Samuel Yau
32732686

FIT2102 – Assignment 1 – Frogger Report

Full Game+ is implemented

Summary

The underlying code of this Frogger game attempts to implement the Model-View-Controller (MVC) architecture. To achieve this, I have segregated parts of the code to their respective sections. These respective sections are listed as follows along with their purpose:

1. **State data** to keep track of current state of game and to be used to update the HTML that a user can actually see.
2. **View** is what the user sees and allows one to make interactive decisions within the game.
3. **Controller** listens for keyboard inputs from the user, which would then be used to make a series of decisions which would update the state data.

State data

State data is initially stored within **initialState,** as a read-only type. To maintain function purity, the data is updated through **reduceState** (within scan operator)**,** which essentially detects the type of event that is occurring (Movement or a Tick; explained later in the report), then returns a new and updated state within the scan operator. Several custom types have been declared to differentiate keys / Event. Classes such as **Tick, Movement, Restart** have also been created to differentiate the types of event. These events are the only ones that could cause a change in game state.

updateFrogState is the primary function for taking in the current state and user-inputted key-press, then updating the new frog's position by returning a new Entity object with updated values to maintain purity.

checkCollisions checks the positions of each entity in the current state to see if it is colliding with the frog or not. Depending on the entity, frog may either:
1. Die upon collision (River, Snail, UFO), which causes frog lives to be decremented.
2. Ride the entity (Logs)
3. Get confused**;** wasd user inputs get inverted (Ghosts).

* There are some other features that occur during collision which will be covered in the Design Decisions section.

isFrogInRiver, isFrogOutOfBounds are helper functions to check if frog has landed in the river or has been swept by the logs off the map screen, in both cases, the frog would die; both helper functions returns a Boolean.

increaseSpeed, increaseDifficulty are helper functions to increase the difficulty via increasing the speeds of all moving entities by 10%. These functions return a State object with updated speeds of entities to maintain purity.
* Conditions for increase in difficulty is covered in the Design Decisions section.

moveFrogOnLog is to simply match the frog's speed with the speed of the log so that it appears to be riding on it. This is achieved by returning an Entity object with the updated x position to maintain purity.

moveEntity is the primary function that allows for the movement automation of non-controlled (Movable non-frog entities).
Highlight: To allow the entities to smoothly wrap around the horizontal axis of the map, the negative value of the entities' widths was used to ensure that they appeared to be smoothly coming in from the left / right, vice-versa. The negative width value was used to account for the rendering of the most right or left of the entity which would show up first if it were to be wrapped around the map.
Highlight: To control the direction of the entities, positive and negative values were used within the speed property of the entities.

Highlight: To enable the state to be frequently updated to update the visual position of automated entities (Eg. Moving logs, ufos), a **gameClock** observable has been implemented which allows the game state to be updated every 10ms regardless if there was a keyboard input from user or not. This observable is merged along with the observables for keyboard inputs.

View

All code related to directly setting / rendering elements onto the HTML is segregated into 1 single function **updateView**; all impure code / functions is in here only. Enclosed within are four functions that does the following:

updateEntityView	**:** Creates an entity's HTML element and renders it onto the canvas or update its position via the x and y attributes.
removeEntityView	**:** Removes an entity's HTML element from the canvas.
updateCounterView	**:** Updates the high-score/lives remaining of player on the HTML. It also displays a death and a game-over notification
restartCounters	**:** Helper function to call updateCounterView to remove death notification and show a game-over notification upon game-over instead.

updateEntityView, removeEntityView are passed as callbacks into the map function to be applied to each of the entities in the game (Eg. Ufos, ghosts, snails, logs, crowns).

Controller

All keyboard inputs are merged as observables within the variable subscription, then pipe is used to process this stream of data. Primary observables for movement inputs are w, a, s, d.

Highlight: To enable the game to be restarted by the user, the operator **takeWhile** is used to take values from the stream of data until it is game-over. Another operator used to help achieve this effect is the **repeatWhen** operator, which essentially restarts observable by taking the data since the start of the stream again when the restart key (r) is pressed.

Design Decisions & Justifications

As previously mentioned in **Summary**, the code follows the FRP style by using pure functions (with the exception of the inevitable **updateView** used to update the position of elements on the canvas). Below are the features with the interesting ones highlighted.

Features

1. Frog : Jumps in fixed distances for ease of predictability for user to guess landing area. Controlled by wasd key inputs as those are the typical inputs used for movements in most games.
2. Ufo : Simple entity that kills the frog on collision. UFO movements are automated.
3. Snail : Simple entity that kills the frog on collision. Snail movements are automated, however it is un-ironically the fastest entity.
4. Ghost : This entity would scare the frog upon collision, causing the frog to be confused and making the user's inputs inverted for 2.5 seconds (Eg. w moves backwards, a moves right).
5. Logs : Frogs can land on the logs and ride them to cross the river. However, if the frog is still on the log as it leaves the canvas view, the frog would die.
6. Crown : This is the only entity that the frog can land/collide into to gain points. Upon collision, taken crowns disappear and frog gets teleported back to its initial position so that the user cannot simply take all crowns at once! However, if the frog dies and still has remaining lives, it will teleport back to initial position and replenishes all taken crowns. Each crown is worth 50 points, however after achieving each crown, the difficulty is also increased by increasing the speed of all moving (non-frog) entities by 10%.
7. Death : Upon death, a death notification "You died!" would be displayed for 2 seconds on the HTML. The timer for 2s is implemented by decrementing the **deathTimer** within the state during every tick.
8. Gameover: Upon game-over, or when frog has no remaining lives (Also displayed on the screen). All entities stop and a game over prompt is displayed on the HTML, prompting the user to press R to restart the game.
9. Highscore : Highscore is kept across all lives in a single round (before game-over). Highscore value is simply the highest score achieved across all lives in a single round.