

**Name: Ahsan Zafar**

**Student ID: 29 805 805**

**Assignment 1 – Space Invaders - Report**

**Summary Of the Code**

The code implements a full game of space invaders (as stated in the specification sheet) with no extra features.

The controls are as follows:

Left arrow: Ship moves left  
Right arrow: Ship moves right  
Space: Ship shoots bullets  
R key: Game restarts

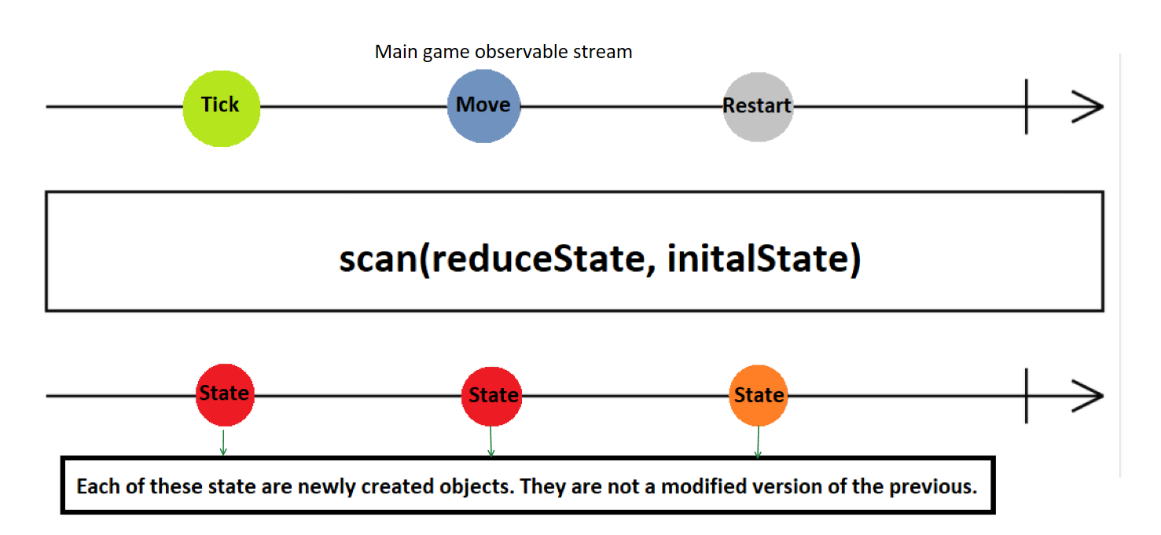
The backbone of the game is the main observable stream which is created by merging a bunch of observable streams. These observable streams are listening for keyboard events which trigger a specified action. The main stream is also merged with interval streams that trigger an action every specified interval.

The scan function is then applied to main stream. All the magic of the game commences in the *reduceState( )* function which is passed to scan. Depending on the action that was emitted from the main observable, the *reduceState()* the appropriate functions which return a new state object with the required changes. Upon each state object that it emitted after the scan, the *upateView( )* function called to perform the required side-effects to the view (i.e., the HTML that will be rendered on the browser).

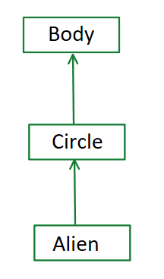
Click [here](https://streamable.com/98cpg4) to watch how the collisions take place in action.  
Click [here](https://streamable.com/ph4sji) to watch how a game restarts.   
Click [here](https://streamable.com/71bkdr) to watch how the levels are increased.  
Click [here](https://streamable.com/jcxuz9) to watch how a player can lose when the aliens reach the bottom.

**Managing State throughout game**

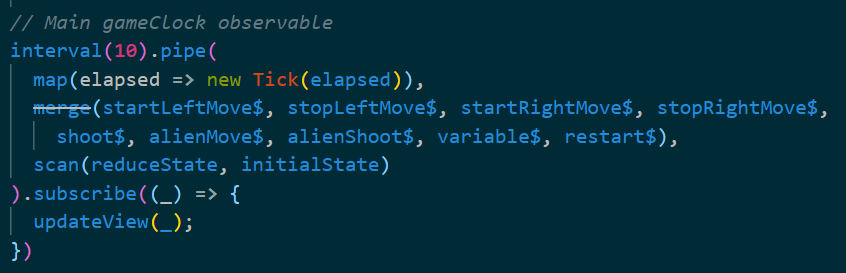
• State Management

* The way that state is preserved throughout the game is done in a pure way. That is, the state is declared as a Readonly object; it is never modified.
* As stated on RXJS website, the “scan operator is used for managing and encapsulating state”. This sounds exactly like what we need to manage the state of the game. By passing in a pure function to scan, (i.e., a function that accept as its input a state object, and returns a newly created State object with whatever change are required rather than modifying the attributes values), we ensure that our scan function is pure and that it has no side effects.
* If the function passed to scan mutated the State object rather than creating a new object, this would be considered impure since it is causing a side-effect. Side effects are not obviously visible, so when they cause a bug/error, it can potentially be extremely difficult to trace the root case. Therefore, using pure functions allows us to create more robust code.
* 

**Design Decisions, Justifications and FRP.**

**•** Representing Objects in the game

* The image on the right displays how objects are being represented in the game. *Circle* is an interface extending the *Body* interface, and *Aliens* extends the *Circle* interface.
* This game comprises of:  
   • Body Objects: Ship, Shield  
   • Circle Objects: Ship bullets, Alien bullets  
   • Alien objects: Aliens.
* The main benefit of using interfaces in this case because of polymorphism – we can inherit the attributes of the parent without having to duplicate code. Additionally, they enforce a certain type of structure which allow our code to be more robust and make it easy to extend.

• The game contains a single observable stream as the main game clock, which is a merge of multiple different observables streams.

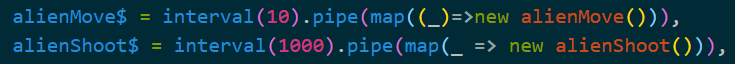
* *Interval(10)* is brought to the top of the pipe, and this allows the game to have a very smooth and usable game play, since we are updating the state every 10 milliseconds.
* This also allows to add new observable streams into our main stream very easily, by initialising the stream, and then merging it along to the main observable stream. This will make the code base easy to maintain, and also when extending to implement new features.
* Additionally, a key feature of this is that if, for example, the right key pressed for a long time (to move the ship to the right), the ship will continue moving to the right with no delay at all.

• MVC Architecture

* All Our pieces of the game represents an MVC architecture.
* The model consists of the state, which is simply a Readonly object that stores information about all the objects within the game. The *updateView( )* function updates our view, which is the HTML that is rendered on our browser. This is the only part where any code with side-effects is inserted. The Controller consists of all our input streams which are made up of observables (keyboard events), and also interval Observables which trigger a specific action.
* The main advantage of MVC architecture is *separation of concerns.* Since the application Model, Control and View are divided, it can be easily maintained because of the separation of concerns.

• Random Number Generation

* Randomness is used to decide which alien is to shoot at any given point. (Note: Only the bottom aliens in each column may shoot). If we were to use the random function in the JS Math Library, this would be considered impure since it violates the property of “produces the same result for the same input”. For example, calling Math. random( ) will yield a different result for the same input.
* To keep the purity of our code intact, we create a RNG class which creates consistent random numbers given a seed.
* To keep the game interesting and prevent the aliens from shooting in the exact same pattern, the seed provided to the RNG class uses the **impure** function Math.random( ). This is the only place throughout the code (other than the subscribe call) where an impure method is used.

• Alien shooting and alien movement is dependent on their respective observable streams.

* For alien movement, an observable stream is created which emits an *alienMove( )* instance every 10 milliseconds.
* For alien shooting, an observable stream is created which emits an *alienShoot( )* instance every 1000 milliseconds.
* Observables provide support for passing messages between parts of an application. For the game, this means that the alien movement and alien shooting will be done exactly after 10 milliseconds and 1000 milliseconds respectively, and all we need to do is “react” when the observable emits. We “react” in the function passed into scan.
* Other input streams are added which listen for a particular type of mouse event.

• Handling Collisions  
Click [here](https://streamable.com/98cpg4) to see collisions (and smooth gameplay) in action.

* All collisions are handled within the *handeCollisions( )* function.
* We find that the logic behind detecting a collision between 2 separate bodies is very similar. To show off our Functional Reactive Programming knowledge that we have learnt this semester, we can apply this knowledge to create small immutable functions, then combine these small and easy-to-understand functions to create complex functions.
* We need to check whether a collision between the following two bodies has taken place:  
  • Alien bullet and Shields (Not a Game over condition)  
  • Ship bullet and Shield (Not a Game over condition)  
  • Ship bullet and Aliens (Not a Game over condition)  
  • Ship and Alien (Game over condition)  
  • Alien bullet and ship (Game over condition)
* I do not want to start explaining “*how*” the collisions are detected, since the report is meant to explain the *“why”* rather than the *“how”.* Essentially, few small, pure, curried local functions are created. Since these functions are small and pure, they are very easy to read, understand and test.
* One benefit of the currying is easy partial application of a function. Consider the type of a collision detection function. It has the following type, and uses *f*  as the predicate to detect a collision between body1 and body2.

This function can be partially applied by specifying f, and this way we get a new function which only accepts , and is ready to be used. This can be re-used now, and will make our code easier to refactor.

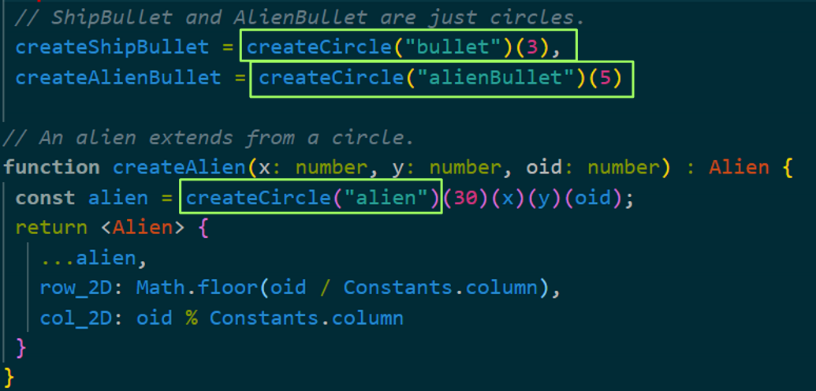
* A more complex function can now be made by using the local functions. This function will accept 2 arrays of bodies (namely, allBody1 and allBody2), and return an object containing the collided bodies. This function can be re-used to detect all collisions listed above. The return result is then used to create a new State to represent the changes.

• Using currying to create bullets and aliens.

* Since ship bullets, alien bullets and aliens are all of type Circle (see “Representing Objects in the game” section), we can use currying functions to our advantage.
* For any given circle, the viewType and radius are all pre-determined (i.e., the viewType for an alien bullet is always “alienBullet”, the viewType for a ship bullet is always “shipBullet”. Similarly, their respective radius are also known). The only properties that vary between different circles is their x position, y position and object ID. Therefore, we can use partial application by passing in the viewType and radius to our curried function, which will return a new function that is just waiting for an x,y and objectID to be passed in.
* Now, since alien extends from circle, it inherits all its properties. The additional properties in an alien is that it has a 2D

x-coordinate and a 2D y-coordinate. Therefore, we can re-use the partially applied function to create a circle, then add the new properties so that the new objects holds true to its interface.

* This truly allows to make the most of our functions by preventing any duplication of code.



• Game Decisions

* Shields are made up of small rectangle objects. Upon collision with either a ship bullet or alien bullet, the collided rectangle object will disappear. Therefore, a shield will deteriorate upon collision. This is done to achieve closest possible game play to the original space invaders game.
* Aliens move across and down the canvas. Upon entering a new level, the aliens will start to move faster across the screen, whilst their vertical speed remains constant.
* Once the aliens reach the bottom of the canvas, this will end the game since it is considered a player loss.
* Only the bottom alien in a column is eligible to shoot. From all the bottom aliens, a random alien is chosen to shoot per tick.

• UpdateView is the only place with side-effects.

* The update view function which is being used in the subscribe call of the main observable stream is the only place where any impure code will be present.
* By ensuring that all code outside out update view function is pure, we assure that it is not the source of any but the most basic display issues, which we expect to identify with a local check of this one function.