FIT2102

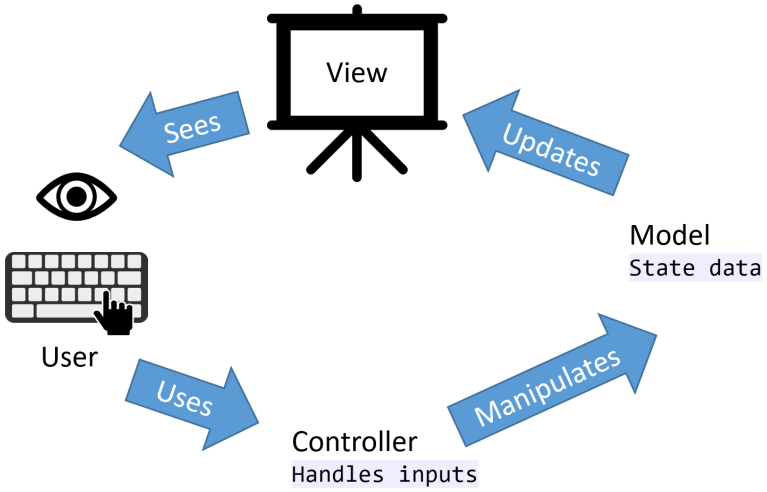
Programming Paradigms Assignment 1 - Report

Benjamin Leong Tjen Ho
32809824

Approach to the Assignment

I started the assignment by reading up Tim's Asteroids Notes and code, and looked up existing Frogger games online to use as my back-end and front-end references respectively. For the game online, I also managed to get the game icons to use in my own game for better visuals. So throughout the assignment, I used these two references as the general foundation of my code. I would like to give credit to both of these resources as they were helpful in this assignment. At the end, I managed to alter this foundation using my own creativity to make something my own with the use of FRP.

Design Decisions



Referencing Tim's Asteroids Notes and Code, I coded using the **Model-View-Controller Architecture** as it divides the code structure into 3 parts.

- **Model** - The logic (behind-the-scenes) of the game and stores the game data
- **View** - What the player/user sees when playing the game (For this game, it only consists of SVG and Image elements, displayed with the help of some HTML code)
- **Controller** - Receives information/input from the user

Doing this allowed me to separate all pure and impure code in their own sections whilst maintaining synchronisation between the front-end and the back-end.

As what the diagram shows, the player provides the input which is then received by the **Controller**. The **Controller** then processes the input and sends it to the **Model**. The **Model** is responsible for maintaining the most up to date game data/state based on the user's input (and game Tick. More about Tick later). This is very important as the **View** updates its elements based on the current game data/state. Every part of the game carries its own role and needs to all work properly for the game to function as intended.

View Component (Front-End)

To further elaborate on the **View** component, I decided to use png images to represent the Entities of the game as it was inspired by the online Frogger game mentioned above. It did mean that I had to learn a bit more about SVG Elements but in the end, I personally felt that it was worth the effort as the results looked really good.

When the page loads up, image elements are created inside the startGame() function and the elements are then given their initial attribute values before being added to the game. This is done so that we can simply reuse these elements and not create new ones, when we update the **View**. Hence, we optimise space usage and minimise any lag, ensuring smooth gameplay. On the topic of reusing elements, the game background is wrapped around itself meaning Entities that go off-screen will reappear on the other side to maintain reusability.

Model (Back-end)

Every Entity (including the Frog) are represented as an Object in the game state, each with their corresponding image element used for the front-end. I needed a way to contain the Entities such that I can effectively make use of FRP to update them as the game progresses. I decided to store the Entities in built-in TypeScript arrays, [], so that I can manipulate them with convenient functions such as .map(), .reduce() and .forEach(). These functions would then be executed for each Tick passed.

The 5 targets on the top row is also represented as an array but contains boolean values, True if the target is filled, False otherwise. This array updates when the frog reaches the final row. The game checks this array during updateView to decide whether each corresponding winFrog element should be shown or not.

One of the more subtle design decisions issues I faced was the Entity movements. The svg background has limited space and could not fit some of the rows due to how large the entities + spacing were. The lack of space resulted in entities overlapping on top of each other. Therefore, to overcome this issue, I added a new property to my Entities called offset which tells the wrap() function how much space to allocate on top of the svg width when updating the Entity's position. This ensures that the entities will have sufficient space and not overlap each other.

State Management

The **Model** is really useful but it needs 2 things; (1) a way to know if changes have been made and (2) an initial state to begin with. For the first point, that's where the **Controller** comes to play. It informs the **Model** the changes that should be made and the **Model** reacts appropriately. As for the second point, the game has an initial state, holding constant values and remains the same throughout the game. It is used at the start of every game attempt, even upon resets.

The state updates using the reduceState() function. This function is the core function for the **Model** when updating the game state. It does several things including updating Entity values accordingly, "moving" the frog's data as well as detect collisions or if the frog moves off screen to determine if the frog dies (or loses a life).

How was FRP used in the game?

Functional Reactive Programming as its name suggests has 2 parts; Functional and Reactive. So, I will discuss them separately.

Functional Programming

I used a lot of **Lazy Evaluation** in order to maintain the purity of my code. Most prominently, to create the initial game state, I need to call a function to create it, which also calls its own set of functions to create their respective game aspects (Frog, Entities, etc.). This is really useful in preventing bugs and ensures consistency in my function's functionalities.

A more impressive use of **Lazy Evaluation** is in a Random Number Generator (RNG) I made for my Fly (additional) feature. **Lazy Evaluation** allows me to create an infinite list which was what I needed for my Fly feature to work. This was done by storing a current value and containing the next in a function, preventing it from being evaluated eagerly.

Some of my Entities are contained within built-in JavaScript array types. This is so that I can make use of convenient built-in **Higher Order Functions** (HOF), such as *map(), forEach(), reduce()*, as well as compose these functions. This results in clean compact code and prevents the need for any spaghetti, complicated or messy code.

A simple illustration of the two points above would be in one of the first functions, startRivers()

```
80      /**
81       * Function to create all 5 river rows
82       *    Used in initialState
83       * @returns All 5 river rows with their respective starting values
84       */
85      startRivers = (): River[][] => range(CONS.FIVE)
86        .map((row: number): River[] => range(CONS.ENT.RIV.AMT[row])
87          .map((col: number): River =>
```

[startRivers displays Lazy Evaluation and makes use of map() and my own utility function, range()]

Lastly, some functions were **curried** so that when I have to frequently call some function with some same first argument over and over, I can simply store the function with said argument into a constant and reuse the constant with various, differing second (or more) arguments without having to input the common argument repetitively.

Reactive Programming

The basis of Reactive Programming in my code is similar to that of Tim's Asteroids Code, but with my own alterations to fit my game. In general, I made use of **observables** in order to write asynchronous code.

The game receives input through key presses on the keyboard which are then made into **observables**. Unlike the Asteroids Code, the **observables** will all emit streams of **Vec** objects. These objects contain information to tell the Model how to update the Frog.

I also made use of the Tick class and had a gameClock$ **observable** stream which emitted **Tick** objects. This serves as the 'time' in the game so that the Model is notified of what changes to make as when a **Tick** object passes.

Just like the Asteroid's Code, I combined my pieces together to create the entire game flow. The streams were **merged** into one single stream *(Controller)* and **piped** through a processing function, **reduceState()**. This ensured that my game state *(Model)* was always up to date with the user's input as well as the game's 'time'. Then for every game state, the visuals *(View)* would update when the stream's **subscription** method, **updateView()**, is executed. The use of **observables** is clearly helpful in illustrating the flow of the MVC architecture.

Besides that, the death and moving animation uses a different **observable, timer()**. The **observable** delays the next action by some period of time, in ms. In this case, the next action would be to change to the next frame of animation.

```
timer(200 * num).subscribe(() => {
    svg.appendChild(deathImg)
    timer(200).subscribe(() => svg.removeChild(deathImg))
})
```

[This is inside a forEach() function]

The death animation in particular had to have a nested **timer()** due to the nature of how forEach() works. When the outer **timer** executes, the forEach() method thinks it has completed its current execution and hence proceeds to the next element (frame). Therefore, every subsequent frame needed a longer timer delay period to produce a smooth animation!

Additional Features

Number of rows

One road and one river row each? How about 5 rows each? As mentioned in my approach to the game, I took inspiration from the online game (as well as the youtube video attached in the assignment specifications) and made 5 road rows and 5 river rows. This meant the game needed 5 arrays of Entities for each of the road and river (There is an outer array containing each of the 5 arrays for each of the road and river sections).

Lives

Accidentally drowned? 2 lives to go! The players now have 3 lives, meaning they now have 3 chances before losing. The lives do carry over across levels to make the game not too easy for players. This was implemented by adding another property to the game state and deducted when a collision occurs. The game ends once all 3 lives are gone.

Pause

About to beat your all-time high score but mom is calling you to do some chores? With this pause feature, it is not an issue anymore. Pause the game in just a press of a button (the P button specifically) and come back to beat that high score as if you never left! The game state remains the same throughout Ticks when the game is paused. Since the View is also dependent on the Model and the Model remains constant, no View will change either.

Restart

What if a player loses but wants to play again? They can simply press the 'R' key and restart the game whilst still being able to see what their high score is from previous games! By adding another observable to keep track of when the R key was pressed, the game state would be (partially) reset and the game can start all over again from the first level.

Speed up per level

To make the game more challenging, after the player has filled the 5 targets, they will proceed to the next level. In subsequent levels, the speed which entities move increases ever so slightly. This increases the difficulty for players so things are not so boring. This was done by making it such that the next x value of every entity was dependent on both their respective velocities and also the current level.

Sinking Turtles

As the name implies, this feature makes it so that some of the turtles will occasionally sink into the river, making it unsafe for the frog to land on. This was done with a similar idea to animating the turtles, by checking the gametime value and setting the turtles "sink state" accordingly. If a frog is on a turtle while it sinks, it will also sink alongside the turtle and drown in the river.

Fly

Froggy the Frog will never survive without any food. So I added flies to keep Froggy full! Flies may occasionally appear at vacant targets for Froggy to eat. If the player manages to fill a target with a fly, they will get a bonus of +200 points! The fly is a separate Entity that only exists on the top row and uses a pseudo RNG to determine the next target to land on.

Gator

It's not a real river without some gators in it. The Gator Entity acts as a partial log; its back is safe to land on, but be weary of its snappy mouth. The implementation was really simple, in the Model, it is simply a log but with a shorter width. On the View end however, it appears the same size as the Log Entity.

Enhanced Visuals

This is not really an additional game feature but I personally think it deserves to be mentioned. The game includes animation (mentioned above) and visual indications such as filled goals have frogs marking them. It definitely required extra effort but it was a fun journey learning to play around with JS + HTML when making the visuals.

Glossary for Consts.ts

The purpose of this Glossary is to give definitions to the short forms to my constants to prevent misunderstandings of what the constants really are. They are arranged alphabetically left to right, then up to down.

AMT - Amount
BNS - Bonus
COL - Column
CONS - Constant(s)
DIM - Dimension(s)
DTH - Death
FAC - Factor
GTR - Gator
G (in **G_MIN_X** and **G_MAX_X**) - Goal
HGT - Height
INIT - Initial
INV - Interval
LAN - Land
MULT - Multiplier
OS - Offset
RTT - Rotate/Rotation
TGT - Target
TRK - Truck
TURT - Turtle
VEL - Velocity
WID - Width

References

Dwyer T. *Tim's Code Stuff - FRP Asteroids* Retrieved from: https://tgdwyer.github.io/asteroids/

Stackblitz *Asteroids05* Retrieved from https://stackblitz.com/edit/asteroids05?file=index.ts

Frogger Online Game Retrieved from https://froggerclassic.appspot.com/