Name: Loh Jing Wei
Student ID: 30856183
FIT2102 Assignment 1 - Frogger

Summary of Code

The code follows a simple structure provided in asteroid sample which is input handling (observable stream from keyboard event) → Manipulate game state → Update view.

Input handling

Usage of Observable

Game transitions is stored in separated class called RightMove, LeftMove, UpMove, DownMove, Tick and Restart. The constructor is used to create instances of RightMove, LeftMove, UpMove, DownMove, Tick and Restart. Each Tick instance contain data about elapsed, each RightMove, LeftMove, UpMove, DownMove instance contain data regarding both direction and rotate data. The function observeKey create observable stream of RightMove, LeftMove, UpMove, DownMove or Restart instances by taking input from keyboard using the fromEvent operator. The gameclock create an instance of Tick every 10ms using interval(10).

Body and Frog

Type for Plank, Car, Crocodile, Fly, Target boxes and type Frog

The Body and Frog interface is declared as immutable type by using the Readonly construct. Therefore, whenever instances of type Body or Frog is created, the instance is deeply immutable. The type Body is used for Plank, Car, Crocodile, Fly and Target boxes, generalising the bodies that participate in collision with the frog. Whereas the type Frog is used for our frog, with a special attribute called 'rotate'. Attribute about the position, velocity or mid-point of Body or Frog is represented using the Vec class (immutable vector class), where each Vec instances contains x and y value. Note that each Body instance has attribute id, this unique id is used to match the view object with the Body.

Vec class

The Vector class creates instance of Vec which contains x and y value. Both the x and y value is Readonly and immutable. All the functions within Vec class does not change the Vec data in place, it returns a new Vec instance to maintain purity.

Type Union

The type *viewType* is union types, it allows more than one option for the type of value that can be assigned to the variable.

Type Aliases

The type *create* is a type aliases, it is needed because the type annotation for argument f in *create_body_array* function is too long and complex.

createBody function - Curried functions

The *createBody* function is a curried function which support partial application, it returns instance of Body type. The *createCar, createPlank, createCroc and createTarget* are all partial application of the createBody function. Since instances of the same type (car, plank, crocodile, target) has the same body width, height, collision factor and viewType, these parameters are applied as the partial application of createBody function. The rest of the parameters such as row number, id, x and y position of body, velocity of body varies from one instance of the body to another, hence these parameters will be applied later on in the *create_body_array* function.

create_body_array function – Higher order function

Higher order function is function that takes in another function as parameters or return a functions. One of the example of higher order function in frogger code is the *create_body_array* function. The create_body_array function accepts an argument call f which is a partially applied createBody function.

Manipulate Game State

State interface - How state is managed throughout the game while maintaining purity?
The State interface is declared as immutable type by using the *Readonly construct*. This is because we want to maintain purity since multiple instances state will be created as the game goes on. Each time the *handleCollisionsAndScore*, *tick* and *reduceState* function is called, the function return a new deeply immutable State instance due to the Readonly construct for State type. This ensure that State's data will not be altered in other functions causing side effect.

Initial state – 'as const' assertion
The constant variable initialState stores an instance of State. The *'as const'* added after creating the State instance makes the instance deeply immutable. The deeply immutable property does not allow attribute of the State instance stored in initialState to be change, ensuring purity of the code.

Reducing States
The *reduceState* function encapsulate all possible transformation of state. When a RightMove, LeftMove, UpMove or DownMove instance is pass into the function, reduceState function returns a new state with the updated frog's pos attribute. Whereas a Restart instance is passed as parameter, it returns a new State similar to the initial state, while retaining the highestScore attribute. Lastly, if Tick instance is passed into the function, reduceState function calls the tick function.

Tick
The tick function apply transformation on the bodies and frog using moveObj and moveFrog function, update score and furthestY attribute of State. The new State after these transformation will be pass into handleCollisionsAndScore function.

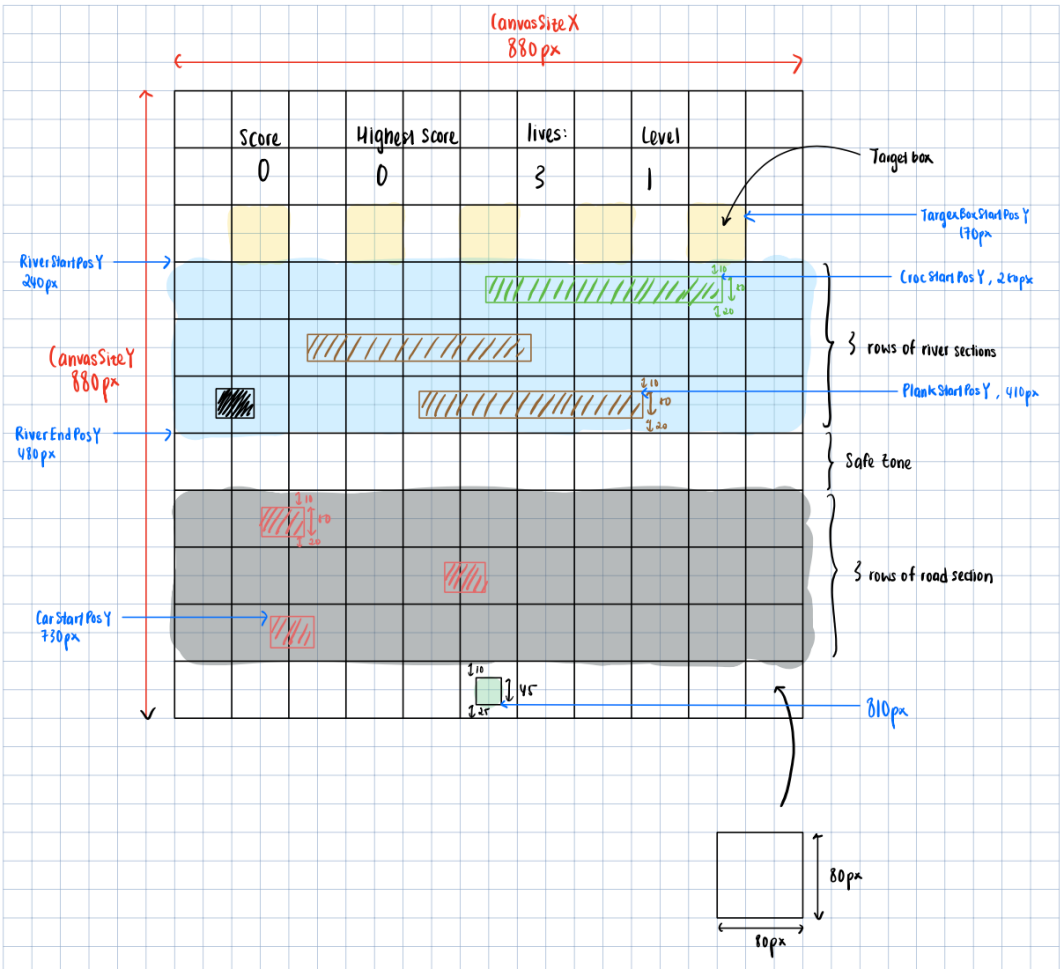
Besides that, tick function check if the State's win attribute is set to true. If the user win the game, it will return a new State with increased difficulty, by creating new array of cars and plank body with increased velocity.

Collisions

The handleCollisionsAndScore function checks for condition in which frog dies or land in target box. When collided with car or drown in river, lives will be decremented. If there is no remaining lives, gameOver is set to true. Whereas, if frog lands on target box, score increase by 100.

Canvas size and frog movement

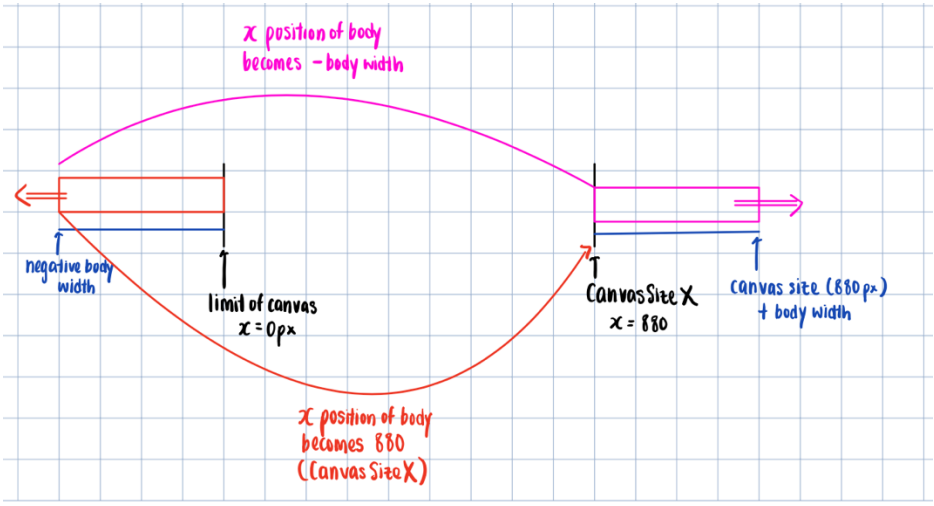
The canvas is 880px * 880px, for simplification purpose the canvas can be viewed as a huge square with a number of smaller squares. Each of the smaller squares are 80px * 80px. Hence, whenever there is a keyboard event of up, down, right or left key, the frog will hop 80px towards the direction indicated from the key.



torusWrapX function

This function allows body such as plank, crocodile and car to wrap around the canvas. The function takes in body's position vector (x,y) and the body's width as parameter. The idea behind the function is when the left edge of the body crosses the right edge of the canvas (i.e. the body's position x is greater than CanvasSizeX), the right edge of the body (the body's position x + objectWidth to get the x value of the body's right edge) is transformed back to the left edge of the canvas (i.e. position x of body = -objectWidth so that the right edge of the body (position x + objectWidth) matches the left edge of the canvas (0)) and vice versa, hence 'wrapping' the body in the canvas.

This is done instead of just returning 0 or the CanvasSizeX as the new position x of body as we would like to have a smooth transition of the body (being able to see body moving out from the left or right edge of the canvas).

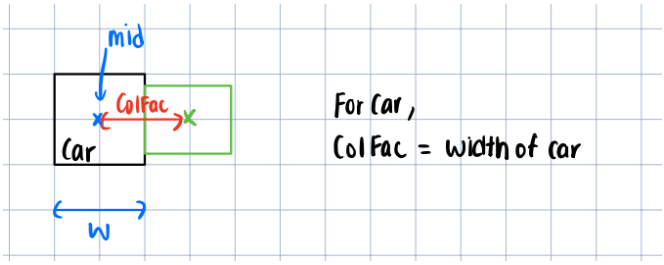


Collision factor and mid-point

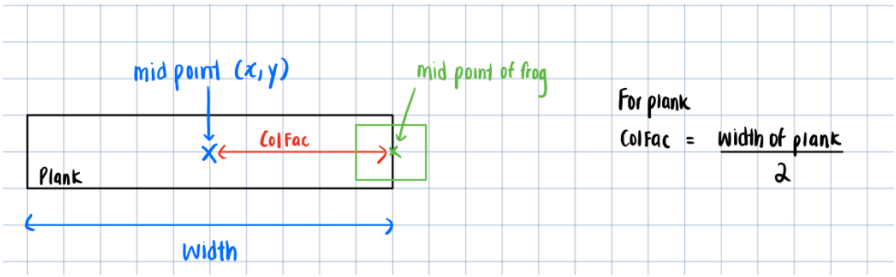
The collision factor (colFac) of a body reference the distance at which the frog collide with the body along x-axis.

In the *bodiesCollided* function, we measure the distance from the mid-point of frog to mid-point of bodies in order to decide if they have collided. We compare difference of the mid-point between bodies and frog with the colFac attribute of the body. If the difference is smaller than the colFac value, we consider it as a collision.

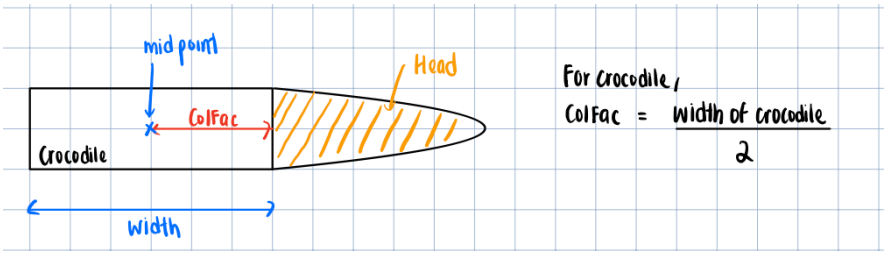
For the car, the colFac is the same value as the width of the car. This is so that when the edge of the frog touches the edge of car, we are able to detect this as a collision.



For the plank, the colFac is half the width of the plank body. The 'collision' detected between frog and plank is used to represent the action of frog landing on top of the plank, as a result we would not want to consider it as a 'collision' when the edge of the frog merely touches the edge of plank. Hence the distance between mid-point of frog from mid-point of plank has to be lesser than half the width of plank (instead of full width of plank) to be considered as a 'collision'.



The crocodile is similar with the plank. In order to make the crocodile head section deadly for the frog, the width of the crocodile is smaller than the SVG used. In other word, the head section is displayed on the screen however the width does not cover the head.



Update view – Side effect

The *updateView* functions takes State instance as an argument. It updates the HTML elements according to the data stored in State instance. This function is impure because we are outputting to the display, creating a side effect.

The *updateView* function call *updateBodyView* function on the Body instances. The *updateBodyView* function will first search for the HTML element which correspond to the Body instance using Body's id, then update their x and y position on display. If the element is not found in the document, the *updateBodyView* function will create a new element for the body.

The State also contains attribute 'utilisedTargetBox' which contains the target box that the frog lands on. The *updateView* function will unhide the frog placed on the target box in the HTML file, showing that the box is filled.

For the 'fly' attribute, when the fly body collide with the frog, *updateView* function hide the fly from display, mimicking the effect that the fly is eaten.

When the game is over, instruction on how to restart will be displayed on the screen. Pressing the spacebar will then create an instance of Restart in the subscription. Resetting the game view back to the beginning while retaining the highest score.

Subscription

The merge operator merge the observable streams created using observeKey function and gameclock into a single observable stream. The scan operation works like a reduce function, it applies the reduceState function on observable stream emitted from merge, and emits the current accumulated State instance. The subscribe operator passed each item in observable stream of State instance as a parameter into the updateView function.

Additional features – 3 lives

The 3 lives feature is implemented by adding a new attribute call lives in the State interface. The initial state will have 3 lives. The number of lives decrement by 1 whenever frog dies and the game is over when there is no remaining lives.