Name: Bryan Jun Kit Wong
Student ID: 32882424

FIT2102 ASSIGNMENT 1 REPORT

Part 1: GAME DESIGN AND CHOICES

Similarly to the classic Frogger game, the map is split into two regions - a road filled with vehicles and a river with floating objects, with a "safe zone" bisecting the two. Six different target areas lay at the end of the map, which Frogger has to reach amidst many perils. On the road, the player has to dodge vehicles of differing sizes, moving at varying levels of speed and direction. But be wary, as merely grazing a car can cause instant death for Frogger, initiating a Game Over. Players can reattempt by pressing the R key on their keyboard. In the "safe zone", players can take a breather and plot their next move, for the next section is no easy feat.

It's the 21st century, and rampant industrialisation and the general disregard for the environment have caused heinous pollution in the river, rendering it highly toxic for Frogger. Ironically, the pollution benefits Frogger as thrown away floating planks in the river can be safely stood on and used to cross dangerous waters. However, They must proactively move along with the planks to avoid certain death. Feral crocodiles roam the waters, a natural predator for Frogger. Luckily, all of them are fast asleep. Unlike the classic Frogger game, players can temporarily safely step on the crocodiles, but not too long! Standing on the crocodiles for more than 2 seconds will wake them up and attack Frogger. Keep moving and count your time on a crocodile, lest you become their dinner.

Your hard work pays off! Reaching the end and filling the targets will grant 100 each to their score. Each unique row moved by the player, their score increments by 5. "Unique" means the player cannot move up and down on the same two rows and generate infinite score. Filling filled targets will net the player 0 score gain. Filling all the targets will add 200 to the player's score and will slightly increase the speed of all game objects, making each subsequent round noticeably more challenging. Frogger is hungry and fulfilling its cravings will reward the player! A fly can spawn somewhere on the map, providing 50 score once consumed. The highest score from previous rounds is saved and displayed at the bottom of the screen, so how high can you go?

Name: Bryan Jun Kit Wong
Student ID: 32882424

POTENTIAL IMPROVEMENTS

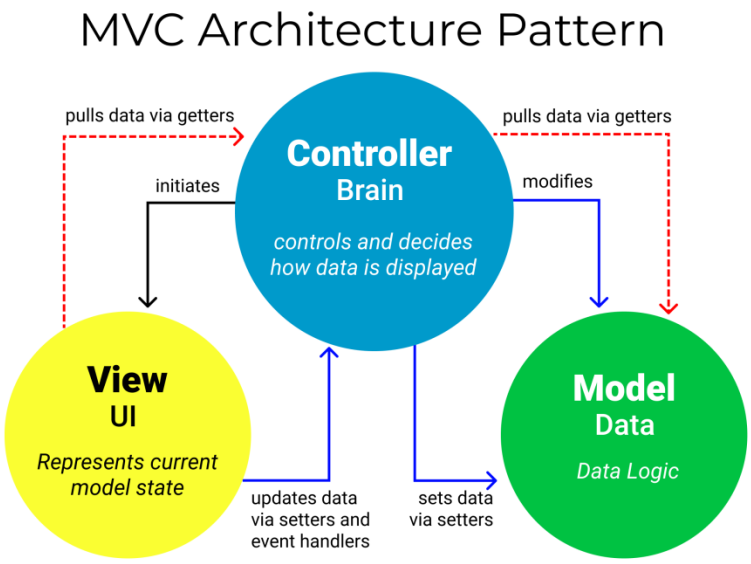
One detectable key issue in the game is the lack of randomness. All objects in the game are the same each round; they have the same direction, speed and sizes from the previous round. At the game start, all game object values are hardcoded. A way to improve this oversight is to implement a pseudo-random number generator, which will randomise game object values every round. Through this, each round will feel fresh and different, leading to less bland gameplay.

On the other hand, generating random values are non-deterministic as every single round; the functions would return different values each time. This goes against the principles of functional programming and referential transparency.

Part 2: CODE DESIGN CHOICES AND OVERVIEW

CODE ARCHITECTURE

The game is built on the ground up using the Model-View-Controller (MVC) architecture. MVC allows the separation of frontend and backend elements while simultaneously providing abstraction to the users. This is illustrated in the diagram below:



As shown above, the View (frontend) and the Model (backend) are disconnected; meaning changes made in one does not affect the other. Through this, side effects are isolated as any mutation within the View will not affect the Model data. This allows my code to be more

scalable or maintainable as changes made in Model or View will not create unintended consequences in the other.

The Model represents and manages the game data, which is done by having a State object, containing all the properties and data of the game. The state is set to ReadOnly to make it immutable and disallow side effects. The View contains all the SVG and HTML elements which display the game UI to the player. The controller is the "brain" of the whole architecture, having the role of "modifying" data (the data of the Model is not modified, just recreated with new values, giving the illusions of modification) based on specific actions. In this case, my observables will cause the controller to act, particularly the input and clock observables. The View is my updateView function, which mutates elements to update the display for the user whenever the state is changed. Since all mutation of impure code happens within the View, the Model is left untouched, resulting in no unintentional side effects and limiting bugs.

FUNCTIONAL PROGRAMMING

Immutability and Purity

All functions are pure, and every variable declared is immutable in my code. Using this allows my code to be referentially transparent and deterministic. My functions will always reproduce the same output for any fixed input, reducing potential bugs produced. Utilising pure functions also improves the maintainability and transparency of the code because I will know everything that occurs in the code. Pinpointing errors or loopholes in my code will be extremely hard if there are side effects. This is because side effects will trigger a domino effect, affecting all the functions that rely on each other, causing a cascade of bugs and errors. As a result, I will be forced to carefully comb through my code to resolve this issue, wasting a sizeable amount of time.

Global variables are avoided (besides some fixed constants), as accidentally mutating the global variables may result in any code that relies on them no longer being referentially transparent, producing unexpected outcomes and breaking the game. All variables being immutable ensure that Frogger will always run the same every time.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Higher Order Functions, Currying and Continuations

Some functions are curried. The function createObstacle demonstrates this as the viewtype is curried. Besides improving the readability of the code, it also reduces repetitions as the newly created function is reusable. An example of this is createCars which curries the viewtype for cars.

Multiple continuations are used like createAllFlies. It will return the result of createFlies to the caller, improving the readability of the code as it decomposes parts of the code into functions. Through this, it separates the massive blocks of code into readable chunks.

OBSERVABLE AND RXJS USAGES

Numerous observables are used in game creation. The key$ observable is one of them. It listens for key presses, filtering out any that do not match the list of acceptable inputs (WASD or R) and then merges them into one observable keyboardAndMouse$. Similarly, we have another observable that imitates a clock. Every 10ms, the clock$ emits an observable, triggering the tick function inside the reduceState function. The mouse$ observable will also emit an observable every time a mouse click occurs over the start button, starting the game when pressed.

Lastly, some interesting observables to highlight are within the showHighlight function. They function similarly to the key$, except instead of moving the frog, they will highlight the HTML elements and provides input feedback to the player. Furthermore, the mouseOnStart$ and mouseNonOnStart$ will listen for mousemove events over the start button. What it essentially does is changes the colour of the start button to be darker if the mouse cursor is over it, implying to the player that it is interactable. Then the entire showHighlights is called every 10ms by subscribing it to the timer observable.

(1197 words excluding headers)