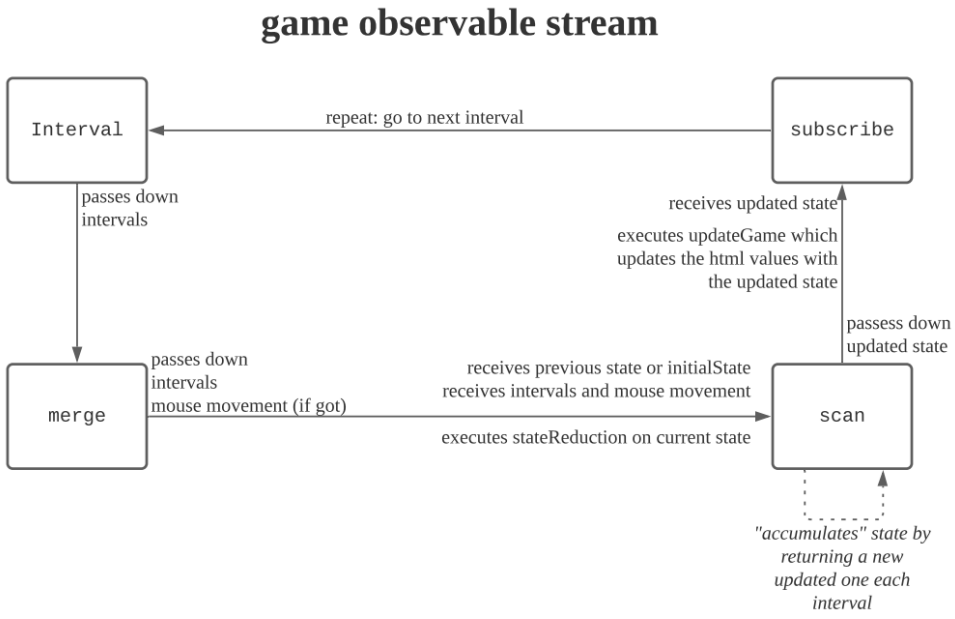
The pong game is done with functional reactive programming through an implementation of Observables from the rxjs library. It is done through observables to ease the asynchronous behavior and to contain the transformation of objects in the game into one unified structure.

About purity

All functions in the assignment are pure except for **updateGame**, which executes in subscription. The pure functions will only deal with a state, and does not access any global values, perform any side effect or randomness. Per the definition of pure functions, the same parameters for the function will always return the same results, and nothing outside of its scope is affected.

The Observable Stream

The **game** (line 225) is the observable stream which asynchronously updates the game. The interval uses pipe to transform elements from one container to another. The stream will merge itself with **mouse$**, which is a mouse event that returns a **Move** object. The **Move** object exists to differentiate between the different values that is being streamed in our later functions. The stream will then use scan. Scan is an accumulator function which takes **initialState**, and executes **stateReduction** which computes changes in the state and returns a deep copy of the previous state where certain values differ depending on the logic of **stateReduction**. No mutability occurs here, when scan "accumulates" our state, it always returns a brand-new state for the next interval. A visualization of the stream is shown below.



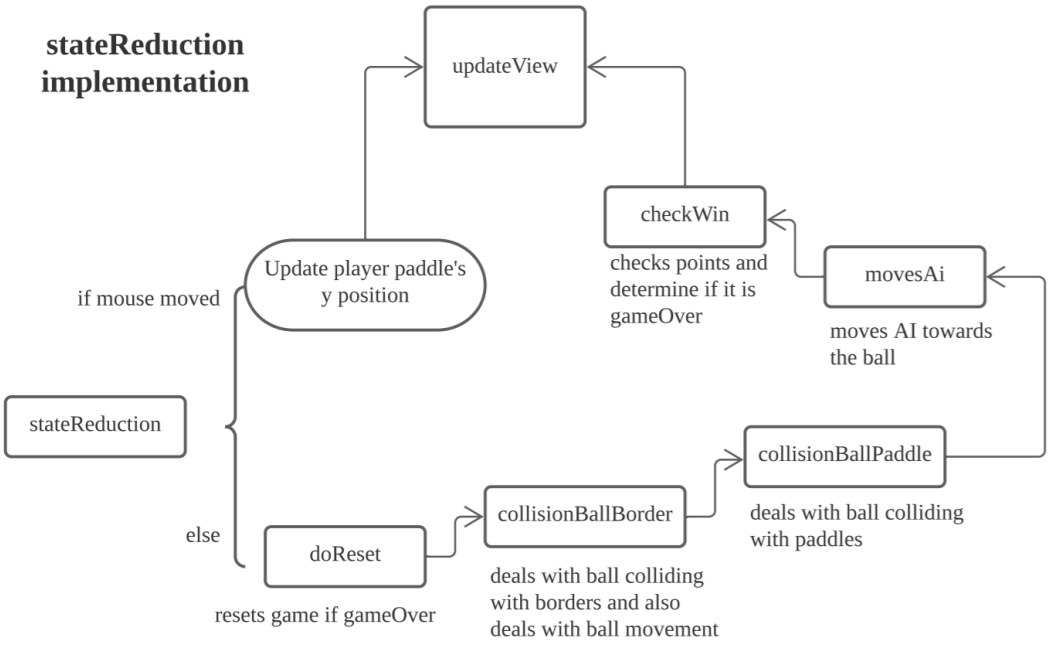
stateReduction and updateGame

Calling **stateReduction** with scan is how we manipulate the values of objects in the state. Function **stateReduction** will receive parameters of (s, e), where s is the previous state and e will return a Move object only if mouse event has been detected. If e is an instance of **Move**, we will return a new copy of the state where changes are made on the y-position of the player paddle. Else, it executes a line of nested functions which performs non-player-controlled actions.

For non-player-controlled actions, **stateReduction** calls **doReset, collisionBallBorder, collisionBallPaddle, aiMovement, checkWin** in that specific order. Every function takes a state and returns a copy of the state with modified values based on what the specific function is meant to do. As we compute new values for each attribute, nothing is mutated, and no side effects were triggered. This allows our functions to remain pure, but still be able to give new values to attributes.

Once scan is done, an updated state will be passed to our subscription, which will then call **updateGame**. The function **updateGame** will take in the updated state and modify the respective html elements. Containing all mutable actions within the subscription will prevent any side effects happening in the logic of the game.

A visualization of the process in **stateReduction** is shown below.



Game Design Notes

Movement is by mouse, no click or drag required, cursor should be in the canvas.
Whoever scores gets the ball moving in their direction except for the start of the game, the ball moves right always.
The wall rebound will flip the y-direction, just like how real-world physics would.
The AI will detect the y position of the ball and move towards it based on its velocity.
The AI only starts detecting once the ball is across the mid-point of the canvas.
The ball will travel faster and direction would change depending on the area of the paddle it collides with, this visualized below.