Assignment 1- Frogger with Functional Reactive Programming

Name: Josh Hernett Tan

Student ID: 32421982

Class: Tutorial 7 – Group 1

About

This is a basic implementation of the Frogger game with Functional Reactive Programming and written in Typescript using the RxJs library by implementing observables to make the program as pure and/or functional as possible.

Game Design

- Only accept 'a', 'd', 'w', and 's' key inputs from the keyboard to move the frog left, right, up, and down respectively.
- There is a score on top, which calculates the current score and the final score if the game ends, the scoring calculation is done by subtracting the canvas height with the current position of the frog on the Y-axis and starting from -50, this is done so that when the frog is on the initial position the score is 0 instead of 50. I also made that if the player manages to reach the win area, there are bonus points added, the bonus is calculated based on the amount of time needed to reach h the goal, the higher the bonus for lesser time needed.

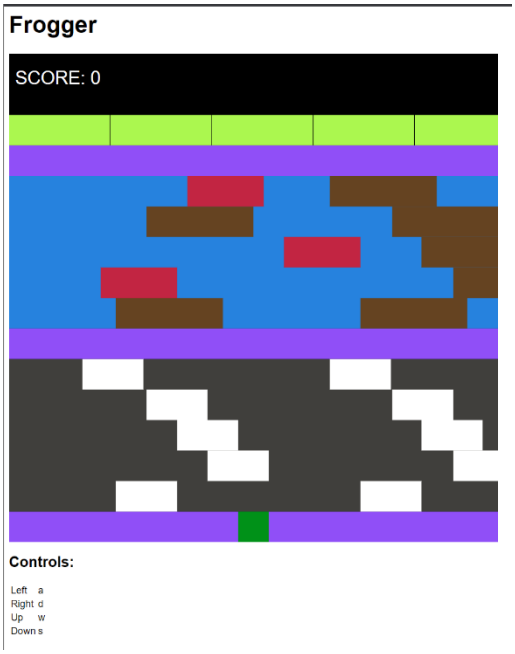


Figure 1 Score of frogger at start

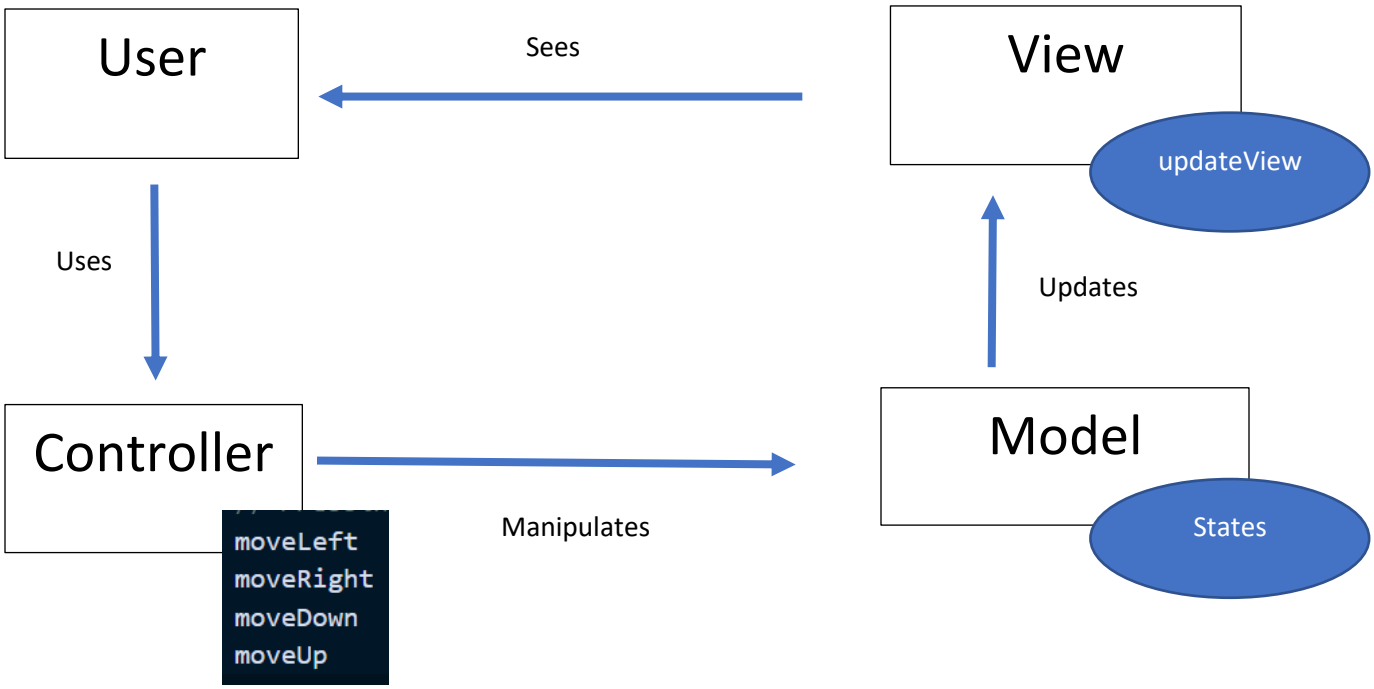
Figure 2 Score of frogger at the first row (+50)

Figure 3 Score of frogger if we won and bonus points is awared

- There are 4 game sections:
    - Safe Zone = Frog doesn't die in any rows that is coloured in purple
    - Ground = Ground section, coloured in gray
    - River = River section, coloured in blue
    - Win Area = The game ends and the player wins if the frog reaches here, coloured in light green
- There are 4 elements, differentiated based on colour:
    - Green = Frog
    - White = Cars
    - Brown = Logs
    - Red = Enemies
- Cars only appear in the ground section and enemies appeared only in the river section; the frog dies if collided with any of the two mentioned elements.
- The frog moves along with the log if the frog is on top of the log.
- Enemies has higher velocity compared to the other elements, they are also allowed to passed through the logs, so the player are recommended to not let the frog stay for too long at the log
- I didn't filter repeat for the key observables, this is done so that the player can have smoother gameplay experience whilst moving the frog (when holding down a key, the frog will automatically move to that direction until the player released the key press).
- All the element shape are rectangles to give it a more arcade vibe.

Model-View-Controller (MVC) Architecture

The diagram shows how I followed the Functional Reactive Style by separating into parts: Controller accepts the input which are the key press observables, the model is the state management, view is the visuals or the UI that's shown to the users, and lastly the users are the players or people who is interacting with the end result. This all tied together using observables in ***subscription*** function.



Observable Stream

subscription is the main observable stream that simultaneously updates the game. We *map* the *interval* (runs every 10ms) using ***Tick*** which is a discrete timestep that is triggered by the interval (so new sequential number, new tick). We then *merge* all the key observables (***moveLeft, moveRight, moveUp, moveDown***) with the stream. *Scan* used as an accumulator function that takes the ***initialStates*** and does the necessary actions or changes using the ***stateReduction*** function and returns the previous state and an updated state that's already changed.

stateReduction

stateReduction function work with two parameter, state and event. State is the previous state Event is either a ***Move*** or ***Tick*** object. If the event is an instance of ***Move***, then it meant a key press was detected. But if it was not a ***Move***, then all the non-frog functions will be called. In ***Tick*** we call ***handleCollision***, thus all the collision behaviour will be handled there, such as ***boundary*** (game over if went over the canvas border), ***carCollision*** (game over if frog collide with car), ***riverGameover*** (game over if is on river section and not standing on a log), and ***enemyCollision*** (game over if frog collide with enemies).

updateView

After *scan* we will update the game view by passing the updated states to subscribe, which then calls ***updateView*** function. This function will use the updated states and make changes to the HTML elements. This function contains impurity, but since the all the mutable actions are within here, this prevents from having side effects to the main game functions.