FIT2102 Assignment 1 Report

Xuguang Wei
32184123

Part 1: Summary of workings of the code

Firstly, the main structure of my workings of the code is referenced to Tim Dwyer's Asteroids game. Similarly, my workings are also interval-based and driven my user input events. Meanwhile, every user input event is also an pure observable stream, or to say, those observables would make required changes by creating new views states rather than altering it in place (details in part 4: state management).

```
// Main
function spaceinvaders() {
  //Observables for user input events
  const startLeftMove = observeKey('keydown', 'ArrowLeft', () => new Move(-1)),
    stopLeftMove = observeKey('keyup', 'ArrowLeft', () => new Move(0)),
    startRightMove = observeKey('keydown', 'ArrowRight', () => new Move(1)),
    stopRightMove = observeKey('keyup', 'ArrowRight', () => new Move(0)),
    shoot = observeKey('keydown', 'Space', () => new Shoot()),
    restart = observeKey('keydown', 'Enter', () => new Restart());
```

The curried function 'createBody' is small and pure, as it takes in multiple parameters, return a new Body object based on those parameters, rather than modify them.

```
const createBody = (viewType: ViewType) => (x: number) => (y: number) => (
  id: number
) =>
  viewType == 'alien'
    ? <Body>{
        viewType: viewType,
        id: viewType + id,
        x: x * 100 + 20,
        y: 100 + y,
        vel: -1,
        radius: Constants.alienRadius
      }
    : viewType == 'shield'
```

Similarly, ‘moveBody’ is also a small and pure curried function, even if its name seems to modify a position of a Body object, in truth, it takes a Body object as input parameter, and then return a new Body object, which only has different positional values with the input one. Therefore, the input one has not been modified and side effects has been kept minimal.

```
const moveBody = (body: Body) => {
  const viewType = body.viewType;
  return viewType == 'shipBullet'
    ? {
        ...body,
        y: body.y - Constants.shipBulletMovement
      }
    : viewType == 'alienBullet'
```

When I create ‘initialShields’ and ‘initialAliens’, I take array indices as parameter to map, which makes the whole array and all its indices remain same, so they are also pure and small.

```
//4 clusters of initial shields
const initialShields = [...Array(Constants.initialShieldNum)].map((_, i) =>
  i >= 0 && i <= 20
    ? createBody('shield')(i % 4)(0)(i)
    : createBody('shield')(i % 4)(10)(i)
);

//3 rows of initial aliens
const initialAliens = [...Array(Constants.initialAlienNum)].map((_, i) =>
```

In terms of the curried function ‘handleCollisions’ and ‘tick’, their main framework is referenced to Tim Dwyer’s Asteroids game implementation, which are also pure. This is because, we take a State object as an input parameter, deal with its values only with pure functions like map and filter, then we return a new State object rather than modifying the values of the input one in place.

Other main functions are related to State management, so I would talk about them in part 3.

```
const handleCollisions = (s: State): State => {
 const tick = (s: State, elapsed: number) => {
```

Part 2: Design Decisions and justifications

1. Spaceship moveable by keyboard inputs

As mentioned before, I take user input events as observable streams, merge them together with the interval stream. Then, those events could be realised while time ticks.

```
const startLeftMove = observeKey('keydown', 'ArrowLeft', () => new Move(-1)),
  stopLeftMove = observeKey('keyup', 'ArrowLeft', () => new Move(0)),
  startRightMove = observeKey('keydown', 'ArrowRight', () => new Move(1)),
  stopRightMove = observeKey('keyup', 'ArrowRight', () => new Move(0)),
  shoot = observeKey('keydown', 'Space', () => new Shoot()),
  restart = observeKey('keydown', 'Enter', () => new Restart());
```

2. Rows of aliens move across and down the screen

As inspired by the demo video on YouTube, I choose to make my aliens move across and down the screen discretely. In the tick function, I firstly check whether there are some aliens reached the left/right side of the canvas, if so, reverse their x movement and going down a little bit; Otherwise, keep current movement pattern. Meanwhile, those changes are pure, as the tick function returns a new State with those new aliens, rather than modifying the input one's aliens positions in place.

```
const alienOffCanvas = (s: State) =>
  s.aliens.filter(({ x }) => x < 0 || x >= Constants.canvasSize).length > 0;
const alienDiscreteMove = (alien: Body) =>
  <Body>{
    ...alien,
    x: alien.x + -1 * alien.vel,
    y: alien.y + 20,
    vel: -1 * alien.vel
  };
const alienNormalMove = (alien: Body) =>
  <Body>{ ...alien, x: alien.x + alien.vel, y: alien.y };
```

```
elapsed % Constants.alienMoveInterval == 0
? alienOffCanvas(s)
  ? {
      ...noCollisions,
      aliens: s.aliens.map(alienDiscreteMove)
    }
  : {
      ...noCollisions,
      aliens: s.aliens.map(alienNormalMove)
    }
: noCollisions;
```

3. The aliens should fire bullets at the user.

In the tick function, if time passed for another interval for aliens to fire, create a new alien based on a random alien position.

This implementation is also pure, as tick function return a new State and the alien bullet creation not modify as input values, as mentioned before, my ‘createBody’ function is small and pure as well.

```
return elapsed % Constants.alienShootInterval == 0
  ? {
      ...noCollisions,
      alienBullet:
        s.aliens.length > 0
          ? s.alienBullet.concat([
              createBody('alienBullet')(randomAlien.x)(randomAlien.y)(
                s.objCount
              )
            ])
          : s.alienBullet,
      objCount: s.objCount + 1
    }
```

4. If the user is hit by one bullet, the user will die and the game will end

In the ‘handleCollisions’ function, similar to the implementation of the Asteroids game, I check the number of alien bullets have hit the ship, if the number is larger than 0, which means the ship has been hit, set the State ‘gameOver’ to be true. And then in the reduceState function, check whether State ‘gameOver’ is true, if so, return current state, which means make the screen view to stay still. Meanwhile, such check is pure and small, as basically only applying filter. And as mentioned before, functions reduceStateand handleCollisions are also pure.
Finally, the html element ‘gameOver’ would not be hid any more.

```
//alien bullets hit ship
collidedAlienBulletsWithShip = s.alienBullet.filter(r =>
  bodiesCollided([s.ship, r])
),
```

```
const bodiesCollided = ([a, b]: [Body, Body]) =>
    calDist(a, b) < a.radius + b.radius,
```

```
return <State>{
  ...s,
  gameOver: collidedAlienBulletsWithShip.length > 0 || aliensReachBottom,
```

```
: //gameover: when game over, set state to be still
s.gameOver
? s
```

```
if (s.gameOver) {
  document.getElementById('gameOver').classList.remove('hidden');
} else {
  document.getElementById('gameOver').classList.add('hidden');
}
```

5. The user has shields, which disintegrate over collisions

Firstly, as mentioned before, the creation of 'initialShields' is small and pure. Meanwhile, I choose to create them in four clusters, which is similar to the shields in the demo video.

```
//4 clusters of initial shields
const initialShields = [...Array(Constants.initialShieldNum)].map((_, i) =>
  i >= 0 && i <= 20
    ? createBody('shield')(i % 4)(0)(i)
    : createBody('shield')(i % 4)(10)(i)
);
```

Then, in terms of making them disintegrate over collisions, in the 'handleCollisions', similar to handle other kinds of collisions, handle the collision between shields and bullets from aliens.

```
//alien bullets hit shields
allAlienBulletsAndShields = flatMap(s.alienBullet, b =>
  s.shield.map<[Body, Body]>(r => [b, r])
),
collidedAlienBulletsAndShields = allAlienBulletsAndShields.filter(
  bodiesCollided
),
collidedAlienBullets = collidedAlienBulletsAndShields.map(
  ([bullet, _]) => bullet
),
collidedShields = collidedAlienBulletsAndShields.map(([_, rock]) => rock),
```

```
return <State>{
  ...s,
  gameOver: collidedAlienBulletsWithShip.length > 0 || aliensReachBottom,
  shipBullets: cut(s.shipBullets)(collidedBullets),
  alienBullet: cut(s.alienBullet)(collidedAlienBullets),
  aliens: cut(s.aliens)(collidedAliens),
  shield: cut(s.shield)(collidedShields),
  exit: s.exit.concat(
    collidedShields,
    collidedAlienBullets,
    collidedAliens,
    collidedBullets
  ),
  score: s.score + collidedAliens.length * Constants.scorePerAlien
};
```

6. Indicate the score of player

Firstly, in the 'handleCollisions' function, update the new returned State score with number of collided aliens, which are hit by ship bullets, times score per alien.

```
//ship bullets hit aliens
allBulletsAndAliens = flatMap(s.shipBullets, b =>
  s.aliens.map<[Body, Body]>(r => [b, r])
),
collidedBulletsAndAliens = allBulletsAndAliens.filter(bodiesCollided),
collidedBullets = collidedBulletsAndAliens.map(
  ([shipBullet, _]) => shipBullet
),
collidedAliens = collidedBulletsAndAliens.map(([_, alien]) => alien),
```

```
),
score: s.score + collidedAliens.length * Constants.scorePerAlien
```

Then in the updateView function, update related html element.

```
score = document.getElementById('score');
score.innerHTML = `Score: ${s.score}`;
```

7. Able to restart when game finishes

Restart is also an observable stream, could be triggered by user keyboard input event.

```
restart = observeKey('keydown', 'Enter', () => new Restart());
```

8. The game progresses to a new level after all aliens are shot

In the ‘reduceState’ function, check if there are no aliens, if so, return a new State with faster aliens, which means the game has come to a higher level.

```
s.aliens.length == 0
? {
    ...initialState,
    aliens: initialState.aliens.map(faster),
    exit: s.shipBullets.concat(toBeRemoved),
    score: s.score
  }
```

```
const faster = (alien: Body) => <Body>{ ...alien, vel: alien.vel * 3 },
```

9. Smooth and usable game play.

All updates are interval-based and could also be triggered by user input events.

Meanwhile, I have taken use of the utility functions in the implementation of the Asteroids game.

Also, ‘createBody’ and ‘moveBody’ functions handle for all kinds of Body objects. And in ‘reduceState’, when set the State back to the initial one, add our current objects to the exit, which is to make sure no current objects still show on the screen.

```
toBeRemoved = (s.aliens, s.shield, s.alienBullet);
```

```
exit: s.shipBullets.concat(toBeRemoved)
```

Part 3: State Management

Based on what we have seen in the previous two parts, we can see that the main State management is in the functions ‘reduceState’ , ‘handleCollisions’ and ‘tick’, which are both pure. In terms of those three functions, ‘tick’ deals with the basic time-based updates, such as bullets expiration and aliens fire/move, ‘handleCollisions’ deals with all kinds of collisions. Meanwhile, ‘reduceState’ deals with user input based events, take use of the returned new States by the other two states and finally get the required update State, to be used for ‘updateView’.

Part 4: FRP Adherence

In the past three parts, based on what I have mentioned, we can see that, except for the subscribe for updating view, all other parts of my design are pure, as all functions not modifying input parameters’ values in place. Instead, they return new object with updated values.

In terms of immutability, similar to the implementation of the Asteroids game, the values of State and Body are almost all <Readonly> and also themselves, which means no in-place modifactions are allowed for those two types.

```
type State = Readonly<{
  ship: Body;
  shipBullets: ReadonlyArray<Body>;
  exit: ReadonlyArray<Body>;
  objCount: number;
  aliens: ReadonlyArray<Body>;
  alienBullet: ReadonlyArray<Body>;
  gameOver: boolean;
  score: number;
  shield: ReadonlyArray<Body>;
}>;

type Body = Readonly<{
  viewType: string;
  id: string;
  x: number;
  y: number;
  vel?: number;
  radius?: number;
}>;
```