Assignment 1 - Functional Reactive Programming Report

For this assignment, a version of the orignal arcade game “Frogger” was developed using a functional programming style. This means that the code was constructed with the idea of using functions to their best ability, keeping them pure and making sure data is immutable among other things. In the context of this assignment this was done by making use of the RxJS module, using TypeScript and making good design decisions.

Observables
The game runs on a tick system meaning that the game updates every tick. This is done through an Observable created using “interval(10)” which outputs ascending numbers every 10ms (game ticks every 10ms). Each action that can modify the game is implemented using an Observable, this includes player movement.

Spawners for cars and planks are also implemented using Observables. This allows us to easily decide what ticks to spawn objects by piping “filter” and makes the code for these parts coherent and easily extendable. Actions may need to store data which is done through individual classes for each action. This allows the use of the instanceOf which makes the code easier to implement and understand.

The game state is stored as a read only object in order to maintain immutability, a core FRP principle. Each time the game state needs to change (an Observable outputs) the reduceState function is called. This is a pure function which takes the current game states and returns a new game state with the given action performed. The use of the observables coupled with the pure function does not cause any side effects and allows us to easily expand and add game functions as neccessary in the future.

Objects
Objects in the game are the River, Cars, Planks, Turtles, Targets and Coins. Data for these objects are stored in a read only “body” type. Having a type for these objects allows better reusability of code and keeps things consistent. The data for these objects are stored in the state in arrays for each corresponding type. This means that as mentioned before when the states need to change

according to an action the objects in the game can also change showcasing the coherency of the code. For example each tick in the game Cars, Planks and Turtles have to move according to the deltax and deltay. This is done by calling “map” on the previous array with function moveObj which updates the objects x and y coords. A similar type of process occurs when objects need to be removed from the game state by using “filter” filtering out objects that have gone out of the canvas. Keep in mind that this is pure as map and filter returns a new array rather than modifying the old one once again maintaining FRP principles.

The location and spawning of the targets and coins are hardcoded which in theory may be bad but keeps the game consistent and is reasonable in this use case.

Collisions
Collisions are checked every tick by calling handleCollisions. This function takes a state and returns a new state which has detected and acted on all the collisions that have occurred in that tick. An example of good functional alprogramming can be seen through the bodiesCollided function that was defined inside the handleCollisions function. This showcases how you can easily separate concerns with functions, handling things separately, allowing for code that is easier to write and easier to understand.

This function allows us to easily add new collision detection for new objects and implement different actions. For example a coin collision acts differently to a car collision as when a collision occurs with a coin the score increases but when a collision occurs with a car it is game over. This highlights some of the pros of FRP.

Something interesting to note is how planks and turtles work, if the player is colliding with the river it is game over unless it is also colliding with a plank. A Turtle is virtually just a plank so if it is active it acts as a plank but when submerged it doesn’t detect collisions. This once again displays the ability to reuse different parts of the code and the extensibility due to good design choices.

Game Over
When the player dies it is Game Over, allowing the game to be restarted. The game still ticks in this period but the player is simply unable to move. There is an Observable for the Space key which resets the game to a state similar to the

initial state but with updated scores. By doing a reset this way it keeps everything consistent and allows the game to flow well.

Difficulty
When the player hits all 5 targets without dying the targets respawn and the difficulty increases. The difficulty simply scales the speed of the objects. This is done by simply multiplying the speed by the difficulty which is a number that starts at 1 and increases by 1 each time the 5 targets are hit.

View
After every observable output once reduceState is called the state is then displayed on the HTML with "updateView" which takes the state as an input. This is the only part of the code that contains side effects as it is modifying the HTML elements and adding/removing things to the document. It contains a useful function display which allows us to easily add new objects to display.

This now allows us to showcase the completed Model View Controller architecture of our game. The user controls the game through the use of observables and pure functions which produce a model, the game state. This state is then viewed with the updateView. This all allows for code that is easily manageable, modular and organised.