Frogger

By Andreas Kardis, 31468438

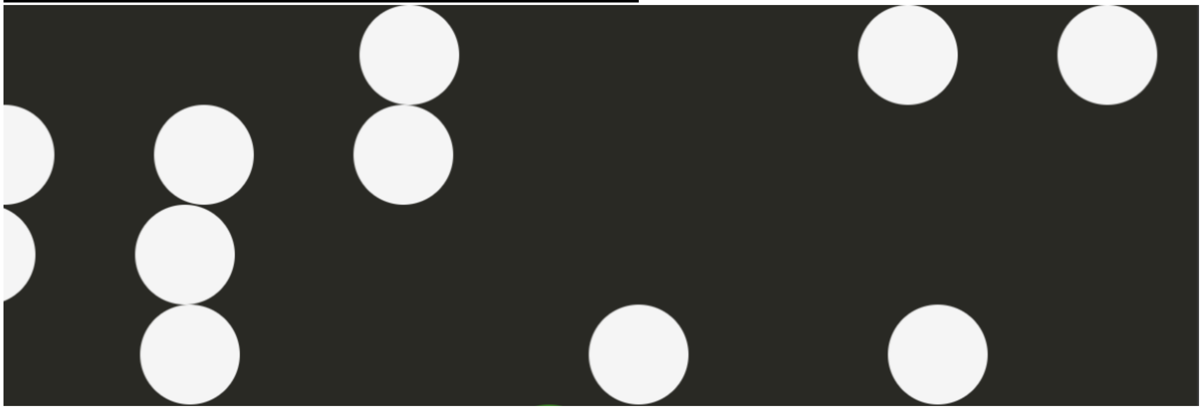
Summary of Code:

The state of the game is updated every 10ms with the use of RxJS Observable streams to handle user inputs and to update the movement of objects within Frogger asynchronously. Due to the asynchronous nature of FRP, the game continuously runs with the use of the interval function. The merge operation further allows any input of the arrow keys from the user to be utilised in scan and inputted into the Move function which allows the position properties of the frog's state to be updated. During these movements from the user or absence of movement, the position of the frog amongst other objects is also checked to see if the frog has collided with these objects. If a collision has been detected which results in the game being over, the restartGame function is called to display that the game is over as well as with the user's score. Finally, returning a new state of the game where the user can now try again. If this is not the case, the game is updated accordingly, by possibly moving the frog with the tree log or turtle it is on, or in the case where there is no collision there is no change to the state. With the results from these past steps, it is now time to subscribe the new updated state to update the game in updateView. This function does not return anything, instead its purpose is to update the graphics displayed to the viewer by rather creating them at the start of the game or updating their position throughout the game.

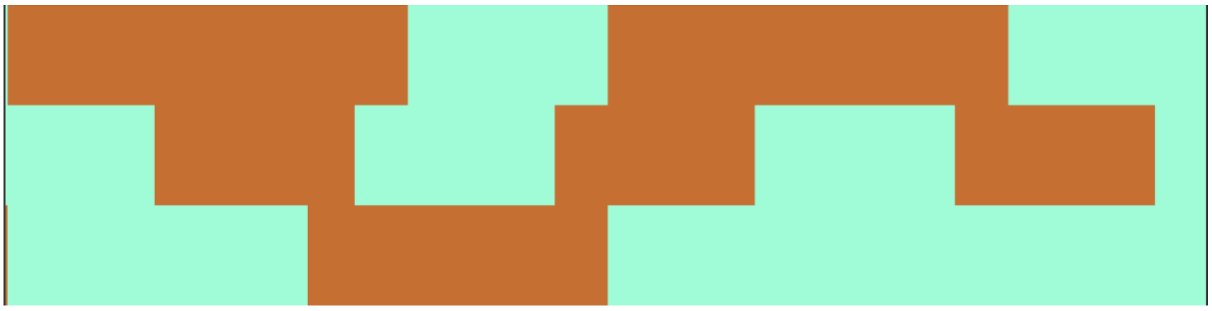
Furthermore, functional reactive programming is represented with the use of higher order functions to create instances of the cars, logs and turtles. This is seen with the map operation on an array, taking in the elements of the array and utilising it to create these instances. Further, the use of filter to rid of any user inputs that are unwanted and forEach to iterate through each car, log or turtle to make individual updates on their position in updateView. Representing how functional reactive programming allows for easier and more readable code. Collision detection, being one of two of the biggest functions, tests if there is any type of collision with the frog. As this encompasses many moving bodies, the utilisation of filter with its higher order function capabilities to iterate through each instance of cars, logs and turtles greatly improves codability and readability.

The initialState constant holds the current state of everything in the game that requires constant updating. As this is the case, it is required to keep it pure, as updating objects within the state that receive incorrect value types will result in the game crashing. To counter this, interfaces are used to define the variable types that are within an object. There are three instances of this with State, Body and logBody, allowing for different attributes to be corresponded with different updating items in the game. Defining this also allows for inputted objects into a function to be corresponded to a type, ensuring that the correct object is received. Furthermore, other inputs into functions, such as number or string, are also type matched as well as the output, to further ensure there is an output of corresponding relevancy. Therefore, increasing robustness and purity.

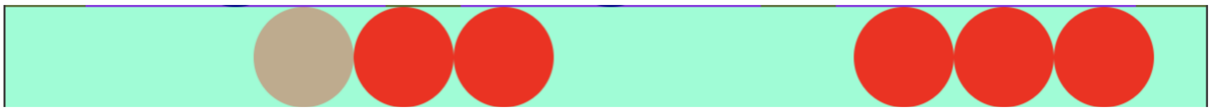
Game Features and How They Are Implemented:



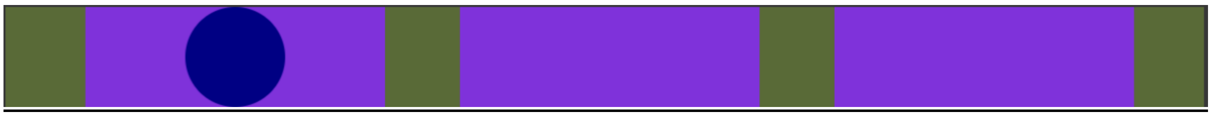
This image represents the car objects in the game as circles that move either right or left at varying speeds. This is done through each row being created individually and having their own state inside inititalState as Body objects. Their speed, distance apart and frequency are hard coded in the curried startCar functions. I have used a curried version of the function here as well as in startLog, startTurtle, startFly and startFinishLines as it allows for the inputs to be more readable. This means the operation of cars are constant through every replay of the game.



Tree log objects are represented here using the logBody instead of the Body object, as this allows for the differentiation of rectangle shapes. The logs are also setup to move at different velocities and directions which are hard coded into the game. Allowing for the game to have the same starting positions as well as directional velocity.

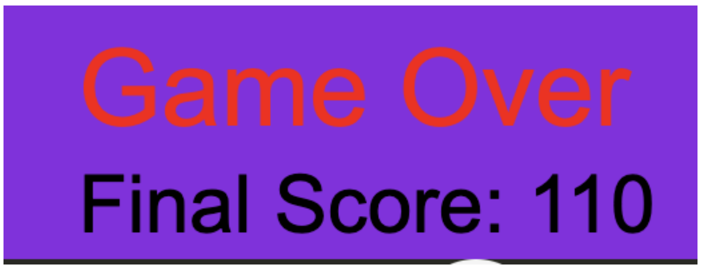


The second last row of moving objects are the turtles which travel in groups of three with the new attribute to disappear into the water. The design decision to allow them to disappear is a two-step process, being similar to the actual edition of Frogger, where the turtle firstly partially disappears to let the play anticipate when it fully disappears. This is done using a non-readonly timer variable in the Body state. With the use of Math.random in every call to update each turtle, the output with this is used to determine if the turtle should disappear. If the turtle is chosen to disappear, it is firstly reduced in opacity for 60 updates and then disappears for 100 updates before reappearing. This should approximate to 600ms and 1000ms respectfully, as the game is updated every 10ms. This could have been done asynchronously with the use of setTimeout to call a function after a set time, however, the same result is produced in the end. Furthermore, if the frog is on the turtle and it disappears, the frog also falls in the water with it due to its visibility attribute.



The last and top line of the game represents the three finish spots that the frog must get into to advance to the next level, as well as the introduction of the fly enemies. The flies appear with the same mechanic the turtles have to disappear. Once a fly appears it appears for 400 updates, approximately 4000ms. This could also be done with the use of setTimeout. Once the frog collides with a fly the game is reset and if the frog lands within the purple rectangle, the frog resets its position by updating its pos vector in its state and hides the rectangle. This attribute will also cause a collision if the user tries to land on the hidden finish spot which will restart the game.

The water is also a collision point, calculated by seeing if the frog is within the specified y-zone and if it is not colliding with the logs or turtles. Another aspect that changes with the advancement in completion of the game is a 20% increase in the velocity of all moving objects. This increases the difficulty every time ensuring that the user fails and creating variability in the scores different skill levels can achieve. Because the difficulty changes, the create functions for cars, logs and turtles takes in a single difficulty variable which is type number to ensure purity. Collision is detected for the circular objects by measuring if the length between the two objects is less than the sum of the radiuses of the frog and object. For the rectangular objects this is measured if the frog is anywhere along the length of the object.



Game Over

Final Score: 110

Upon collision with an object, resulting in the game restarting, a game over message as well as the score achieved message appears. This is done through asynchronous function handling with the use of setTimeout. The message appears for 6000ms and updates if the player has another collision during this time. The function setTimeout was used as it allowed the player to keep playing with the updated message appearing to let the player know they collided with an enemy and for it to shortly disappear without being on the screen too long.

Score: 50

The score is updated in the bottom corner to let the player know their score in real time as well as it being mostly out of the way of gameplay. Each new best vertical distance is awarded 10 points, getting to the end is awarded 100 points and finishing the game is awarded 200 points. This is done through updating the inititalState at the end of each movement in the collision function and Move function.