Name: Tan Wah Yang  
Student ID: 32772246

**Design Report**

Firstly, to ensure purity of code, objects are all not created individually, but instead created as a batch. Although, it is also seen that frog can escape the frame to appear from the other side, this is an issue because it would be cheating if frog could appear from the other side to clear the game instantly. Moreover, due to certain scaling issues, elements that moves possess a somewhat weird behaviour when they reappear or disappear. That is if a log or a car fully disappears to the right, it reappears as a whole on the left. On the other hand, when a log or car wants to fully disappear to the left, it doesn’t, but slowly reappears from the right. So despite following the conventions of code purity, the functions does not run as well when fully encoded for all objects.

Now what makes the game FRP made, is that first, we are using observables for most of the part. That would mean looking at a batch of elements instead of singularly which would make it imperative. With that said, the logs’ and cars’ behaviours were determined together, so that the their actions are much more uniform and controlled. These behaviours cannot be forced onto them, and instead taught to them in a way. Moreover, this would also allow the elements to perform many different sets of actions like a normal living being, with it being done over an actual time. This makes FRP apparent as it is to model events over occurrences at discrete points in time. To add on, each of these objects being mapped to a function is necessary because the semantics to an FRP is to have continuous functions over a duration. To dive deeper, FRP is designed for data sets or types that varies over time, and that is shown by the different data sets and data types created, which are first the cars, logs and exits, then the data types of viewType, Rectangle, ObjectID, Event and Key. Both of these represent values of some sort, which once changed, should return into some action or reaction. For example, the keyboard events in the game for frog’s movement is its corresponding value. This is highly useful for this game, where it enables capturing and using these variables and values with correspondence to different applications (functions) specifically for animations or GUI.

I tried to limit side effects and maintain purity as much as possible because it makes programming much safer and neater to all coders. Functions created are small, precise, simple, safe and straightforward to reuse. However, it is hard to say when coding on handling collision, because the function’s behaviour is to identify the different interactions that the frog has with other elements. It is where the game knows when the frog hits a car, when a frog falls into a river, was sat on a log or even reach an exit point. These are all within one single function, and to ensure some form of purity, more curried functions were created within to handle the many conditions present. From simple interactions to filtering new sets of relevant elements, all to ensure that a single tick is a new initial state. One unique point here is that every time when frog hits the exit point, the exit point should be deactivated. If code is done imperatively, it’ll be impure to remove each of these boxes one by one every time it ticks. This would also make the game non-FRP because of how the state of the game is not the primary source to how the nature of the game should work. In order to fix that, the idea is that every time a condition that checks for frog is within an exit box returns true, there should also be a function that removes that particular exit. This is done by creating these exits as a batch, and if filtering the batch of exits to exits that has not been in any interaction with the frog, the new batch of exits returning to state will only have relevant exits, meaning that since an exit has been used, that element should be excluded from the stream entirely.

Moreover, to ensure purity, all functions besides the function that updates the game view, will always return the exact same type of input as output. However, these inputs are all external data, and as shown in the codes, they all take in either an object body, or the state of the game, these are all external large data. With that in mind, we also have parallelization, meaning that since there is no shared state, we can easily parallelize our calculations across multiple threads through the streams. This is shown when torusWrap function was created to wrap all objects in one go, it may not be ideal, but it is much safer to avoid any unnecessary side effects. To add on, it is also easier to test in reference to when the game is running, meaning that each function is unique to itself, and it should only do what it’s meant to do without manipulating any external state or cause dependency. Of course, with the many curried functions present, it is also meant to ensure that with purity comes with lazy evaluation.

One final thing to look out for is that working with circles and rectangles are not easy, this is because their coordinates are determined from different parts of its body. The circle is determined from the centre of itself (cx,cy), whereas the coordinates of a rectangle is determined by its top left edge (x,y). Creating conditions around these facts can be tricky but with simple math of reducing or increasing these coordinates by the radius of the circle is made easy.