FIT2102 Assignment 1 Report Submission by Khong Lap Hoe 32114818

Code and game highlights

In my **`tick()`** function, **different states are maintained** by returning different states based on different conditions that are presented by the input state. This is handled for going to new levels, and ending the game. It is also used to animate the alien movement through the **`animateAliens()`** function.

When playing the game, the aliens moves faster as more aliens are killed. As aliens on the left are killed, the aliens on the right will move more towards the left as well.

When the ship wraps around both end of the screen, there is a "flashing" kind of animation, that "smoothly" wraps the ship and I have kept this because I found it cool.

`Static types` are the initial positioning of the static aliens and static shields before it starts moving. It is called static to show that it is not meant to move at the first instance of the start of the game.

Design of Functional and Reactive Code

The functional programming paradigm ensures that there is no mutability of any global state, and this helps us to form a **Model-View-Controller architecture** that ensures functional purity. This is heavily enforced within the **`spaceinvaders.ts`** by **using pure observable streams** to handle **reactiveness** and writing functions that return new state objects instead of changing an input state in-place. In essence, there is no state in our program. It's merely one input state serving as a guide on how to produce the new state.

In our main subscription call of **`line 513,`** an observable operator called **`Scan`** is used to transform the **`initialState`** into a new state, that is determined by our **`reduceState`** function, not altering the global variable **`initialState`** itself. The **`reduceState`** function returns a different state depending on the type of input instance, generated by the asynchronous behavior of user input or interval steams, such as a **`keydown`** or **`keyup`** event. For each event, the stream is transformed into

objects of different event classes, such as **`Tick`** or **`Motion.`** This is needed for **`instanceof`** pattern matching within the **`reduceState`** function, like in FRP Asteroids. Different states are needed as there are different inputs from the user, and there are also different game conditions.

After reducing our state into one final state (for a particular **`gameTime`** cycle), we finally call **`updateView`** in our subscribe call, which is the only impure function, and is the bread and butter of how the user can see the change according to an input state. This is a high-level overview of how the program looks like, showing that there is really, **no global state to mutate in the first place.** Do note that a **`State`** is a **`type`** with **`readonly`** properties.

How is purity maintained, mutability and imperative style avoided?

The **`readonly`** declaration of any JavaScript container, such as **`ReadonlyArray`** or **`Readonly<>`**, plays an important role to ensure no mutability for any object, as the values are read-only. This is extensively used in my code where it is declared practically everywhere for any Type/Arrays/instance variables used in the code. For example, in **`line 132`**, properties like **`exit`** and **`shields`** are **`ReadonlyArrays.`**

Arrays functions are also extensively used in my code, like **`map`**, **`reduce`** and **`filter`** for **`ReadonlyArrays`**. Array functions are referentially transparent, in which there are 0 side effects, also ensuring that a *new* Array with performed changes are returned. Using such functions can also avoid using for-loops, and local mutability which all eventually leads to imperative style of programming.

When accessing global values (the only ones are just constants at the top of the code), it is declared **`as consts`** meaning there won't be chances where it is accidentally mutated. As mentioned, typescript type annotations also ensure we receive and return the correct types to further ensure immutability.

Curried functions are also used in order to partially generate static objects at the start of every level or during game restart. This partial evaluation of the **`createStatic()`** function allows for the usage of both **`createStaticAliens()`** and **`createStaticShields()`** anywhere else in the code, although they do the same thing. This is useful during level up's and restarting the game.