FIT2102 Assignment 1 Report

Code working and game highlights

I will start off by talking about my *tick()* function, as it is the backbone of the whole game. My *tick()* function returns distinct states based on various circumstances that are supplied by the input state, therefore various states are kept in existence. It is used to handle collision and ending the game. It is also used to animate objects by using the *animateObjects()* function.

Another important aspect that my *tick()* function ensures is the wrapping of objects from both ends of the canvas using the *torusWrap()* function makes this achievable.

While playing the game, the game's objects move more quickly when the frog approaches a specific target location.

Crocodile is one of the enemies of the frog therefore, landing on its head will cause the frog to be eaten resulting in ending the game, landing on its tail results in game running normally. Similarly, a *turtle* is another enemy of the frog, landing on it while it is *submerged* in the ocean will result in game over.

Static objects like cars, trucks, logs etc are first positioned using *fixedObjProp* types before they begin to move. I have referred to it as fixed to demonstrate that it is not intended to move at the first instance when the game first begins.

Application of Functional Reactive Programming in Code

By making sure that no global state is mutable, the functional programming paradigm enables us to create a *Model-View-Controller* architecture that ensures functional purity. I have strictly followed this in code by creating functions that return a new state rather than changing the input state. Furthermore, I have ensured FRP by using *pure observable streams* to handle reactiveness.

Looking at ***line 604*** of my code you may notice I have used an observable operator called *scan* which does not alter the initialState (the global variable) rather it changes the initialState to a new state which is decided by our *reduceState* function. The *reduceState* function outputs a distinct state depending on the type of input instance formed by user interaction (like a keydown or keyup event) or interval streams.

Similar to FRP Asteroid, each stream of event is transformed into *objects* of classes, such as Move, Restart, Tick. This is required for the *instanceOf* objects pattern matching in the reduceState function. Different events return different states depending on the game condition.

Moving on, after getting one final state from the *reduceState* function I called the *updateView* function in the subscribe call ***line 607***. This is an impure function and is responsible for displaying the change of states to the user.

This high-level description of the program's design demonstrates the fact that there isn't actually any global state to change in the first place.

How is purity maintained in the code

Declaring a javascript container as *Readonly*<> or *ReadonlyArrays* is important as it plays a crucial part in ensuring that no object is mutable because the values are read-only meaning that they cannot be changed. In my code I have made use of this frequently in every Type, Array or Instance variable used. For example ***line 146*** Type State is declared as *Readonly*<> and ***line 149*** Array Cars is declared as *ReadonlyArrays*.

I have made use of array functions like *filter* and *map* to cater my ReadonlyArrays. There are no side effects when using array functions because they are referentially transparent, and they also guarantee that a new Array with the changes made is returned. By employing such functions I can avoid for loops that ultimately result in *imperative style* coding.

Furthermore, in my code I have declared my global variables *as const* meaning that there won't be any case where they accidently mutate. Coding in *TypeScript* gives us further purity as we use type annotation which strengthens *immutability.*