Hee Zhan Zhynn

31989403

Space Invaders

The program is made using Functional Reactive Programming (FRP) which is heavily referenced from FRP Asteroids by Professor Tim (2021).

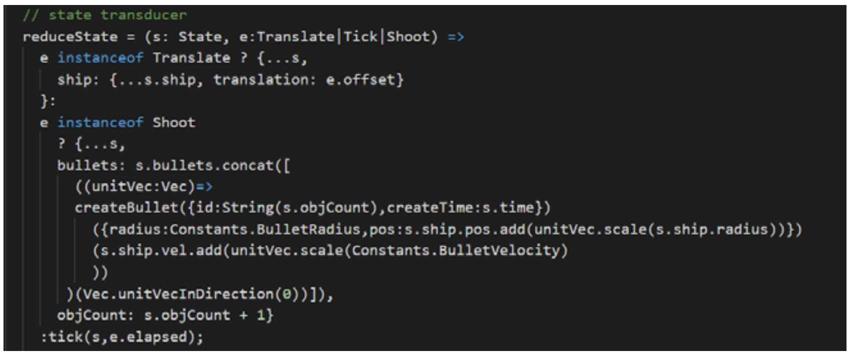
Design Decisions

Const variables are used instead of let to prevent our variables form being mutated, typehints are also added to our input parameters to improve code readability. Our states, bodies and objects are defined as “Readonly” members to prevent mutability. An example is the interface for state objects which is defined as “Readonly” includes all the require states for our game which will be implemented in our initialState and reduceState but does not allow functions to mutate the state. The type viewType which is the view elements types of our game (eg: ship, invaders, bullet…) is declared to identify our objects. For example, in creating bullets and invaders, since they are just circles, we can generalize the function createCircle to create both objects which serves different purpose by using the viewType as the parameter, exploiting parametric polymorphism. Pure functions are also used to transform the states as it has minimal side effects and mutability. Pipeable operators which takes an Observable as input and return another Observable are also widely used in our functions.

Managing game states

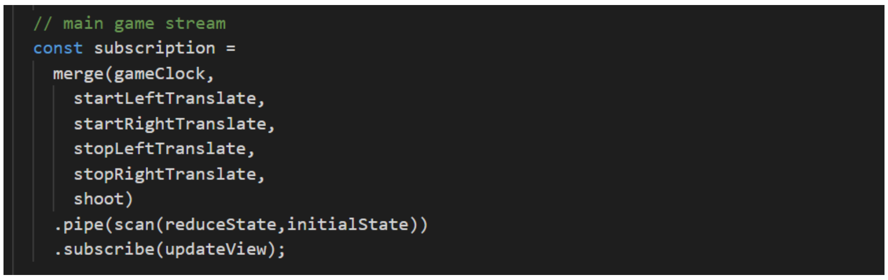
The Observable operator “scan” allows us to capture state transformation inside the stream and apply pure functions to transform the state such that it creates new output state rather than just altering it, i.e the pure function used to move the ship creates a new output state in the stream.

The possible transformations are encapsulated in a pure function known as reduceState, the transformation includes ship movement, bullets shooting and tick which involves collisions and moving non-player objects. The reduceState function is where new output state is created and passed in the observable stream.



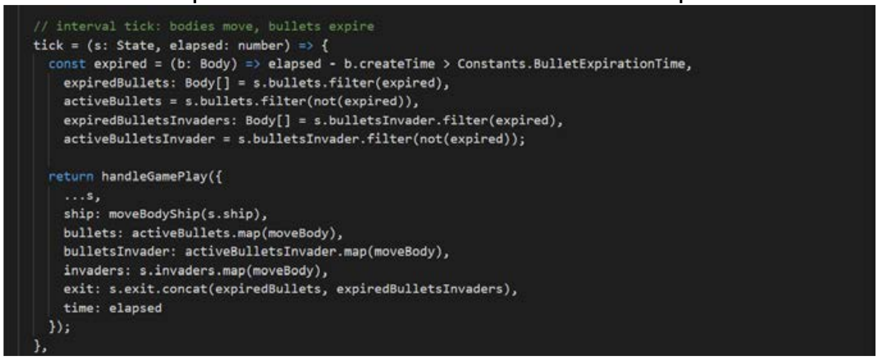
```
// state transducer
reduceState = (s: State, e:Translate|Tick|Shoot) =>
  e instanceof Translate ? {...s,
    ship: {...s.ship, translation: e.offset}
  }:
  e instanceof Shoot
    ? {...s,
    bullets: s.bullets.concat([
      ((unitVec:Vec)=>
      createBullet({id:String(s.objCount),createTime:s.time})
        ({radius:Constants.BulletRadius,pos:s.ship.pos.add(unitVec.scale(s.ship.radius))})
        (s.ship.vel.add(unitVec.scale(Constants.BulletVelocity)
        ))
     )(Vec.unitVecInDirection(0))]),
    objCount: s.objCount + 1}
  :tick(s,e.elapsed);
```

The different inputs from our keyObservables are merged, followed by scanning the initialState and reduceState while the final subscribe calls the updateView to render the new output state, abiding to FRP style in most parts.



```
// main game stream
const subscription =
  merge(gameClock,
    startLeftTranslate,
    startRightTranslate,
    stopLeftTranslate,
    stopRightTranslate,
    shoot)
  .pipe(scan(reduceState,initialState))
  .subscribe(updateView);
```

To provide a smooth and usable game play, the bullets that were created will be removed after a certain period which has been defined the expired function.



```
// interval tick: bodies move, bullets expire
tick = (s: State, elapsed: number) => {
  const expired = (b: Body) => elapsed - b.createTime > Constants.BulletExpirationTime,
    expiredBullets: Body[] = s.bullets.filter(expired),
    activeBullets = s.bullets.filter(not(expired)),
    expiredBulletsInvaders: Body[] = s.bulletsInvader.filter(expired),
    activeBulletsInvader = s.bulletsInvader.filter(not(expired));

  return handleGamePlay({
    ...s,
    ship: moveBodyShip(s.ship),
    bullets: activeBullets.map(moveBody),
    bulletsInvader: activeBulletsInvader.map(moveBody),
    invaders: s.invaders.map(moveBody),
    exit: s.exit.concat(expiredBullets, expiredBulletsInvaders),
    time: elapsed
  });
},
```

The only impure functions in the observable stream is addInvaderShoot where Math.random is used to select invaders at random when a random number generator with specific seed should be used instead. It is impure as Math.random uses external seed to generate different sets of random numbers each time. However, even knowing that it is impure, it is still incorporated in my code as it was my best attempt at generating random numbers without compilation errors.

Game Implementation

The function that create rows of invaders was referenced from Learn RxJS website.

Game instructions in the HTML is also referenced from Purdue University.

Apart from using Professor's Tim bodiesCollided function, I have created my own function known as shieldCollided to determine whether an alien's bullet has collided with the ship's shield. The idea was to check whether the x and y-coordinates of the bullet is within the range of the shield.

The game progress to a new level when all the aliens are shot. The aliens will reset to initial position but will now move faster. This is done by recreating the aliens but passing in a velocity that is based on the game score.

Users are also given a choice to restart the game once it is over.

Improvements

Improvements such as using a random stream generator with specific seed to select random aliens for shooting so that it is a pure function.

References

Learn RxJS (2020). Space Invaders Game. Retrieved from https://www.learnrxjs.io/learn-rxjs/recipes/space-invaders-game

Professor Tim Dwyer (2021). FRP Asteroids. Retrieved from https://tgdwyer.github.io/asteroids/

Purdue University (2021). Space Invaders Game. Retrieved from https://engineering.purdue.edu/OOSD/F2009/Assignments/IPA/invader.html