FIT2102 Programming Paradigms 2021

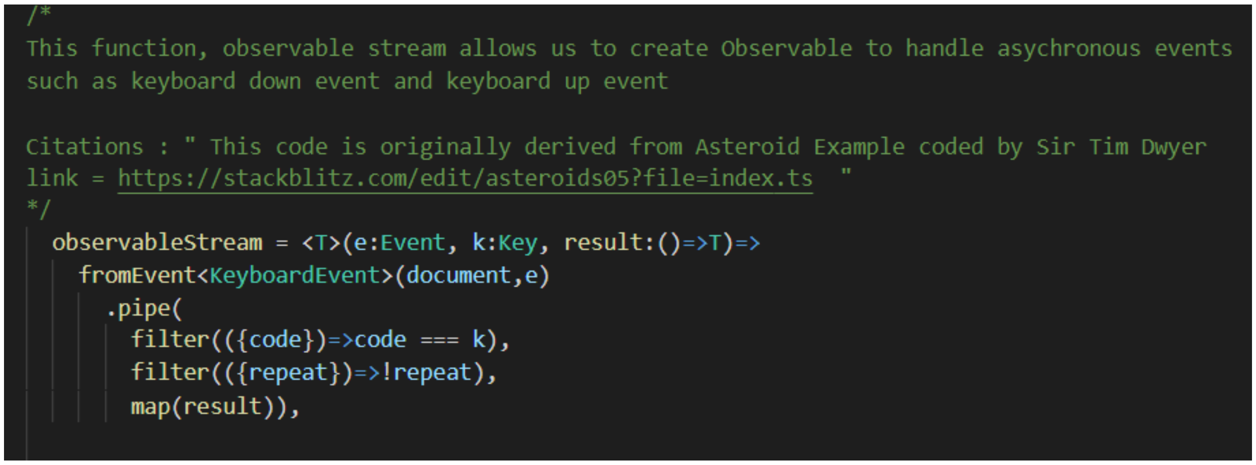
Assignment 1: Functional Reactive Programming Reports

Name : Seng Wei Han

ID NO : 32229070

Throughout this assignments , I have learnt to create a classic arcade game using

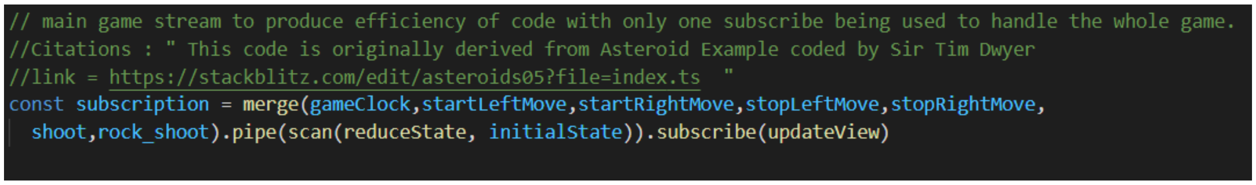
Functional Reactive programming (FRP) techniques. The first techniques includes creating a function called observableStream which is shown below :



```
/*
This function, observable stream allows us to create Observable to handle asychronous events
such as keyboard down event and keyboard up event

Citations : " This code is originally derived from Asteroid Example coded by Sir Tim Dwyer
link = https://stackblitz.com/edit/asteroids05?file=index.ts  "
*/
  observableStream = <T>(e:Event, k:Key, result:()=>T)=>
    fromEvent<KeyboardEvent>(document,e)
      .pipe(
        filter(({code})=>code === k),
        filter(({repeat})=>!repeat),
        map(result)),
```

This function is reusable throughout the code and it also allows us to create multiple observable stream for each unique event occurs in the game such as moving the ship or allow the ship to shoot bullets to the rocks and many more.

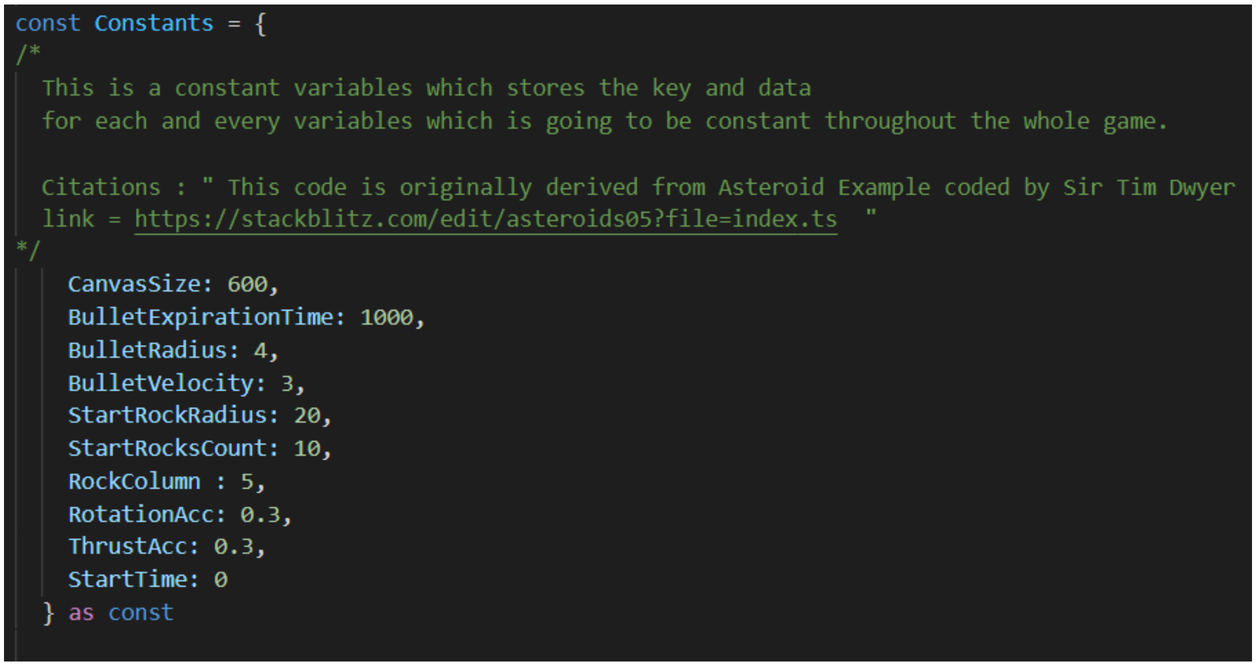
Then I have also created multiple small observable streams to handle each event such as shooting, moving of ships and stopping of ships. However, the main observable in my program is called the subscription as shown below:



```
// main game stream to produce efficiency of code with only one subscribe being used to handle the whole game.
//Citations : " This code is originally derived from Asteroid Example coded by Sir Tim Dwyer
//link = https://stackblitz.com/edit/asteroids05?file=index.ts  "
const subscription = merge(gameClock,startLeftMove,startRightMove,stopLeftMove,stopRightMove,
  shoot,rock_shoot).pipe(scan(reduceState, initialState)).subscribe(updateView)
```

This main observable will merge all of the observable stream previously created into one observable streams and uses the scan function from rxjs to accumulate the state using the reduceState function. Such implementation allow us to only use one subscribe method to handle all the asynchronous events which is very efficient I would say. By doing this way, I have already achieved 50 percents

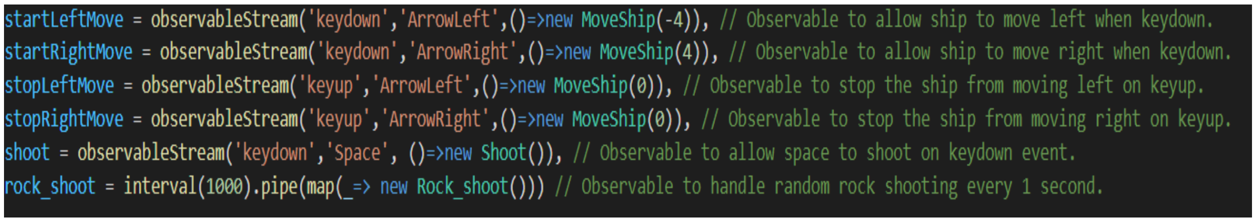
of Functional Reactive programming (FRP) techniques. Once I have handled the observable part, I move on to create immutable types such as the constants which will remain unchanged throughout the whole game as shown below:



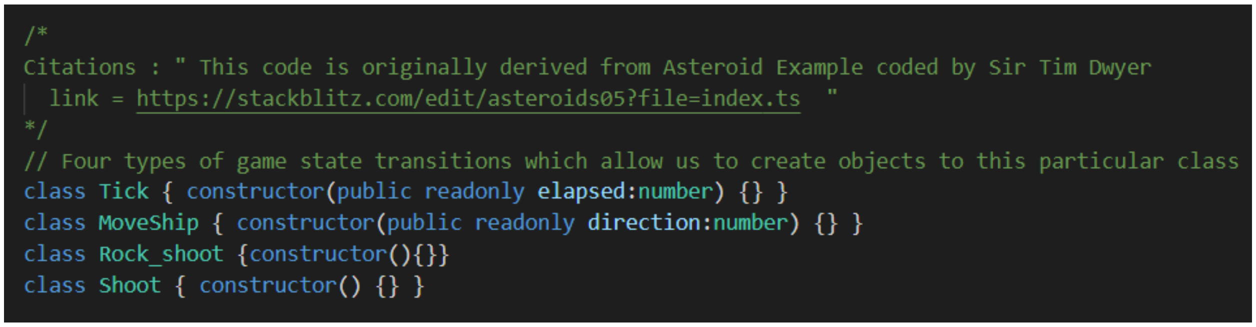
```
const Constants = {
/*
  This is a constant variables which stores the key and data
  for each and every variables which is going to be constant throughout the whole game.

  Citations : " This code is originally derived from Asteroid Example coded by Sir Tim Dwyer
  link = https://stackblitz.com/edit/asteroids05?file=index.ts  "
*/
    CanvasSize: 600,
    BulletExpirationTime: 1000,
    BulletRadius: 4,
    BulletVelocity: 3,
    StartRockRadius: 20,
    StartRocksCount: 10,
    RockColumn : 5,
    RotationAcc: 0.3,
    ThrustAcc: 0.3,
    StartTime: 0
  } as const
```

I have also created other immutable type such as state , body , initial state and many more. By using this implementation , we maintain the immutability of all the data in the game which is consistent to the implementation of a functional style which do not allow the use of ‘let’ keyword and instead promote the use of ‘const’ keyword. I know some of you might question that since most of the state are immutable, how are we going to change them without causing any side effect on them. The answer to this question is that , we will be using the function reduceState to allow transformation to the state without causing any mutation.

The idea behind this is that , for every event triggered by the user such as pressing the arrow left, the observable will detect it and create a new object of that particular class that was initially predefined which is shown below :

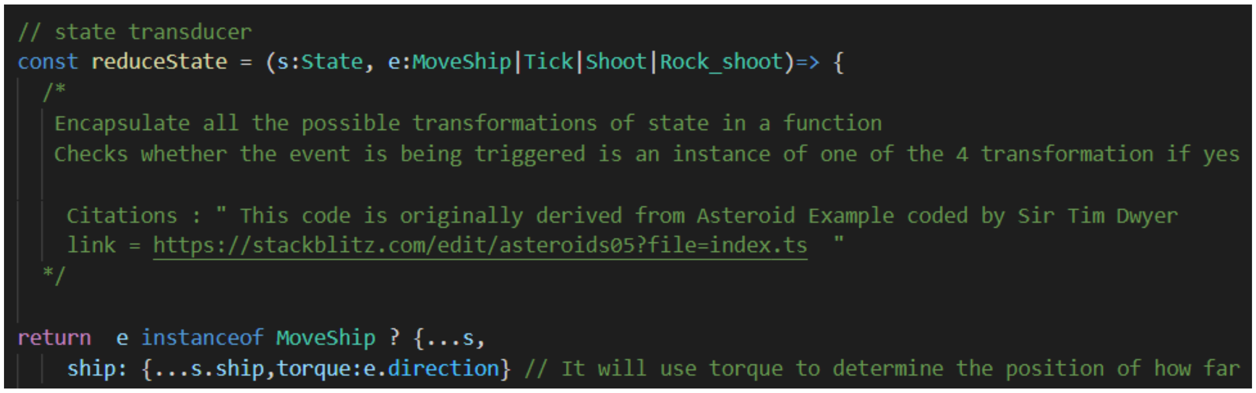


```
startLeftMove = observableStream('keydown','ArrowLeft',()=>new MoveShip(-4)), // Observable to allow ship to move left when keydown.
startRightMove = observableStream('keydown','ArrowRight',()=>new MoveShip(4)), // Observable to allow ship to move right when keydown.
stopLeftMove = observableStream('keyup','ArrowLeft',()=>new MoveShip(0)), // Observable to stop the ship from moving left on keyup.
stopRightMove = observableStream('keyup','ArrowRight',()=>new MoveShip(0)), // Observable to stop the ship from moving right on keyup.
shoot = observableStream('keydown','Space', ()=>new Shoot()), // Observable to allow space to shoot on keydown event.
rock_shoot = interval(1000).pipe(map(_=> new Rock_shoot())) // Observable to handle random rock shooting every 1 second.
```



```
/*
Citations : " This code is originally derived from Asteroid Example coded by Sir Tim Dwyer
  link = https://stackblitz.com/edit/asteroids05?file=index.ts  "
*/
// Four types of game state transitions which allow us to create objects to this particular class
class Tick { constructor(public readonly elapsed:number) {} }
class MoveShip { constructor(public readonly direction:number) {} }
class Rock_shoot {constructor(){}}
class Shoot { constructor() {} }
```

Then , the class object will be passed to the reduce state function as shown below



```
// state transducer
const reduceState = (s:State, e:MoveShip|Tick|Shoot|Rock_shoot)=> {
  /*
  Encapsulate all the possible transformations of state in a function
  Checks whether the event is being triggered is an instance of one of the 4 transformation if yes

   Citations : " This code is originally derived from Asteroid Example coded by Sir Tim Dwyer
   link = https://stackblitz.com/edit/asteroids05?file=index.ts  "
  */

return  e instanceof MoveShip ? {...s,
    ship: {...s.ship,torque:e.direction} // It will use torque to determine the position of how far
```

and it will check whether that event is an instance of one of the four transformation class . If it returns true, then based on that it will do those transformation to the state and on the other hand, it will also pass it on to other function to do additional transformation required for the game and finally created a new states without actually causing any side-effects. Those states are accumulated using the scan function which eventually allow us to store an ongoing state of the game. These states is being passed to the updateView function which update the current new state and draw them out in the svg canvas. The only impure function in my implementation is the updateView function itself. This is basically how I manage the state throughout the game with the help of observable and reduceState function.

The most interesting parts in my implementation I guess is the usage of reduceState to maintain immutability and also allow our state to be managed.

The co-operation between observable, classes and reduceState is indeed very interesting in such a way that they behave. Other than that, I don't think there's something more interesting as compared to this. The handle collision part is also quite interesting in the sense that how those collision are calculated and filtered out from the svg using just an exit array is quite mind blowing I would say.

In conclusion , I would like to give most of the credit to Sir Tim Dwyer for providing us an asteroid template to refer to or else I guess I will be stuck on this assignment for the whole week. This assignments really taught me a lot about function programming paradigm and I hope to learn more about it in the future.