Acknowledgement

1. All the knowledge used for this assignment about RxJS including the usage of all the operators, and functional programming were from Learn RxJS and Course Notes.
2. The structure of the codes were mostly learnt from the given Asteroid FRP code by Tim Dwyer.
3. All the requirements and marking rubrics of the assignment are linked here.

Overview of Code

The code created is for a simple gameplay of Tetris, with all the simple operations, including all types of Tetrimino blocks, movements, rotations, score incrementation, row deletions, collision detections, difficulty levels, game endings when necessary.

Game instructions

1. Controls

- 'A' key - move tetromino block left
- 'D' key - move tetromino block right
- 'S' key - move tetromino block down
- 'X' key - rotate tetromino block anticlockwise
- 'C' key - rotate tetromino block clockwise
- 'R' key - restart when game ends
- 'Space' key - instant drop

2. Rules

- Block stays within the canvas, and does not exceed horizontally or vertically.
- Blocks can be moved right, left, and down. Continuous pressing of the keys will result in continuous movements.
- Block is rotated based on the Super Rotation System, with wall kicks implemented, where the block will not rotate if a collision with either the wall or another block is detected.
- Tetromino blocks will stop when collision is detected, and a new block will be generated, based on a random number generator.
- Rows will be cleared when it is fully filled.
- Difficulty level increases with the level.
- Game will end when any block reaches the top of the canvas.

Overall Structure

1. Game Loop

The core logic of the game is all within the main function. This includes setting up the game canvas, handing user inputs, managing the game state, and rendering the visuals of the game.

2. Render

The render function inside main is used for rendering the game's visuals on the canvas. It updates the position of the blocks, displays the score, level, highscores, and handles gameover and restart mechanics.

3. User inputs

Handles all user inputs via keyboard using RxJS observables, to create streams. Observables serves as the bridge between the user input and the game behaviour.

4. Operation Functions

Manages the essential gameplay aspects, including enabling tetromino movement, check collisions, remove filled rows, calculate scores, adjusting tick rate, and rotations. These functions ensure a smooth gameplay experience.

Design Decisions and Justifications

1. Immutable State

In my specific design, the state is the main object of the game. The state stores all the important values that will be changed often, including the current and next tetrimino, score, level, highscore, tickrates and game status.

The state is declared as ReadOnly, to prevent mutations, or impure and imperative programming. The state is updated through pure functions inside the main function, through user inputs and game events. These functions take in the current state as input, and return the updated state without mutating the original state. This ensures that the game state remains unchanged unless updated.

2. Modular Structure

The code is structured into separate modules, as this improves code organisation and maintainability. Each separate module has a singular purpose, making it easier to

understand, update, and debug. This adheres to the principles of modular programming, allowing the extension of each component without affecting the others.

3. Random Number Generator

The RNG I created embodies the principles of functional programming. It facilitates the generation of pseudorandom integers within a defined range using a linear congruential generator (LCG) algorithm. The LCG employs a seed value to produce deterministic but seemingly random numbers, adhering to the functional paradigm by maintaining state solely within the class instance. The seed that I selected is 987654321. This is used in the random generation of the tetromino blocks.

Functional Reactive Programming Principles

1. Observables

Observables are a fundamental part of FRP, as it is a powerful tool for managing and transforming data streams. Observable streams can be manipulated, and transformed, allowing us to create complex data flows.

I used observables in various scenarios including event streams, asynchronous data emission, and subscription management. I created observables using the imported 'fromEvent' function, to capture keyboard inputs, to create a stream of events for movements, rotations, and restarts. Then, I used the 'merge' function to combine multiple observables into one stream, representing all the actions that will affect the game state. Ths 'scan' operator is used to manage the state based on all the actions, including the determination of game end.

The usage of observables makes the code more organised and maintainable, allowing us to easily debug the code when errors occur.

2. Time-based Control

The 'interval' observable used is done to control the timing of the game events, including the constant drop of the tetromino block, based on the tick rate. The tick rate is changed according to the level of the game.

3. Functional Transformations

Several functional transformations were used to manage the game behaviour, using the RxJS library. These are all used in the game for a more reactive and structured approach. This allows the code to be more modular.

- fromEvent - Create observables from Document Object Model (DOM), convert keyboard events to observable streams.

- map - Used to map key events into specific operations and action strings.
- filter - Only allows certain conditions to pass through, to ensure only specific events are continued.
- scan - Accumulate value over time.
- switchMap - Switch to a new observable when a new event occurs.
- merge - Combine multiple observables into a single stream.
- BehaviorSubject - Retains the most recent value and emits it to new subscribers.

Additional Features

1. Drop down feature

I implemented a dropdown feature, where when the space key is pressed, the tetromino block will instantly drop down, to the closest colliding block, or the canvas. This allows a smoother gameplay and user experience.

The method I used for this is very similar to the movements and rotations. Firstly, I added the space key. Then, I made a pure function called dropdown, where it utilises recursion to keep dropping the block's position by 1 as long as the base case is not met, where there is no collision met with either the existing blocks or the canvas ground. A dropdown observable is then created, and merged together with all the other movement observables into the main stream. In this specific action, a new block is immediately added after the block is being dropped down. This is done to ensure that once the block is dropped, it leaves no time for users to continue pressing the spacebar, still acting on the same tetromino, creating some unwanted results.

Summary

To summarise, the code represents a functional and modular implementation of the game Tetris, following all the principles of functional reactive programming. Key design decisions were made to enhance the code quality and maintainability. Then, immutable state management ensures predictability and alignment with FRP principles, by changing the state through pure functions. Additionally, a modular structure separates concerns and promotes the readability of the code.