Assignment 1 Report

BY

Samuel Tai Meng Yao

32025068

Table of Contents

Game Overview

The game is built with reference to the classic Frogger Arcade Game. The concept is similar where the user must control the frog to avoid all the obstacles and try his/her best to score the points at the target area with some extra features in the game. The game is separated into 2 zones, which are the ground and the river. The ground is safe on its own and having some vehicles that move along the ground horizontally that can kill the frog if collided. There are 2 safe zones on the ground where there are no moving obstacles that might kill the frog. On the other hand, the river consists of planks which is safe for the frog to land on and a crocodile that is safe if the frog landed on its body but will be eaten if it lands on the mouth of crocodile. The frog with move with the planks/crocodile that it landed on depending on the obstacle's speed. There are also flies in the river where it can boost the points of the user by 100 if he/she hit the target area after consuming the fly. Landing in the river will kill the frog. If the user successfully hit all the target area, he/she will advance to the next level where the difficulty is increased (obstacles are moving faster). The player must complete each level by 200 seconds, and they will have 5 lives throughout the whole game before the game over.

Code Summary

All the codes that are managing the actual game execution is written under the main() function. The main() function consists of some utility functions, the codes that initialize and maintain the state of the game. The game is developed in a pure fashion way to keep the side effect into the minimum except the functions that update the SVG objects, the handling of restart after game over and the RNG class where I utilize the Math.random() to choose the seed for the lazySequence of random numbers to ensure freshness every time we use the function. The base of the game is developed on five observables which keep listening to the keyboard input where we choose what action to perform, a restart button observable and a special observable (BehaviourSubject) that keep track of the state of the game (game over or not) to decide the subscription of the main game stream. The codes have been broken down into different sections in accordance with the Model-View-Controller(MVC) architecture to increase the readability. The game is started by calling the startGame() function which subscribe to the main game stream which begin with an initial state, reduce state and update the view of the game according to the action performs by the user.

Design Decision/Choices

To begin with, I have declared a deeply immutable Constants object at the beginning of the code which store all the fixed value in the game such as the height, width of cars and more. It is not about hardcoded the state of each object but to ensure that the game will have a specific pattern to play. Besides, declaring these states as constants can improve the readability of the code and maintain the conformance of the value if we used it in different part of the code. This design choice follows the Functional Programming style as the value cannot be modified to maximize the purity.

The objects that physically participate in the game is mainly composed of 2 types, which are the Body type and MovingObject type. The difference between them is the latter has speed while the formal does not have. The MovingObject is of type IMovingObject which is an interface that extends the Body type to reuse the similar attributes declared in the Body type.

The game started with the initialization of the objects in the game. The batches of different objects/obstacles are created separately using multiple small pure functions that combined to be used as the code needed to create these objects are the same. Hence, grouped them into functions can guarantee the extensibility, maintainability, and readability of the code with adherence to the modular programming style. To ensure the freshness of the game, I have randomized the movement of the moving objects (e.g., speed and direction) by utilizing the pure RNG class that is called in a lazySequence data structure.

The observables are used as the input stream that keep track of the actions performed by the user. Once any of the KeyBoardEvent is detected by the observable, it will create a class that associated with the respective action. The classes is needed in this case because I am using instance of pattern-match inside the reduceState() function to update the game state. Furthermore, I designed the game in such a way that if the moving objects has moved beyond the size of canvas, it will always return to the opposite side of the canvas and the frog will stop if it reaches the boundary of the canvas. The reason I did this is to make sure that we can always reuse the SVG objects created earlier to maximize the efficiency of the game and maintain the logic of the game as it will be a bug if the frog can reach the target area by jumping back.

Next, I have designed the game such that the frog will move along with the planks/crocodile depending on the speed of the obstacle that the frog landed on as this makes more sense and more interesting than having the user itself to move the frog. The interesting interaction of the frog with all the moving objects in the game is handled by the handleCollision() function which is called in the tick method Since the handleCollision() function handling all of the interactions logic in the game, it will requires a lot functions in the game. So, I have built some composite functions to define the complex transformations from simple, reusable function elements which follows the Functional Programming style as this encourages factoring functions for maintainability and code reuse. All the functions are done in a pure manner by returning new state with update value without manipulation of the original arguments. For instance, for the operation on the array, I will always use the function that will return the new array with updated value (e.g., map).

Meanwhile, I have designed the game such that there are 2 different restart features. One is by using the “KeyR” to perform in-game restart that remembers the previous round’s high score and the current level while the other one is game over restart by clicking the restart button that appear after game over where everything start from the initial state. It is designed in this way so that the game appears more interesting and encourages the user to play with cautions. Both involve the use of different observables to keep monitoring the whether the restart event is happening.

Finally, the updateView() function to render the state of SVG objects and it is also the place where most of the impurity and side effect are contained. I designed it in this way to limit the spreading of impurity and it is easier to manage. The other impure code outside of the function is when I use Math.random() to generate the seed for RNG class and the restart button where I need to hide and show the button by manipulating its attribute which is unavoidable unless I cramped everything into the updateView() function.

Functional Programming / Functional Reactive Programming style

- **Application of functions to elements of containers to transform to a new container**

    - All the functions I used ensure purity without mutating the state of the container but updating the new value and copy it into a new container
    - For example, using filter, map, etc. over arrays or using scan and map in observable.
- **Pure functions**
    - While updating the data structure in the code, I always used pure functions such as the spread operator "…" to copy all the properties from a data structure into a new one and redefined the one that needs to be updated. Hence, ensuring purity of the code.
- **Eliminating Loops**
    - I used Array.map, Array.reduce, Array.forEach, etc to replace the loops as logic of the loop body is specified with a function which can execute in its own scope, without the risk of breaking the loop logic
- **Callbacks**
    - I have combined the KeyBoardEvent and mouse click event listener to the key and the button with the observables to allow asynchronous execution of the code while waiting for the events to happen.
- **Method Chaining**
    - I used the method chaining in the code when it is available to simplify and improve the readability of the code.
- **Composition Function**
    - As mentioned in design decision section
- **Generic types**
    - I have declared and used multiple function of generic types as utility functions to improve the extensibility and reusability of the code

Usage of Observable (beyond simple input)

- Used of observable to keep track of the time in the game
- Introduce a new special Observable (behaviourSubject) which is used in the main game observable subscription to decide whether it should be unsubscribe (when game over).
    - The value of the BehaviourSubject is initialize to true
    - When game over, change it to false using next() to decide its next emitted value
    - Created a function that retrieve the value of the subject using getvalue()
- Using merge function to merge multiple observables (KeyBoardEvent) together and pipe it into takeWhile() and scan() function
    - **Where** takeWhile take in the value of BehaviourSubject defined earlier to determine whether to continue the subscription instead of manually unsubscribe in updateView() function
    - **Where** scan to keep reducing the state of the game and pass them into the subscribe function.

Additional features

1. **Added in life**: 5 lives for each round (each game play)
    a. Number of lives will remain the same until the game is over (In-Game restart will not restart the frog's lives)
2. **Added in timer**: 200 seconds for each level

a. Lost 1 life if the player does not complete a level within the specified time

The reason that I have added these 2 new features is to increase the tension during the game as well as giving the player more chances to play on the current level which they might have spend some time to advance to this level.

The new timer features that I have added in basically utilize the gameClock Observable Stream to keep track of the elapsed time. Besides, both new features also required the state management to be done well as I will need to keep track of the calculation of the timer when in-game restart or level up happen as these conditions will not triggered the resubscribe of the game, hence keeping track of the restarting time or level up time is essential.

Same as before, all the necessary manipulation of state of SVG objects or HTML text content are contained in the updateView() function to ensure purity of the code. Most of the new code is affecting the state management in the handleCollision() function where I decide to group up similar code in the function to a variable to ensure the reusability of the code and the conformance of the result in different kinds of scenarios. All these manipulation follows the Functional Programming and FRP style