Name: Matin Raj Sundara Raj
Student ID: 32124260
FIT 2102

Assignment 1: Functional Reactive Programming Report

1. Game Overview

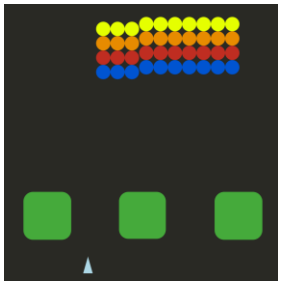

Diagram 1: Game interface

The Game implemented has similar feature sets to the classic arcade game 'Space Invaders', but has its own distinct working principles. The structure of the source code along with some of its core functions were adapted and modified from Lab Solutions and The Asteroid Game. There are a default of 4 rows with 10 Aliens each. An alien in each row account for different score from a range of [1-4]. The alien shoots bullets from random points only when the user attacks. There are 3 shields that disintegrate upon collision. Game is over when either user gets hit by an alien bullet or the alien comes in contact with the ship. If all aliens are shot, the game progresses to a new level and when the game is over, user may restart by clicking a button that appears below the canvas.

2. Design decisions & implementation

2.1 Controls

The game functions through keyboard input of left, right arrow keys and spacebar. We choose to monitor these events through the rxjs function 'fromEvent' which creates an observable stream, instead of Event Listeners which is imperative handling. Furthermore, to prevent continuous stream of keyboard events which happens when user presses a key down for a while, we filter them out by using the 'repeat'property. This allows users to play the game in a discretely controlled manner similar to the classic game. <MouseEvent> was also used to observe the click input of the user in order to restart the game, as a button has been implemented which only appears when game is over. This is done through createElement() function. As for the 'click' observable, we only take one observable input, the first one through take(1). This is to prevent the user from making multiple subscriptions to restart the game before unsubscribing (game over).

2.2 Bullets

For shipBullets, we simply create circular bodies whose movement is controlled by the Vector class. This is because at each instance of which the user presses the space bar, a new instance of shoot class is created. This allows for the creation of a new bullet body which can be concatenated into an array and display the movement through updateView() function.

As for the alienBullets, since we need pseudo-randomness, the RNG class was adapted from the Week 4 Lab. This allows us to create a random number pair of (x,y) values and pass only pure functions to the

observable operators. The alien bullet positions are mapped to the canvas and not individual as this allows for a guaranteed attack even when some alien bodies are removed. Furthermore, the Vector observable stream which is used to obtain random positions on the canvas, creates alien bullets discretely. This allows us to treat them as bodies and concatenate them to array and handle collision better instead of subscribing directly to updateView().

2.3 Aliens

The translation of the aliens across the canvas is done through the Vector Class (adapted from the Asteroid Game). Since the aliens switch directions upon reaching the vertical ends of the canvas, time was chosen as the indicator for which the aliens would move in the opposite direction. This was done because it allows for a synchronous control for the motion of the entire group of aliens through the GameClock observable. If position(x,y) was chosen instead, we would have to cater for each individual alien which would lead to an imperative style code, as an alien body can be removed from the array upon collision. The canvas size was chosen as the indicator to simulate the discrete "step down" motion of alien row instead of continuous. Implemented though moveDownLeft and moveDownRight functions of the Vector Class, in the game, each alien body will move down the y axis upon passing, x = CanvasSize/2. This can be adjusted for when the number of alien bodies increase through the use of constants.

3. Use of FRP style and State Management

The state is managed carefully by separating them into immutable states first. This is done through the state transducer 'reduceState', which is called by scan and managed overtime from its initialState instead of DOM. We merge all our primary observables together before subscribing. GetAttribute and SetAttributes are not used until the end of the observable pipeline that is, only in the updateView() function. This is because, they allow for mutable states as we can modify and obtain attributes directly. The state is updated regularly at constant intervals through the GameClock observable stream. This allows us to amend necessary changes through pure functions before updating the visuals in the impure function of updateView. Collisions are handled through the handleCollisions function which takes the state and check is the length between one body and another is less than its radius. If it is, the body is removed. All checking and modification is done through the creation of new array by concatenation or by copying the body itself and then returning it after making changes. This is to ensure that immutable objects are used and the function remains pure. Throughout the code, we have applied functional programming principles of using functions such as map, filter and reduce. This is also done though higher order functions and reusable functional elements.

Impurity:

Side effects in the updateView() function are caused by mutating objects though the set and get attributes which is necessary to allow for the update of visuals on canvas. Only one mutable, randomPos variable is used, and it is used only as a temporary container for every instance of the observable. This is because, by creating the alien bullet body discretely through concatenation of array, we are able to fire them only when the user shoots. This also allows for collision handling instead of direct subscription to updateView().