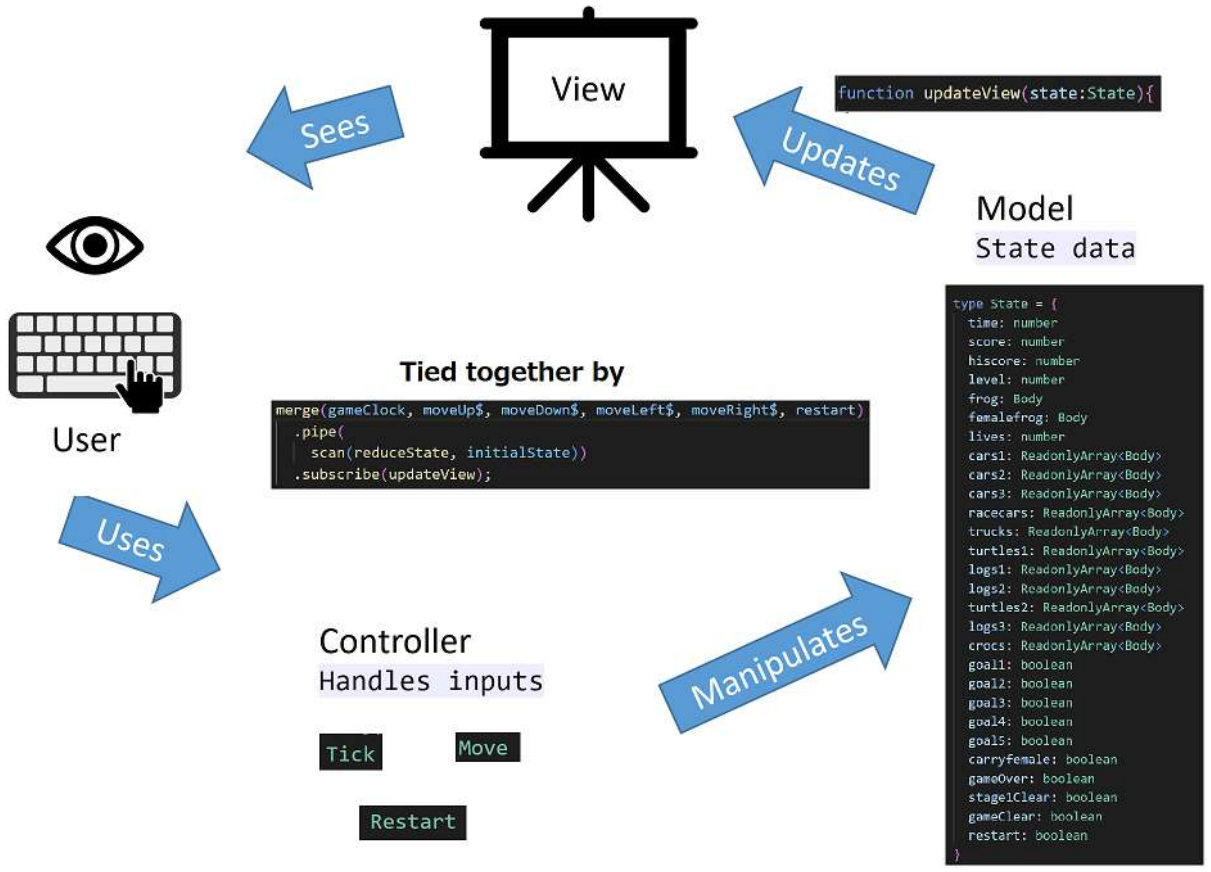
Overview

The following diagram illustrates how I have coded using FRP style. Following the Model-View-Controller (MVC) architecture, I made the game by separating state management, its input and manipulation, updating visuals into different parts, and finally combining all the pieces into one merged Observable stream.



```
function updateView(state:State){
```

```
merge(gameClock, moveUp$, moveDown$, moveLeft$, moveRight$, restart)
  .pipe(
    scan(reduceState, initialState))
  .subscribe(updateView);
```

```
type State = {
  time: number
  score: number
  hiscore: number
  level: number
  frog: Body
  femalefrog: Body
  lives: number
  cars1: ReadonlyArray<Body>
  cars2: ReadonlyArray<Body>
  cars3: ReadonlyArray<Body>
  racecars: ReadonlyArray<Body>
  trucks: ReadonlyArray<Body>
  turtles1: ReadonlyArray<Body>
  logs1: ReadonlyArray<Body>
  logs2: ReadonlyArray<Body>
  turtles2: ReadonlyArray<Body>
  logs3: ReadonlyArray<Body>
  crocs: ReadonlyArray<Body>
  goal1: boolean
  goal2: boolean
  goal3: boolean
  goal4: boolean
  goal5: boolean
  carryfemale: boolean
  gameOver: boolean
  stage1Clear: boolean
  gameClear: boolean
  restart: boolean
}
```

General

Declaration of Constants

All values used in the game code are declared within a constant object shown in the screenshot below. This mitigates side effects since the constant values can never be changed elsewhere as well as provide flexibility in making updates to the game.

Usage of TypeScript

Throughout the code, type annotations are used in function parameters and function returns to ensure correctness of the code at compile time. Union types are also used for the possible events that could be used in the state transduction phase. It uses instanceof to query the type of the event and does the right thing in each case. An interface for all the Body objects is declared which defines the set of properties and their types which is expected to be available to all objects of that type.

```
const CONSTANTS = {
```

Observables

Observable Streams

I created different Observable streams to detect the various in-game events. There are Observable streams for moving up, down, left, right; restarting the game, and the game clock. The keyObservable function is created to reduce repetition of creating the same type of Observables from keydown events. The observable streams are then merged into one using the merge function for the main game stream.

Method Chaining

The main game stream is managed by chaining a sequence of functions which follows a fluent programming style.

State Creation

State object

An object of type State is created, which will be manipulated throughout the gameplay. All gameplay updates will be changed solely inside this State object.

Game States

There are three game states: Move, Restart and Tick. The movement of the player is detected by keyboard events upon pressing w (up), s (down), a (left), and d (right), which will create an Observable with a Move signal. Likewise, a keyboard event of pressing r creates an Observable of a Restart signal. Interval creates an Observable stream of Tick signals, which represents the game clock, which emits signals every 100 milliseconds.

Functional Patterns

The logic of looping through arrays are abstracted into functions such as forEach, map and filter to eliminate loops. The 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.

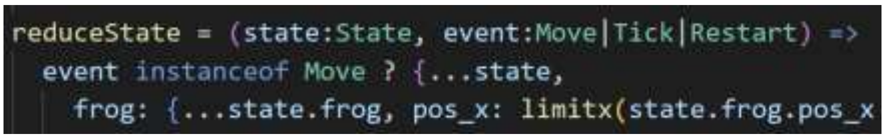
Lazy Evaluation – Curried Functions

Multiple curried functions are declared for the use of object creation so that I can provide computation to obtain the value at a later time. I provided the view types of the object to the create__ functions, but not their specific object id that the Rect object itself. Their actual parameters are passed in later during the object creation phase.

State update and Manipulation

Updating Data Structures with Pure Arrow Functions

The data structures within the code, such as State and Body, are updated with pure functions. As shown below, all the properties from the existing state into a new object using the spread operator, followed by more JSON properties that can potentially overwrite those of the original. The same is applied to change the frog, which is a Body. Note that arrow functions are used to produce more readable and succinct code.



```
reduceState = (state:State, event:Move|Tick|Restart) =>
  event instanceof Move ? {...state,
    frog: {...state.frog, pos_x: limitx(state.frog.pos_x
```

Higher Order Functions

The createObjList function is a higher order function that takes in a function that creates an object to be used for creating a list of the specific objects. This enables code customisability and reuse to create lists of different objects.

Movement Handling

The wrap() function is to make sure the planks, crocodiles and turtles wrap back around the other end of the game when moving off screen. Since the frog's movement is different from the other objects, the functions limitx() and limity() are to prevent the frog from moving off limits of the screen. This also includes when the frog is being carried by a plank, crocodile or turtle. When the carrier moves off the screen, the frog does not stay on it and will eventually fall into the river and die if the player chooses to not move the frog away.

Collision Handling

When the frog collides with any of the cars, it dies; whereas colliding with the planks, turtles and the crocodile's back makes the frog move along with the Body that it has collided with. A vel attribute is defined for each Body to keep track of its velocity. That way when the frog collides with any of the planks, turtles, or crocodile backs, it can then gain the same velocity as the Body.

State Management while Maintaining Purity

All functions aside from those in the updateView() function are only changing the state itself, there will be no side effects. The functions that are causing side effects are contained within a single updateView function. Upon restarting the game, the state will be reset back to the initial state, whereas the player progressing to level 2 will have the state set to a new one named as level2.

Game Clear and Game Over Logic

The game is cleared when the player clears levels 1 and 2 of the game. (i.e. filling the frog in all 5 goals for the first level, and then filling the frog in all 5 goals again for the second level). The game ends when the frog is hit by a car, falls into the river, or collides with the crocodile head. All of the game clear and game over logic is handled within the game state itself.

Design Rationale

Object Creation

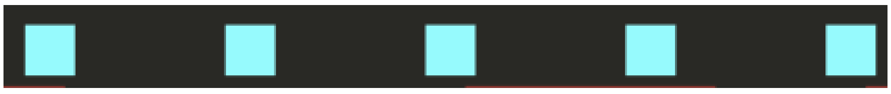
I treated all objects (frogs, cars, planks, crocodiles and turtles) as rectangles to facilitate collision handling. There are 3 types of distinct objects, namely cars and planks, crocodiles and turtles.

Safe Zone

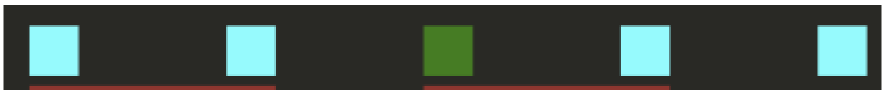
There is a total of two safe zones in the game, one where the frog spawns, and one in the middle of the game map (coloured in purple). There will be no objects to harm the frog in the safe zones.

Goal Indication

There are a total of 5 goals that the player has to fill in. When the player fills a frog into the target area (shown below), the area becomes green as an indicator that the player has already filled up the area and will not get points for filling that certain area again.



All goals unfilled



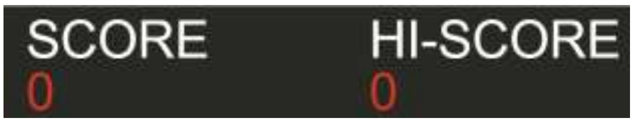
Middle goal filled

Repeated Goals

I made the game function as if the player tries to fill the frog in the same target area that has already been filled, it yields zero points. This will force the player to try to fill in other areas to gain higher points.

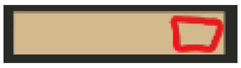
Score calculation and High Score Tracking

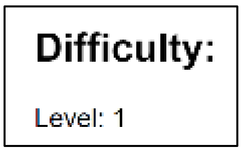
The score is calculated by every step the frog gets vertically closer to the goal (10 points) and when the frog reaches an unfilled goal (500 points). The current highest score that the player gets is tracked within the state itself, even when the player restarts the game by pressing the 'R' key, the high score is still maintained throughout the game. It will only be updated when the player obtains a higher score in the current game. The score and high score will be displayed to the player on the top of the game screen.



Increase in Difficulty when Stage 1 is Cleared

When the first stage of the game is cleared (indicated by Level: 1) where all five target areas have been filled by the player, the goals are reset and the game increases in difficulty (Level: 2). More race cars are spawned and a crocodile will be spawned in the stead of a plank in the river. The crocodile's back is safe but however, if the frog comes in contact with its head (i.e. the highlighted red part of the crocodile below), the frog dies. The turtles also behave in a way such that when the frog lands on a turtle, all turtles of the same row start moving in the opposite direction.





Difficulty:

Level: 1

Indication of the current level

Restarting the Game

The player is able to restart the game whenever they want to, let it be clearing the game, or the frog dies, or even when nothing is happening. By pressing the 'R' key, the state is reset back to the initial state and the originally created objects are removed from the view. The goals and lives are reset back to their initial colours.

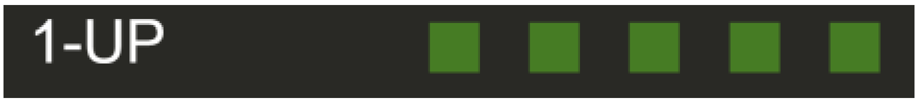
Showing Controls and Instructions to the Player

The controls are displayed next to the game to let the player know what each of the keys do to the game. There are also instructions to let the player know how to clear the game and score calculation to let the player know how to earn points.

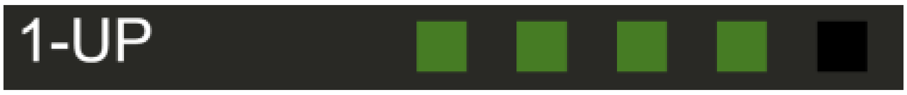
Additional Features

Multiple Lives

Having just only once chance of scoring may be quite frustrating, especially when you are close to beating your previous high score. Therefore, I gave the frog five lives (i.e. the player has five chances each game). Upon dying, the frog loses a life each time. If all five lives are used up, the game is over. There is also an indication for the player to keep track of how many lives are left in the current game (displayed as 1-UP in-game). The number of remaining lives is kept track in the state.



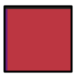
All five lives remain



When one life is lost

Female Frog

There is a female frog spawned beside the frog. The player will have a chance to “pick up” the female frog and take it to the target area with the frog. Once the female frog is “picked up”, it will follow the player wherever they go. If the player dies whilst carrying the female frog, the player has to pick up the female frog from where it was last left behind. . A successful landing in the unfilled target area with the female frog yields an additional 200 points , which encourages the player to take on the challenge to retrieve the female frog to earn more points.


The female frog