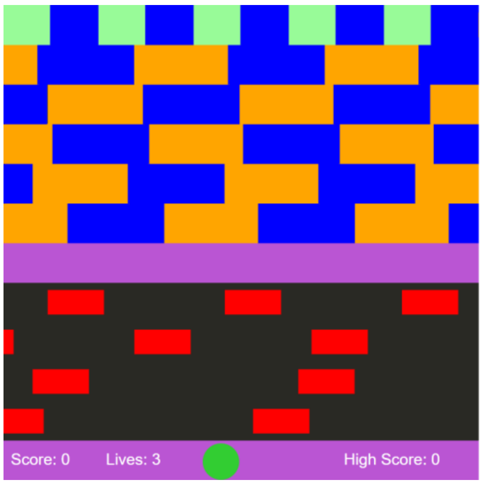
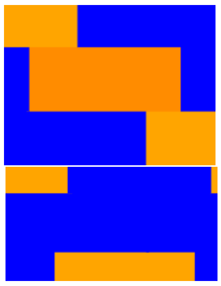
Josiah Schuller

How to play:



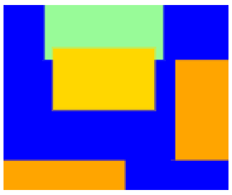
The aim is to direct Frogger (the green circle) to its favourite water lilies (the light-green squares at the top).

Frogger must avoid cars (the red rectangles) and make it over the logs (the orange rectangles) without touching the water (the blue stuff).

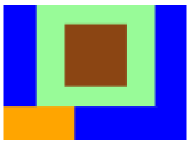


Watch out! Some of the logs submerge!
Hint: the logs which submerge become water-logged and turn dark brown before submerging!

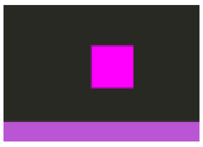
But wait - there's more!



A bird (the gold rectangle) flies around the area. Don't get swooped!



Sometimes a fly (the brown square) will appear on one of the lilies. Gobble it up for extra points!



Sometimes a flower (the pink square), floating in the breeze, swings into view. Pick it up and take it all the way to one of the lilies for extra points!

You have 3 lives! See how many points you can get!

Design report:

Observables:

The game uses observables to detect user input and to create a tick, thus following functional reactive programming techniques.

The only user input being observed is the keydown event for arrow keys. There is no need to check for keyup events because there is no need for holding keys down (this would make the game harder to play!)

A general "OBSERVE_KEY_DOWN" function is used for generating observables for each arrow key. This function has a generic type used for the return type of the function applied in the map of the observable. In my code this generic type can be an instance of a Tick or Move class, but this allows for future development if I want to add more events to the game.

All observables are then merged into a single observable (using the rxjs merge tool). Then the scan operator is used to update the game state. Then this is all subscribed by a function to update the HTML (the only impure function in the game!)

Game state:

The game state keeps track of everything going on in the game. This is the reason mutable variables are not necessary.

The state keeps an RNG class, the stage number, the number of elapsed ticks, the high score, the current score, the number of lives left, the number of end lilies reached, the width and height of the canvas, the position and size of Frogger, and a list of "objects".

The initial state is set at the start and includes all the start-of-the-game values.

"Objects":

"Objects" (from now on referred to as objects without quotation marks) are all the rectangles/squares in the game (we're not talking about the JavaScript data structure here). They can be cars, logs, water, lilies, birds, etc.

All objects are created in the initial state. I created a curried function GENERATE_ROW to generate a row of objects. I then used this curried function to create more functions for generating certain types of rows, e.g. a row of cars, a safe row, a row of logs and water, etc.

The reason I created functions to create rows is so that I do not have to create each object individually. I made the function curried so that I could input certain parameters specific for a type of object without inputting all the parameters. Thus functions could be used to generate certain types of rows.

Manipulating the state:

Manipulations to the state occur every time a value is streamed from the observables, through the reduceState function (which is applied to the observables using the scan operator).

The reduceState function takes a state and the event triggering the observable, and it returns a state. It acts purely, not mutating the inputted state or any other variables.

The reduceState function calls multiple other functions subsequently. I did this so that multiple things can be processed before the final result without mutating the state at all. It also helps to split the logic up into various sections, making the code easier to read.

Restarting the game:

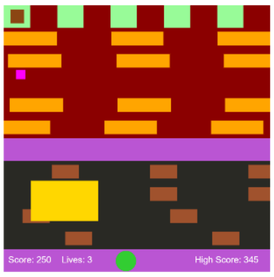
If Frogger loses all its lives, then the game restarts. This is done by simply returning the initial state with an updated high score.

Detecting collision:

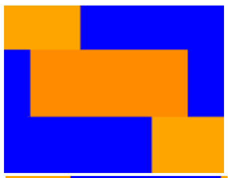
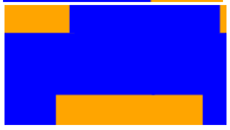
Collisions are detected within the reduceState function, or rather the resolveCollisions function which is called by the reduceState function. resolveCollisions takes a state and outputs the resulting state.

It uses the reduce method on the array of objects in the state in order to check and resolve collisions for each object individually. I used the reduce method because it purely applies collision checking to each object, then passes the resulting state as an input state into checking the next object, and so on.
For actually detecting a collision, I implemented collision strictness, which determines how strict to be with the collision. With objects, that will kill you, I made the collision a little milder so that Frogger would have to have at least a quarter of its body touching something deadly before it died. This is to make the game more enjoyable and less frustrating.

Updating the HTML:
There is only one impure function: the updateView function which is subscribed to the observables, receives a state as input and updates the HTML. This allows for the rest of the code to be completely pure.

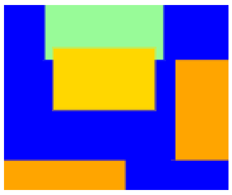


Stages:
Once Frogger has reached all water lilies (light green squares at the top), the game moves onto the next stage. At each stage, the game becomes increasingly more difficult. All objects move faster, and the bird grows larger and more dangerous. (Also, the game looks cooler! The image on the right is from stage 3!)

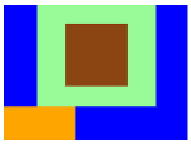



Submergible logs:
As explained briefly in the “How to play” section above, some logs submerge under water. The logs become water-logged, submerge, and then re-emerge on a timed basis. Such timed actions were achieved by checking the number of ticks elapsed and using the modulus operator (%) to repeat the action on a timed basis.
The top image shows the log becoming water-logged (darker brown), and the bottom image shows the log being submerged.

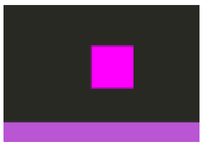
RNG:
Pseudo-randomness is used for the bird, fly and flower. I did not use the default random function because it is impure. I generated a pure RNG class from the tutorial exercises. I then fed the class a seed based off the number of ticks before the user’s first input. Because humans are unpredictable, this means my RNG class will behave differently every time. Because the RNG is generated from the user’s first input, random events (i.e. the bird, fly and flower) do not occur until after the user’s first input.



Bird:
The bird uses RNG to spawn at a random location. Its direction of movement is also set randomly and changes randomly.
Something that really annoyed me when I was playtesting the game was that sometimes the bird would be at Frogger’s spawn location, meaning when I reached a water lily or lose a life, I would lose all my lives and die because I spawn right underneath the bird. So I put a safeguard so that the bird would never fly near the spawn point.



Fly:
The fly uses RNG to spawn and disappear randomly on one of the water lilies.



Flower:
The flower uses RNG to spawn at a random location at a random time. Its direction of movement is set randomly and changes randomly, but only moves horizontally.
In order for Frogger to carry the flower, I added a “beingCarried” Boolean value to the object, which becomes true once the flower is picked up. When this Boolean value is true, whenever Frogger moves, the flower moves by the same amount.