FIT 2102 Assignment 1 Report

I've edited the HTML file to display the instructions for the game, and the CSS file to add some styles for word display.

Set up types, elements, a Direction array, Vec, RNG & torusWrap *[line 5-103]*

For the typescript file, the code starts with getting the canvas element's ID from HTML, and setting necessary types**: Key**, **Event**, and **Direction** (the 'e' in some directions (eg eNW, eNS) are used to differentiate directions after hitting the edge of a paddle, with just normal directions(eg NW, NS), as the y-velocity, ie speed of the ball should increase when hitting a paddle's edge. An array of Directions is also created. Following, are the Vec class, torusWrap function and RNG class referenced from sources as cited. Respectively, they are used to create positions, to ensure flexibility of player paddle movement (paddle that exceeds canvas would be wrapped and brought in from the other direction), and as a simple, seedable, pseudo-random number generator. I've decided to use RNG as a random number generator, instead of using Math.random(), to maintain the purity of the code. This is because Math.random() is an impure function that creates side effects, while as we'll see later, RNG would be adapted to an immutable object, with a seed that is accumulated in 'scan' in 'subscription', and generate random numbers as well. After that, we create the player and opponent paddle and score, ball and divider, and append them to the canvas.

keyObservable observable streams *[line 104-122]*

Moving on, we create a function **keyObservable** of generic type, which takes in an Event, a Key and a result that returns a generic type, as inputs; and returns an observable stream of keyboard events, which would be transformed into the generic type result in the end. This function displays parametric polymorphism, as it could be reused with different types of 'result' parameter passed in. Then, we create four separate observable streams, each with different Events and Key Codes, and each being a stream of a number which would be used to change the y-position of the player paddle. Note in this case, the generic type is passed in a number type, where if later we want to create a new observable stream of strings, the generic type would just be passed in a string type, which is completely fine.

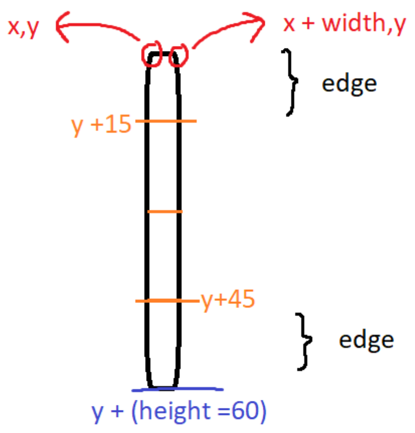
Types Body, State & initialState *[line 123-174]*

Next, we create two read-only types: **Body**, for ball and paddles; and **State**, containing the information of the game. We also create functions for each Body, that contains the Body's most starting position. Then we have an **initialState** for the start of a game, where each object is at an initial value, and we set the first ball direction to be NE. The 'seed' also starts from one, which would be accumulated and rng would be a new RNG with larger and larger seed.

newState function and its closure *[line 175-245]*

The **newState** function takes in parameters: the current state, a number returned from the previous observable stream, and a new posNum used to adjust the ball's position. It would return a new state, where the state in the parameter is not modified at all, only its value is being used, thus again being pure and having no side effects. Inside newState, let's divide the created functions into three parts:

1. **The 1st part** contains **functions that check if the ball hits any paddle/wall**, by checking the ball's position against the paddle/wall's position. For example, ballHitsPad1 checks if the ball's x-position is within the paddle's x-position + width of 10px, and the ball's y-position if it is within the paddle's y-position + height of 60px (The x,y position of paddle is on the top left, hence its top right x-position is x+width, bottom right y-position is y+height).
   Edge checks if the ball hits the paddle's edge. We split the paddle of 60px into 4 parts, so starting from the paddle's y-position, 0-15px and 45-60px are considered 'edges' of the paddle. *(see illustration of a paddle below:)*



2. **The 2nd part** contains **findDirection** function. It uses the current state's ball direction, along with any result above ie if the ball hits a paddle/ a wall etc. to produce a next suitable direction for the ball to move to. If the ball hits the player paddle, it will further check if it hits the edge, and if so, the direction would have an extra 'e' in front.
3. **The 3rd part** would be **findNextPosition,** which uses the new direction found from (2), and returns a new ball position with 'value' changes to the current state's ball position. For edge directions, we add an extra 0.5 to value so that the ball would move an additional 0.5px in its x and y position, which means faster ball.

Finally, this newState returns a newly created state with updated information on the Body and game state. Some notes for this:

i. Computer paddle (pad2) would always follow the ball, but its y-position would always be 0.7times higher than the ball, to allow 'realness' instead of a perfect AI that always hits the ball directly and never loses.

ii. matchOver checks if a match is over if the ball touches a player's wall
iii. gameOver checks if a player's score is more than 7
iv. seed is accumulated and added by a 1 each time, where rng would use that seed to generate a different random number each time

reduceState function *[line 246-269]*

Now, the above newState and its enclosing functions are called in **reduceState**. This function would first check if a match is over (under matchOver which result is updated in the return state of newState), and if it is, a new match begins: returns back the state, except that the ball's and paddles' position are back to its initial position. matchOver is also set back to false, and the ball's next direction would be randomly chosen from the Directions array we created earlier, where the 'random number' is generated by rng. If a match isn't over, check if either player has a score of >=3; if yes, the 'posNum' passed in to newState would be bigger =0.8, to allow faster movement of ball; else the posNum is 0.5. This also means that at scores 0 to 2, the ball would move by 0.5px per coordinate.

Subscription observable stream *[line 270-277]*

Finally, we have an observable steam, **subscription** that merges all different inputs from the observables, and updates the state of the game. 'scan' would start with the initial state, and use reduceState to keep on creating new states with accumulated/new values. Using the reduced state, 'subscribe' would call the updateView function, which does the necessary updates on the Bodies and the scores in the canvas and HTML.

updateView function & restarting a game *[line 278-343]*

Following we have our very last function, updateView. This is the only function that produces side effects, due to it directly changing/updating the attributes of the elements itself. However, the path that brings us all the way down to this function is completely pure with no other side effects, because as mentioned, the functions return a newly created state which does nothing to the input state except using its stored information. Inside updateView we have a function attr that sets the given attributes of a given element, and we use that to update the position of the paddles and the ball. Then we update the scores both on the HTML and in the canvas.

If a game is over, we unsubscribe 'subscription' to stop the observable from on-going, and add to the canvas messages indicating the winner and instructions on how to restart a game. Lastly, there's a restartGame observable waiting for the user to press the spacebar key. Once that is pressed, we remove all created elements from the canvas, and reset the HTML score. Then, we call the function pong(), which would bring us back to a whole new game.

Final explanations

The above codes show a follow of FRP style. Everything passed in to ‘subscription’ are functions (except the state in scan), and all functions have no global mutable variables, which then avoids imperative programming and side effects. They are all functions that produce a newly created object without modifying anything (again, only exception is updateView).

The use of functional programming principles is also obvious, eg map, filter, merge that is applied to elements of the observable stream, or the observable stream itself. Higher order functions are also used, such as the reduceState function which returns another function newState.

Moreover, this ‘push-based’ architecture of observables allows the observables to capture asynchronous behaviour, until an unsubscribe is called. Thus, the elements of the page could be constantly updated without having to always refresh the page, which is one of the main useful feature of FRP.

Throughout the game, new states are always created. The initialState would never be modified, rather it really is just an initial state, or a ‘seed’ for reduceState. For each element coming through the observable in ‘subscription’, ‘scan’ applies reduceState, and this reduceState would take in the stream element (in this case it’s a number of 0,3 or-3) as an argument, which then in reduceState would call newState, and as we said above, all returns a new state. This new state contains the new information of everything needed in the state of a game, including the paddles’ and ball’s position, player score, match and game state. Then this state would be passed into updateView in ‘subscribe’, where all new information of the elements of the game are reflected onto the canvas/HTML.

Lastly, one extension I did was to have a faster ball, which is as said in the reduceState function part.