Matthew Siegenthaler 31485111

Assignment 1: Functional Reactive Programming

Summary

The task was an implementation of the retro arcade game Space Invaders using Functional Reactive Programming (FPR) techniques in TypeScript whilst using RxJS observables for asynchronous inputs and events.

Design Decisions

The fundamental design centred around the expected requirements which include:

- A playable ship controlled by keyboard which the user may fire bullets.
- Rows of aliens that fire bullets at the user.
- When hit, the user will die, ending the game. The user may then restart the game.
- A scoreboard displaying the player’s score.
- Shields that may be damaged due to bullets.
- Additional levels when all enemies are killed.

A keyboard approach was decided for the controls of the game as this would allow for more precise positioning of the player’s ship. In addition, the player is limited to one projectile at a time to prevent against spam use. This projectile is stored together in an array with alien projectiles which may have several projectiles onscreen.

Aliens also slowly accelerate during the level, providing a sense of urgency for killing all the aliens quickly. Upon completion of a level, the acceleration of aliens increases. This provides a layer of increasing difficulty after completion of levels as the user has less time to shoot all aliens. All onscreen aliens are all capable to shoot projectiles as it was difficult to determine

Unlike the asteroids example which used the centre body and radius for the shape of bodies, I opted to use the top-left corner with width and height as this was more suitable for the rectangular objects which allowed me to just calculate the opposing corner for its shape.

New Features

Additional features included as part of the implementation include

- A high score feature that utilises local storage.
- Player may also shoot enemy projectiles to destroy them.
- The bonus ufos that occasionally flies across the screen that gives bonus points

Implementation

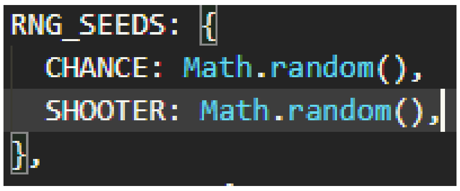
Objects

All interactable and movable are stored as a Body type which stores identification information of an object as well as positional and movement stored in vectors. This was used also for stationary objects such as shields and holes for positional and identifying information for collisions. In addition, a single controller body is used as a reference marker to control the position of the aliens that could not be destroyed by the player. This was because they tended to accelerate to prevent them moving offscreen. This would check if it had passed a threshold boundary and shift all remaining aliens back within the designated boundaries.

Purity

Like Dwyer's FPR Asteroids example (n.d.), the Space Invaders implementation follows a Model-View-Controller architecture where user inputs update the state of the game, the new state then updates the display so the user may react to it. As such, it is vital we know exactly how the state is manipulated, which is achieved through functional programming practises such as function purity.

By passing around the state as an argument of our functions, Function purity is maintained by using impure functions only during the initialisation of the state or during the View stage.



```
RNG_SEEDS: {
  CHANCE: Math.random(),
  SHOOTER: Math.random(),
},
```

The function *Math.random()* is used initially to generate a seed for a pseudo-random number generator shown by Tim Dwyer (2021). This generator is then used as part of a stream for random events, as well as an attribute of the state used to achieve synchronous random numbers *(for generating a random index)*. Since this impure method is immediately assigned to a constant during the initial state, we can be certain it will not mutate whilst the game is running ensuring no side effects.

On the other hand, we have isolated our impure functions to the *updateView* function where we manipulate the SVG canvas the user views the game. This ensures we known exactly where we manipulate our html for our interface and where our side effects occur.

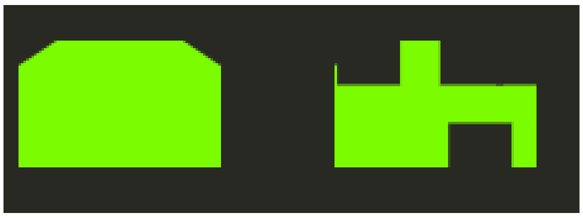
User inputs

The state is manipulated through events emitted by the games main observable stream which includes user inputs such as moving, shooting, and restarts, the game's clock which consistently updates the state of the game, and random events occurring such as an alien shooting. This is merged as one stream. Once the state has been updated from events, the *updateView* in the *subscribe* method is called.

Restarting & level proceeding

Restarting and level resetting is implemented in a pure manner through use of a renew state function. This returns a state like the initial state except retaining the score and increasing alien acceleration if it is the next level. Items from the previous round such as bullets, holes, and ufos are put in the exit array where they will be removed from the canvas in the *updateView* function.

Shield disintegration



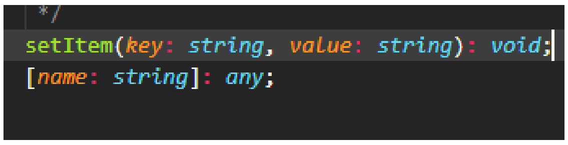
Shield disintegration was achieved with a by placing rectangles the same colour as the svg canvas. These were stored as bodies in the state and would be placed when a bullet collided a shield whilst not in a hole. A hole would allow a bullet to pass through if the bullet collided with it.

Curried functions

Curried functions served to promote use of generic functions to reuse previously existing pure functions such as comparison functions for filtering arrays, function *cut* using *elem* to compare *id* so items may be removed from an array, and not for inversion of functions with Boolean outputs.

High score local storage

To implement a persistent high score, the browsers local storage was used to access and store the high score. Since the *setItem* method for *local storage* is an impure function, The setItem mutates the value in local storage.



```
*/
setItem(key: string, value: string): void;
[name: string]: any;
```

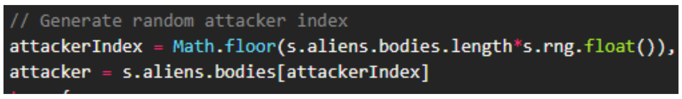
To combat this, Local storage is accessed only once at the beginning to get the previous high score and is assigned as a constant PREV_HI_SCORE.

This is stored in the initial state as the attribute hiScore. This is updated when the player beats the previous high score, when returning a new state object. Mutation of local storage occurs only in the impure function *updateView* where it sets the hiScore of the state in local storage.

Random events

The mystery ufo event as well as alien shooting behaviour is controlled by a pseudo-random number stream as shown by Tim Dwyer in Pi Approximations (2021). A floating-point constant containing the chance of the event occurring is compared against the values emitted by the stream. This stream is taking an initial seed *(A random float generated during the initial state)* and uses it to generate a "random" number *(this number is manipulated with set operations to appear random).* This generated number is then used as the next seed to generate the next random number, providing the illusion of randomness whilst remaining pure as the values always map to the same value. Once a generated value from the stream meets the threshold of the event chance, an event is emitted to the main game stream where it is handled by the state transducer to perform the required operation for the event.

To determine which alien will shoot, an RNG object is also assigned to the state. This is called to get a decimal which is multiplied by the number of aliens in the alien's array and rounded down to get an index. The random number will then used as a seed to generate a new RNG class object using the next method which must be reassigned to the state, otherwise the same value will be emitted.



```
// Generate random attacker index
attackerIndex = Math.floor(s.aliens.bodies.length*s.rng.float()),
attacker = s.aliens.bodies[attackerIndex]
```

References

General structure follows Tim Dwyer's implementation of asteroids.

- Dwyer, T. (n.d.). FRP Asteroids. Retrieved from https://tgdwyer.github.io/asteroids/
- Dwyer, T [Tim Dwyer]. (2021). PiApproximationsFRPSolution [Video file]. Retrieved from https://www.youtube.com/watch?v=RD9v9XHA4x4

Alien sprites

- SVG Repo. (n.d.). Space Invaders SVG Vector [svg]. Retrieved from https://www.svgrepo.com/svg/275959/space-invaders
- SVG SILH. (n.d.). spaceship space ufo alien [svg]. Retrieved from https://svgsilh.com/f44336/tag/ufo-1.html