# FIT2102 Assignment 1 Report

*By Ong Li Ching 31190057*

---

### Summary of the code and game highlights

The **Frog** is the main character of the game and is displayed as a **green circle** in the game. The starting position of the Frog is at the bottom middle. It can move forwards using the up arrow key, backwards using the down arrow key, left using the left arrow key, and right using the right arrow key.

The game is divided into **5 sections**: the Ground section, the Safe Zone, the Water section, the Target Areas and the Score section. All the **objects** in the game are displayed as **rectangles**.

In the **Ground** section, the objects are Cars, Trucks and Flies. There are two rows of Cars with different direction and velocity, same goes to Trucks. The Frog **dies** when colliding with a **Car** or **Truck**. If the Frog eats a **Fly**, the Frog will receive **50 bonus points** when it reaches a Target Area. The flies only appear in the first 10 seconds of each level.

In the **Safe Zone**, the Frog can take a rest since there are **no objects**. In the **Water** section, the objects are Logs and Turtles. There are two rows of Logs moving with different velocity and one row of Turtles. The Frog can **stand** on Logs and Turtles and it **dies** if it **lands** in water. There are two types of Turtle. One is a normal Turtle where the Frog can stand on it. Another is a Turtle that **teleports** the Frog back to its starting position.

The Player scores **100 points** when landing the Frog in a **Target Area** that is **not filled**. If a Target Area is filled, the Player will not score any point if the Player tries to land the Frog in the target area, and there will be a **big green circle** indicating the Target Area is filled. If **all** the Target Areas are **filled**, the Target Areas will be **cleared**, the difficulty **level** will be increased by **1** and **50 bonus points** will be given as a reward. When the difficulty **level increases**, the **speed** of the Cars, Trucks, Flies and Logs **increases**.

The highest score over the previous rounds, the current score and the current difficulty level is displayed in the **Score** section. When the Frog dies, the game is over. The Player can **restart the game** by pressing the key **r**.

### Design of the code

The **Model-View-Controller architecture** is formed by using the functional programming concept to ensure the **purity** of the code.

I have used **pure observable streams** which will return a new state object without modifying the original state. The **scan** Observable operator is used to transform the **initialState** into a new state. The new state will be decided based on the conditions in the **reduceState** function. There are four types of event class in the code. The reduceState function will match the event triggered with the conditions in the ternary operator using the **instanceOf** operator. For example, the event triggered is an instance of the **Tick** event class. The reduceState function will execute the expression where the condition the event is an instance of Tick event class is true.

Both **tick** function and reduceState function returns a new state based on some conditions. The tick function returns a state corresponding to the input state.

After that, the **updateView** function in the subscribe call is executed. It is the only impure function in the code, and is called at the very end so that it does not mutate any state that is being used at the moment in the Observable.

Curried functions are used, for example **createRectangle** function to create objects at the start of the game. It helps us to avoid passing the same variable again.

### How is purity maintained?

The **readonly** keyword is used very frequently in the code, especially for **type** and **array**. It ensures that the properties in the type and array are read-only. Hence, we cannot re-assignment the properties.

Besides that, array functions, such as **map** and **filter** are used for **ReadonlyArrays** in the code. This is because the array functions have no side effects. The functions ensure there are no modifications in the original array, and return a new array. Hence, avoid imperative programming style.

The **Constants** variable contains values that are constant, and is declared with the **as const** keyword in the code. This means that the values in the variable cannot be changed. It helps to avoid the cases where the values are changed accidentally in the code.

### Improvement can be made

Firstly, there is a bug in the code. When the Player restarts the game, the flies are supposed to appear for 10 seconds, however they do not. This is because when the Player restarts the game, we create a row of flies with the state AppearTime equals to 0. However, the game clock does not restart from 0. This can be the cause of the bug. More time would be needed to fix the bug.

Secondly, creating different states for objects of the same type with different velocity may not be efficient. In the future, if we want to create cars with even faster speed, we would need to create another state of the type ReadonlyArray to store the cars with faster speed. This is not efficient. Instead of creating new states for objects with the same type, we can combine them using the concat function.

Moreover, we can generalise the sections as rectangles instead of having separate functions for each of them. For example, generalising the water section as rectangles makes it easier when we want to create multiple water sections as we do not need to create multiple functions for each water section.

Lastly, when the difficulty level increases, we calculate the change in velocity of the objects one by one manually. This can lead to mistakes in calculation easily. We can create a function that calculates the new velocity.