Summarisation of Code

Basic Frogger Gameplay

My frogger game has a grid-styled canvas, all units are in pixels. The canvas contains 15 rows in below manner (from top to bottom):

- 1 row for score display
- 1 row for goal (lily pad) section
- 5 rows of river
- 1 row of safe zone (grass)
- 5 rows of road
- 1 rows of safe zone (grass)

and 11 rows, where each grid is of size 50*50.

I have implemented the classic frogger game rules. All objects are differently coloured rectangles, having height of grid size and various widths. In the road section, there are two types of vehicles (cars, trucks) moving at random speed and direction (left/right), and the frog dies when colliding with any vehicles; In the river section, there are logs of different lengths moving at random speed and direction(left/right). When the frog is in the river section, it will move with any log it is on, and it only survives standing on logs while dies when fall into water. The frog is also dead if more than half of it falls outside the canvas. The frog is spawned in the middle of the bottom grass, it has to cross the road and the river section to reach any goals. Goals will be marked as visited once reached, the frog can only land on unvisited goals. Upon a successful landing of a goal, the player will score 100 points and respawn in the start position.

Extra features

I have implemented a fly feature as another way to score points. Flies are smaller black squares that are spawned anywhere in the movable area of the canvas, having random speed and direction (any). Flies can be 'eaten' by (collided with) the frog or 'vanish into thin air' (expire) after a fixed time period. The player will score 10 points for 'eating' a fly.

I have also implemented a snake feature as another risk factor. Snakes are randomly spawned on the logs, and they stay on the log for a fixed amount of time before diving back into water. The player will 'die' if colliding any snakes, which increases the difficulty to pass the river.

When the game is over, the player can restart the game with 'R' key; When the player fills all goals in the current level, he/she can progress to the next level with 'N' key. The current score

of the player is displayed on the top left, while the current level is displayed on the top right. The highest score achieved so far will also be recorded and displayed at the bottom left of the canvas, this record will persist unless the page is refreshed.

The game's difficulty increases as the level increases, namely, the number of goals required to be reached will increase, as well as the speeds of the vehicles and logs.

Design Decisions and Justifications

I have declared constants for the game to define the important overall game attributes, such as the canvas size, grid size, frog spawning position, maximum number of snakes, etc. This makes the game extensible since all modifications to important game attributes can be done in one place. It also improves the readability of the code.

I have used interfaces and types for defining all objects in the game. This includes:

- Simple (Static) objects, e.g., road
- Movable objects e.g., log
- Target objects e.g., goal

Note that movable objects and target objects **extends** simple objects. Creating different objects using appropriate types ensure that all objects only have the attributes that they are supposed to have, and they do not need to have a value for attributes that they do not use. For example, it does not make sense for road objects to have a velocity.

Curried functions are used to create objects. Since movable object and target object are subclasses of object, the function for creating an object is reused, only adding the attributes that are unique in creating respective object types. Function currying enables function creation signatures to be simplified, so the code becomes more flexible, extensible and readable. Some texts are created as objects too; however, it does not subclass the object class as its attributes do not largely overlap with the object's.

Randomness is introduced when creating flies and snakes. A class RNG for generating random numbers is created. The code is from the FRP tutorial by Tim. This is to avoid the use of `Math.random` functions, which is considered impure. Whenever a random number is required, a new RNG with seed of the current game time is created, to generate pseudorandom float numbers.

Following FRP style

The FRP style is strictly followed throughout the code. The game transforms containers without any use of loops or helper variables, it uses map, filters, etc. to ensure there is no side effects. The keyboard events are recorded as movement streams, which subscribes to the `updateView` function to achieve update of HTML elements, rather than using any event listeners. This is in accordance to the MVC model.

Game State Management

The game state is initialized, with default attributes. This includes default road, water, grass, goal and frog positions, pseudorandom vehicle and log positions, no flies, texts or score. The game status is maintained with `gameOver` attribute, which is true when the game finishes (frog dies). The `exit` attribute is for removing unwanted objects, in our case this is just the flies.

The game state is only changed in `tick` and `handleCollisions` functions. The `tick` function handles time-related change of state, for example, moving movable objects, create flies/snakes and remove expired flies/snakes. It also freezes the game state if the game is finished. The `handleCollisions` function, on the other hand, handles collision-related change of state. This includes:

- Giving the frog the velocity of the log if it is on one
- Giving the frog the position of respawn if it is on any goal
- Removing any 'eaten' flies
- Updating game status (if game is over/current level is completed)
- Creating texts according to status
- Scoring

The game's restart and finish are achieved by updating the attributes of the game state, rather than recursively calling the main function. We have a `createInitialState` function that takes in some attributes, and create the state according to those. For example, if the player progresses to the next level, the initial state for the next gameplay will be set using appropriate initial values, whereas everything but the highest score will be reset to default when the player loses a game.

Usage of Observables

The `keyObservable` function is used to map any keyboard action to game actions. There are

six keys in total that we listen to. A `gameClock` which ticks every 10 milliseconds is also used to define the game's frequency for update. All these streams are merged together, so all keyboard events are captured asynchronously, and the merged stream is passed into the reduce function. The reduce function here will account for the change of state, starting with the initial state, update the state with relevant functions based on keyboard events. Finally, the stream subscribes to `updateView` function, which updates the HTML DOM tree.

It is important to notice that we are **not** unsubscribing this stream in the game. This is because we still want to listen to keyboard events for restarting or progressing with the game even the game is finished, we will not be able to do this if the stream is unsubscribed. This means, our game will continuously listen to keyboard events once it is started.

Appendix

Figure 1: Game at Level 1

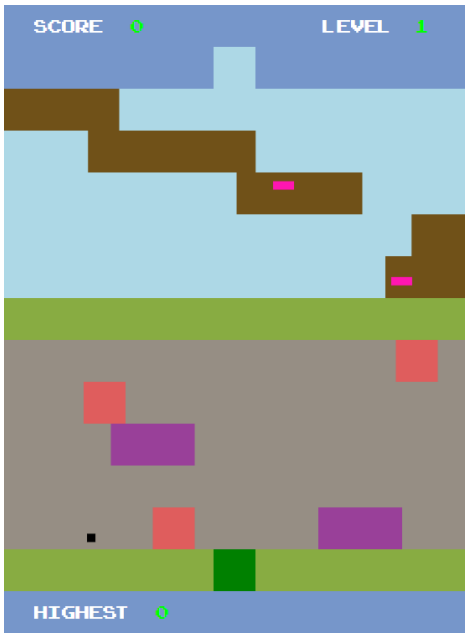

Figure 2: Game at Level 5

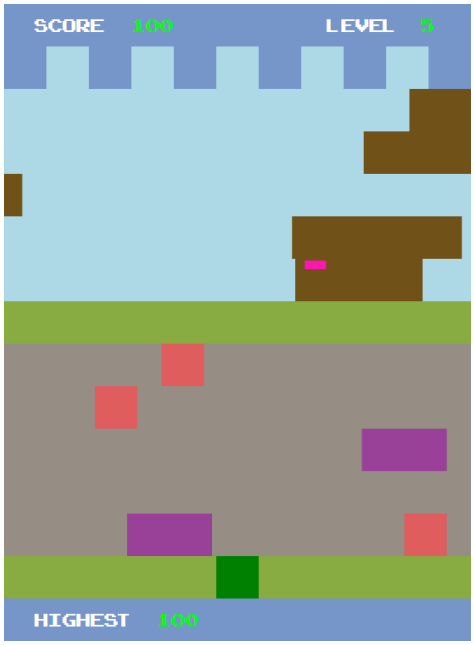

Figure 3: Game Over