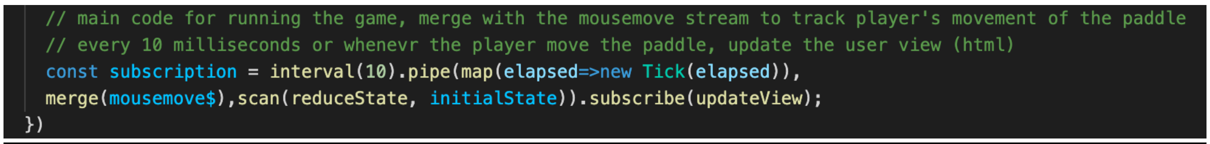
2102 Assignment 1 report

Main idea

One pure observable stream is used throughout the program. This stream is merged from 2 observables :
 (1) the **interval(10)** call, which fires every 10 milliseconds, and pipe a **Tick** object which indicates a discrete timestep, referred to the asteroids code.

(2) the **mousemove$,** which tracks the x-y location of the mouse and streams the location data wrapped by the **PlayerMove** class. This class only includes the y axis location data of the mouse, since the x axis location of the paddle is fixed and we only care about what values of the paddle's y axis location does the player want to change.



```
  // main code for running the game, merge with the mousemove stream to track player's movement of the paddle
  // every 10 milliseconds or whenevr the player move the paddle, update the user view (html)
  const subscription = interval(10).pipe(map(elapsed=>new Tick(elapsed)),
  merge(mousemove$),scan(reduceState, initialState)).subscribe(updateView);
})
```

Initially, the **initialState** (a const with read-only members) is passed to the pipe. This object contains the initial setting of the game variables. When an event is triggered (either a timestep or a mousemove, it will go through the **reduceState** function, to transform from the previous state, depending on the event triggered. This is a pure function since it outputs new state object every time it runs rather than altering the input state.

After going through the reduceState, the state object passed to the subscribe() will contain the correct values of the game variables in the current game situation. A call to **updateView** is then performed to update the user view (html document).

The whole program follows the FRP style. The flow of data is event driven, from top to bottom. It is pure in a way that none of the parts in the flow are dependent on some shared, mutable data. Each part of the flow takes the state data, works independently and passes it to its next.

Since this structure is based on observables, the stream is inherently asynchronous which allows asynchronous user-input and concurrent operations(Lew, 2020), in the case of this game, i.e. the async mousemove event and the concurrent timestep event.

Side effects are contained inside the **updateView** function that passed to the subscribe call. It is for updating the html element which is unavoidable.

Ball's bouncing angle/ position calculation

The logic to calculate the ball's bouncing angle is referred to a post in stackExchange.com which is listed in the reference section is this report. By modifying the x-y velocity for the ball whenever it hits the paddle/wall, and add the velocity to the ball's location for every tick(timestep), the "bouncing effect" can be achieved.

For bouncing from the upper/bottom wall:
The x velocity of the ball remains same and times the y velocity by -1 to make it "bounce" to the opposite direction.

For bouncing from the paddle:
Calculating the normalised distance between the centre of the paddle and the point where the ball hits, times it by the MAX_BOUNCING_ANGLE, which I set to pi/3 (60 degrees).
Denote the product above as **angle**:

For the left side paddle:
X velocity = MAX_SPEED (I set to 5) * cos(angle)
Y velocity = MAX_SPEED * sin(angle)*-1

For the player paddle:
X velocity = MAX_SPEED* cos(angle)*-1
Y velocity = MAX_SPEED * sin(angle)*-1

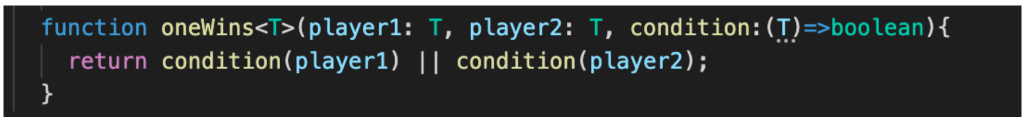
Other Conditions

- **left paddle follows the ball's movement:**
  Achieved by the function **transformAutoPaddle.** It follows the ball's direction when the ball's y location is not in between the paddle, the speed is 4 pixels/timestep. There is a wrapper function **paddleWrap** inside the **transformAutoPaddle,** which is used to restrict the paddle not to move out of the canvas.

- **Start game and end game**

  The game will start when the user clicks the start button. This is achieved by subscribing to the mouse click event of a button element. Once the user clicks the button, the button will be set to invisible. The main stream is embedded inside the button subscription. Hence, it will run only when the button is clicked.

  A function **oneWins** is used for checking if any of the player has reached the max score. It is a function that takes generic type parameters, which is more reusable compared to normal functions.



```
function oneWins<T>(player1: T, player2: T, condition:(T)=>boolean){
  return condition(player1) || condition(player2);
}
```

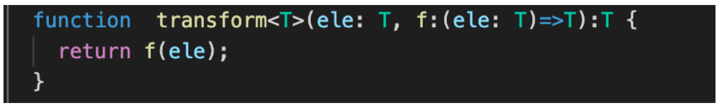
If one of the players wins, unsubscribe the main stream, announce the winner by adding text to the html element, and also reset the button to be visible to allow the player to restart the game.

Addition notes

- **The Vec class:**
  This class is reusing part of the code from the asteroid code. There are 2 functions in the class. They are both pure as they output a new Vec object instead of mutating the object itself .

  1. **Add**: add the value of another Vec object to the Vec object.
  2. **Transform**: transform a vector using the input function f.

- **The transform function:**
  This is a function with generic type for transforming an element. In the pong game, this function is used when transforming the scores of the players.



```
function  transform<T>(ele: T, f:(ele: T)=>T):T {
  return f(ele);
}
```

Reference

- In Pong, h., 2020. *In Pong, How Do You Calculate The Ball's Direction When It Bounces Off The Paddle?*. [online] Game Development Stack Exchange. Available at: <https://gamedev.stackexchange.com/questions/4253/in-pong-how-do-you-calculate-the-balls-direction-when-it-bounces-off-the-paddl> [Accessed 15 September 2020].

- Lew, D., 2020. *An Introduction To Functional Reactive Programming*. [online] Dan Lew Codes. Available at: <https://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/> [Accessed 18 September 2020].