Contents

[Explaining Code: 1](#_Toc82137824)

[Lines (14 -59) 1](#_Toc82137825)

[play(level: number, player: State = null): void 1](#_Toc82137826)

[createEnemy = (oid:number)=> (time:number)=> (radius:number): Enemy 1](#_Toc82137827)

[createBulletFromEnemy(s: State, randEnemy: number): ObjectState 1](#_Toc82137828)

[Lines 153-172 (Class Declarations) 2](#_Toc82137829)

[Lines 177-237 Observables and Changing elements of gameState 2](#_Toc82137830)

[Usable Gameplay: 3](#_Toc82137831)

## Explaining Code:

Space invaders function is called as soon as html file loads. This is done at the end of the code. Similar style to python if “\_\_name\_\_” == “\_\_main\_\_”. This space invaders function encapsulates the entire game and includes all the games code but does **not start** the game, the function that starts the game will be mentioned later.

### Lines (14 -59)

This part of the code is very important, this is where we define the object and state types in the game.

#### Type ObjectState

ObjectState is the state of ingame objects, such as bullets or ship or enemies. It is ReadOnly this is because in the code functions return this object and when an object is returned by a function, elements in the function can be changed so we make it ReadOnly to stop changes unwanted to these objects.

#### Type State

State is the state of the GAME itself, and points to the ALL objects in the game, so they aren’t lost, only objects in the gameState are outputted to the html file, so if an object is created it must be included in the state type. State is Readonly for the same reasons as ObjectState

### play(level: number, player: State = null): void

This is a pure function which starts the game, it takes input of the level number, which in turn makes things harder like enemy bullet speed and movement and the amount of enemies in the game

Whenever a next level is to be created this function is called. It also takes the state of the player as the second argument, and if a player isn’t passed it assumes that the player score needs to be reset. This mechanism was created for multiple lives, for whenever you die, your player stats would be reset but you would still be on the current level, but for this assignment the requirement was that the game would end when the bullet hit the ship so essentially you always had one life, so I kept the possible functionality because if this were to be an actual game it would be good in the future. And its always good practice to make code as easy to extend as possible.

### createEnemy = (oid:number)=> (time:number)=> (radius:number): Enemy

This is a pure function and was initially created to have multiple sized enemies, but due to time constraints I kept this implementation in case for further development of code. The function creates an enemy and returns it as an Enemy Object which extends ObjectState

### createBulletFromEnemy(s: State, randEnemy: number): ObjectState

This is a pure function which returns an ObjectState with bullets from a random enemy ship. The way I made this random and still have the same output for the same inputs is by passing the random value through the parameter, so the random part is done outside the function, so the output will always be the same for the same inputs. To make 100% sure of that I use Math.Ceil to make sure the randEnemy isn’t a decimal value and breaks the code.

## Lines 153-172 (Class Declarations)

Class declarations are solely used to identify the observables and what they do, this is good if we have multiple observables doing the same actions such as moving your mouse and using arrows to move.

Another reason the classes are used is for the value in the constructor, it introduces encapsulation and prevents changing the value when its being passed.

The alternate direction class is different, has a number parameter which is used a way to alternate between 1 and -1

## Lines 177-237 Observables and Changing elements of gameState

Simple way of describing it. Observables output classes which are inputted into reduceState with gameState which change the objects for each respective class in the gameState object, then after updateView is called and the elements in the gameState are updated in the html document

**keyboardMoveInput** takes in a ‘keydown’ event and is filtered to arrowLeft and arrowRight **AND** if the input is not repeated. We do this because we make out own repeating input with ‘interval’, I make this interval 5 because it somehow helped the bug mentioned, the bug isn’t fixed, but I made the ship fast enough so the bug wouldn’t show. The mergeMap is to merge the input of the keydown event and the keyup event into an array. This is done so we can access the keydown movement and see if the keyup is the same key as the keydown event. Now, The interval keeps outputting observables every 5 milliseconds until a key up event is found of the **SAME KEY** then the interval stops and the entire array is then mapped to just the initial event value which is the keydown event. Then the event is mapped to a Move instance and the parameter depends on if it is a Left or a right keyInput this makes the result observable a Move Object.

keyboardShootInput is the same as keyMoveInput but with a faster interval and it returns a Shoot instance instead

mouseMoveInput is different because an observable is everytime the mouse is moved, so all we need to filter here is if the mouse is in the svg or not. So we call ‘mouseInSVG’ to deal with that. The problem with this now is that we cant use the Move class because that takes inputs on how much to move. And when I did that with mouse using the attribute ‘movementX’ it became very glitchy between the mouse going in and out of the svg, so instead I created a new class which took the x and y of the mouse and replaced the ship x with the mouse x. Everything essentially is the same, this Observable stream outputs ChangePosition instances

mouseShoot is the same as mouseMoveInput but takes a ‘mousedown’ mouseEvent and returns a shoot class so the observable stream outputs a Shoot class

tickInterval is the interval created for the gameTick, this interval is dependent on the level, so the higher the level the harder it gets. To make the game not impossible to play at harder levels, I made the decrease a logarithmic function with a limit of 4 (asymptote of log function) being removed from the initial value which is 10. This observable stream returns a Tick class with the parameter of the interval event value as interval returns an increasing number (by 1) every interval

enemyBullets is an interval that returns an EnemyShoot class instance as the observable in the stream. This interval also has a logarithmic function to decrease the initial value of 1000, making the amount of times the enemy shoots to be faster.

reduceState is the function which decides what to do for each of these classes that the observables are made of, reduceState replaces the gameState object with a modified version of the gameState object depending on each input. Made sure whenever an object is created to increment objCount

subscription then merges everything into one stream and applies the reducestate function which is what the input does to the objects in the game and then the stream is subscribed to ‘updateview’ which maps all the objects onto the html svg and handles when the game finishes or continues to a next level.

To handle collisions you essentially just subtract one objects vector to another objects vector and if its smaller than the radius’ of each object combined it means there is a collision. When something collided with the ship I had to hardcode an offset to make sure the hitbox is decent, essentially once we have filtered the objects we want to remove, we have a ‘not’ function and filter the array for each respective object with **NOT** the bullets that collided. Made sure to **update the objCount** because that is important with the ids in the html doc

Tick updates the positions of eachObject if they require moving, object that would require this is like bullets or enemies. It also makes sure if bullets expire or run out of their life time.

Followed FRP style by maintaining pure functions (making sure attributes used in functions are passed in through parameters and most importantly for the same input has the same output), Immutable data, using Observables. This is done with class encapsulation, function encapsulation, Classes and readOnly types.

Designed enemies to be in a specific grid so its sometimes uneven block and sometimes even blocks

Examples of reusing code, essentially every function but here are some examples:

* (Line 326) -> createText
* (Line 348) creating an array of ObjectState and then using foreach twice
* (Line 395) clearGame function
* (Line 49)Extending objectState with enemy

EXTRA STUFF

## Usable Gameplay:

One Bullet at a time (like the actual game)

Made the game have a logic where the ship can only shoot one bullet at a time, just like the actual space invaders.

Restart

The game restarts if you click on the game after ‘GameOver’

Next Level

Next level ‘animation’ (not really the countdown)

Level Progression difficulty increases over time (faster enemies, faster bullet speeds, faster enemy movement)