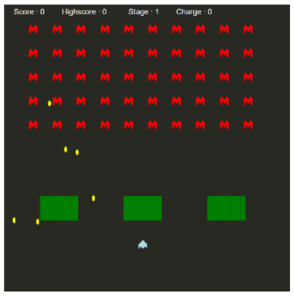
Kin Jian Xin
31165087
FIT 2102
Assignment 1

Assignment1 report

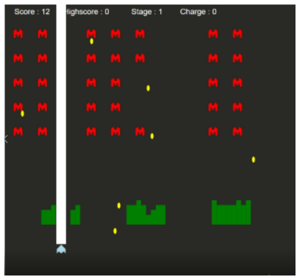
Introduction

In this assignment, a classic space invaders game with appropriate extra features is created using ‘rxjs’ and observable streams hosted in a html webpage. The game code is written in functional reactive programming style such that the program is able to process the observable streams with pure functions while linearising the flow or control and avoiding the implementation of general loops.

Gameplay and design choices



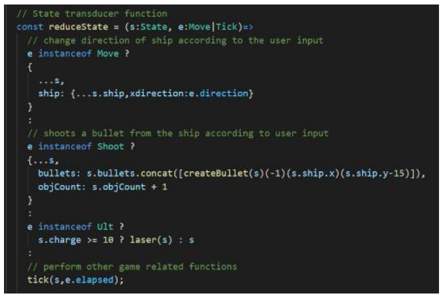
Just like the classic space invaders game, the player plays as a spaceship trying to shoot down rows of aliens before they can reach the player. Unlike the original game, the movement speed and firing rate of both aliens and the player has increased to speed up the pace of the game, thus improving the overall gameplay experience. The newly implemented stage system and highscore system also provide the player with the sense of achievement and replayability, with each subsequent stage having aliens with faster movement speed and firing rate. Besides, during the playthrough of each stage, the remaining aliens will have increasingly higher firing rate the lesser aliens there are currently on the screen to provide even more challenge to the player. There is no actual ‘end’ of the game as the stage count will increase infinitely, so the only main goal is to obtain as much score as possible in each playthrough, as the highscore will be recorded and shown on the top right corner of the screen, this is to provide players with the feeling of a ‘roguelike’ game.



Once the charge is filled up to 10 by killing 10 aliens, the player will have a laser beam charged up. This will act as a final resort to the player as once activated, it will disintegrate anything in the wake of the laser, namely any shields, aliens or bullets. Since this is only intended as a last resort to save the player from danger instead of a way to eliminate aliens faster, the aliens killed by the laser will not reward any score to the player.

Coding/progamming aspects

Controls



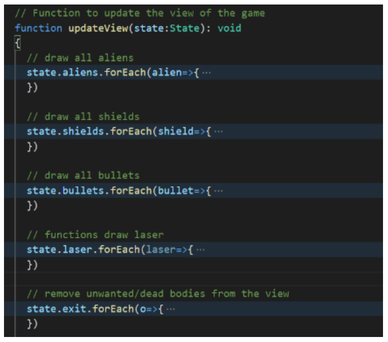
```
// State transducer function
const reduceState = (s:State, e:Move|Tick)=>
  // change direction of ship according to the user input
  e instanceof Move ?
  {
    ...s,
    ship: {...s.ship,xdirection:e.direction}
  }
  :
  // shoots a bullet from the ship according to user input
  e instanceof Shoot ?
  {...s,
    bullets: s.bullets.concat([createBullet(s)(-1)(s.ship.x)(s.ship.y-15)]),
    objCount: s.objCount + 1
  }
  :
  e instanceof Ult ?
    s.charge >= 10 ? laser(s) : s
  :
  // perform other game related functions
  tick(s,e.elapsed);
```

The game is implemented using MVC architecture with the help of rxjs and observable streams. To handle the controls, multiple observables were used to wrap user inputs into streams, after that the streams were merged into one single stream for the ease of processing. Instead of the allowing state of the game to be globally mutable, which goes against the rules of functional reactive programming, the progam uitilises scan operator to perform state transformations in the stream using the pure function : *reduceState*.

Kin Jian Xin
31165087
FIT 2102
Assignment 1

View

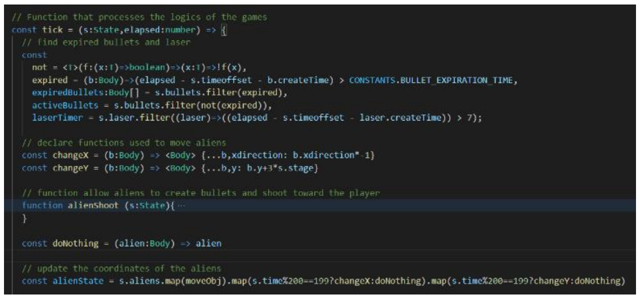
On the other hand, to handle the view aspect of the game, the merged observable stream will subscribe to a *updateView* function that does what it sounds like, updates the view. The function goes through attributes in the current state that contain objects and attempts to draw them out in the html according to their attributes. Furthermore, the function is also used to display multiple other gameplay elements such as the scores, stage count, charge and winning/losing message.



```
// Function to update the view of the game
function updateView(state:State): void
{
  // draw all aliens
  state.aliens.forEach(alien=>{ ···
  })

  // draw all shields
  state.shields.forEach(shield=>{ ···
  })

  // draw all bullets
  state.bullets.forEach(bullet=>{ ···
  })

  // functions draw laser
  state.laser.forEach(laser=>{ ···
  })

  // remove unwanted/dead bodies from the view
  state.exit.forEach(o=>{ ···
  })
```

Managing game state and other game mechanics

To properly update the game state, most of the background logic processing is done in the *tick* function, which is also a pure function such that it has no side effect. The *tick* function called by the *reduceState* function each time the observable stream fires. Inside the *tick* function, it filters out expired bullets, moves the aliens and allows aliens to shoot bullets toward the player.

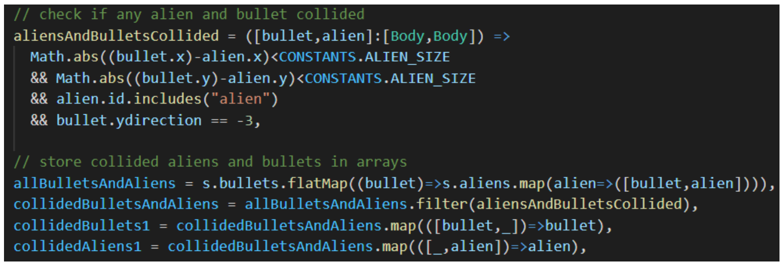


```
// Function that processes the logics of the games
const tick = (s:State,elapsed:number) => {
  // find expired bullets and laser
  const
    not = <T>(f:(x:T)=>boolean)=>(x:T)=>!f(x),
    expired = (b:Body)=>(elapsed - s.timeoffset - b.createTime) > CONSTANTS.BULLET_EXPIRATION_TIME,
    expiredBullets:Body[] = s.bullets.filter(expired),
    activeBullets = s.bullets.filter(not(expired)),
    laserTimer = s.laser.filter((laser)=>((elapsed - s.timeoffset - laser.createTime)) > 7);

  // declare functions used to move aliens
  const changeX = (b:Body) => <Body> {...b,xdirection: b.xdirection*-1}
  const changeY = (b:Body) => <Body> {...b,y: b.y+3*s.stage}

  // function allow aliens to create bullets and shoot toward the player
  function alienShoot (s:State){ ···
  }

  const doNothing = (alien:Body) => alien

  // update the coordinates of the aliens
  const alienState = s.aliens.map(moveObj).map(s.time%200==199?changeX:doNothing).map(s.time%200==199?changeY:doNothing)
```

After that, the function checks the current state of the game, if the player is shot, reset the game by returning the initialState of the game. Similarly if all aliens are shot, increase the stage count, carryover the current score and reset everything else. If the game is not over yet, returns the state of the game after it has been processed by *handleCollisions* function.

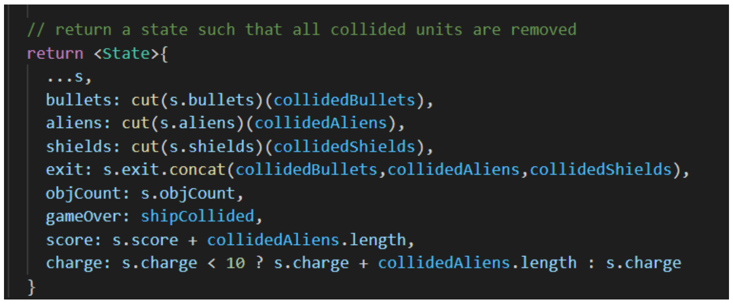
The function *handleCollisions* also does what it sounds like, it handles collisions of all the objects in the game. The function goes through the position of all objects in the game, namely aliens, shields, bullets and the player, then check if their hitboxes has collided.

For example, in the checking between bullets and aliens, instead of using loops to create arrays that contain combinations of all bullets and aliens, flatmap and map operator is utilised to keep in line with the rules of functional programming. The same is done for all other pairs of objects.



```
// check if any alien and bullet collided
aliensAndBulletsCollided = ([bullet,alien]:[Body,Body]) =>
  Math.abs((bullet.x)-alien.x)<CONSTANTS.ALIEN_SIZE
  && Math.abs((bullet.y)-alien.y)<CONSTANTS.ALIEN_SIZE
  && alien.id.includes("alien")
  && bullet.ydirection == -3,

// store collided aliens and bullets in arrays
allBulletsAndAliens = s.bullets.flatMap((bullet)=>s.aliens.map(alien=>([bullet,alien]))),
collidedBulletsAndAliens = allBulletsAndAliens.filter(aliensAndBulletsCollided),
collidedBullets1 = collidedBulletsAndAliens.map(([bullet,_])=>bullet),
collidedAliens1 = collidedBulletsAndAliens.map(([_,alien])=>alien),
```

All the collided objects are stored in arrays so that the function can return a state such that the collided objects has been removed from their respective attribute and added into the exit attribute.



```
// return a state such that all collided units are removed
return <State>{
  ...s,
  bullets: cut(s.bullets)(collidedBullets),
  aliens: cut(s.aliens)(collidedAliens),
  shields: cut(s.shields)(collidedShields),
  exit: s.exit.concat(collidedBullets,collidedAliens,collidedShields),
  objCount: s.objCount,
  gameOver: shipCollided,
  score: s.score + collidedAliens.length,
  charge: s.charge < 10 ? s.charge + collidedAliens.length : s.charge
}
```