Title: Reactive Tetris Game - Functional Programming Report

Name: Lim Jun Kee

Student ID: 33207798

Introduction:

This report provides an overview of the implementation of a Tetris game using Functional Reactive Programming (FRP) principles. The game is designed to showcase the use of FRP, observables, and pure functions to manage the game state and interactions.

Overall working of the Code:

```
  const reduceState = (s: State, action: Action) => action.apply(s);

  const source$ = merge(gameClock$, left$, right$, down$,
rotate$,teleport$,restartClick$)
  .pipe(scan(reduceState, initialState));

const subscription = source$.subscribe((s: State) => {
  render(s);

  if (s.gameEnd) {
    show(gameover);

     // Unsubscribe from the source$ observable
  } else {
    hide(gameover);
  }
});
}
```

The interesting part of this code is merge() function combines multiple observable streams into a single source using the merge operator, simplifying event handling in a reactive application. By piping this merged stream through the scan operator with an initial state, it elegantly manages state transitions by applying a pure reducing function. This approach enhances code readability and maintainability, as it enables concise tracking of state changes over time while preserving immutability.

Design Decisions:

- The decision to use FRP was driven by the need for a clean and declarative way to handle game events and state changes. RxJS observables provide a powerful toolset for achieving this.

- Immutability is emphasized throughout the codebase to ensure that game state changes are predictable and do not introduce unexpected side effects.

- Functional purity is maintained by using pure functions wherever possible, reducing bugs and making the code easier to reason about.

- The choice of TypeScript as the programming language further enhances code quality and maintainability by providing strong static typing and better tooling support for error detection and refactoring. This ensures that the code remains robust and less error-prone, aligning with the goal of producing a high-quality game application.

The core loop of my Tetris game revolves around a merge of various Observables, including gameClock$, left$, right$, down$, rotate$, teleport$, and restartClick$. These Observables handle both asynchronous input and default Tick updates. The key to this loop is the invocation of the apply() method within specific class objects representing game actions. This method updates the game state based on the input and maintains a reactive approach. The resulting state is then passed to the rendering function, which updates the HTML elements to reflect the current game state. This cycle ensures a dynamic and responsive gaming experience by seamlessly integrating user input and automated game updates.

Another core implementation of my code is I use 2D array to visualize the blocks, this makes me check collision and remove the blocks more easily, so every Action my state will update the 2D array.

FRP and Observable Usage:

-My code effectively follows Functional Reactive Programming (FRP) principles by employing Observables to manage asynchronous events and pure functions for state updates. This approach simplifies the control flow, maintains immutability, and allows for the composition of event streams. By modeling game actions as Observables and using pure functions to transform the state, my code embodies the essence of FRP, enhancing readability and predictability.

- Observables are used not only for handling user input but also for managing game animations, timing, and event propagation. One of the interesting features is we can use pipe to effectively manage game events. By merging multiple Observables into source$ and applying the scan operator with reduceState and initialState, we ensure a predictable and pure functional way to update the game state

State Management:

- In the code we will return a deep copy of an object or array with modified property to maintain immutability. We also make sure to not use impure function and variables to also maintain the code

immutability such as let, for loop and while loop. In order to achieved loop effect, we use recursion to maintain the code purity, in terms of modifying an object or array, we return a deep copy of the original object and array with newly update properties. This allows us to properly keep track the correct information about the state, make us debug easily and prevent unpredicted event happens.

Usage of Observable beyond simple input:

In my Tetris code, I've harnessed the power of Observables beyond handling simple input. Specifically, I've used the `interval` Observable to manage the game's tick rate and track the current state creation time. This utilization aligns with Functional Reactive Programming (FRP) principles. The `interval` Observable provides a consistent time reference for regulating the game's ticks, ensuring smooth and predictable gameplay. By maintaining the state creation time within the game loop, I can effectively control the timing of game events like block movements and updates, adhering to FRP's functional and reactive approach to managing game behavior. This demonstrates how Observables extend their utility beyond input handling to maintain game state and regulate game mechanics. For additional by adjusting the interval observable I can make my tick rate faster which makes my blocks move relatively faster.

Additional Features:

-Keeps track of high score across previous rounds:

In implementing this feature I create a class called Restart class that will return an updated state, when a restart button is being press, I assign it to the merge() function which makes it asynchronous, the reduce() method will invoke the apply () method and return a deep copy of updated state with its highest score being update. This action is aligned with Functional Reactive Programming (FRP) principles as it returns a new state (immutability) and ensures a predictable and pure function, without side effects. Additionally, it updates the "highestScore" property if the current score is higher, maintaining consistent state management in the FRP style.

-All Tetris pieces that can be built using 4 blocks:

In my Tetris code, I've implemented an array containing all possible Tetris blocks. To introduce an element of randomness and variety, I've designed a function called "spawnRandomNewBlocks(stateCount: number)." This function leverages Functional Reactive Programming (FRP) by taking a random number and scaling it between 0 and 6 through a hashing process. This scaled number is then used to select a specific Tetris block from my pre-defined array of blocks, ensuring unpredictability and excitement in the game. The chosen block is returned as a deep

copy, aligning with FRP's principles of immutability and predictability, as it doesn't alter the original block array and maintains a clear and functional flow of block spawning. This approach adds a dynamic and engaging element to my Tetris game while adhering to FRP principles.

-Next shape preview:

In my Tetris code, I've used the concept of a "previewBlock" as a property of my game state. Before introducing each new block into the game, I update the current state by assigning the "current fourBlock" property with the "previewBlock" from the previous state. Following this, I utilize the FRP principles of immutability and predictability to update the "previewBlock." This is accomplished by invoking the "spawnNewBlock()" method which will spawn a deep copy of the hardCoded fourBlocks in my code randomly. This FRP-based approach enhances the gameplay dynamics and maintains clear state transitions within my Tetris game.

-Upcoming shape is randomly selected:

In my Tetris code, I've implemented a pseudo-random number generator (PRNG) using an abstract class called `RNG`. This class employs the LCG algorithm with specific constants to generate a sequence of hashes based on a given seed. To introduce an element of randomness in a Functional Reactive Programming (FRP) style, I've created a function named `spawnRandomNewBlocks(stateCount: number)`. This function takes an integer `stateCount` which will be different for every new state updated, uses the `RNG` class to hash and scale it within the range [0, 6], and then selects an upcoming Tetris shape from my predefined block array. The use of the PRNG and scaling ensures unpredictability and randomness in shape selection, aligning with FRP principles to enhance gameplay dynamics and maintain clear, functional transitions in my Tetris game.

-Able to restart when game finishes:

In my Tetris implementation, I have a `gameEnd` Boolean property in the game state, serving as an indicator of whether the game has concluded. Leveraging Functional Reactive Programming (FRP) principles, I've designed a rendering logic that checks the `gameEnd` property. When the game ends (i.e., `gameEnd` is true), it displays the "Game Over" message along with a restart button in the game's HTML. The key FRP aspect comes into play when the player interacts with the restart button. Upon clicking it, the `apply` method of the `Restart` class is invoked, resetting the game state to its initial values, including setting `gameEnd` to false. This ensures a seamless transition out of the game over state, following FRP's emphasis on clear and predictable state management while enhancing the player's experience by allowing them to restart the game.

-Increase game difficulty (Increase block speed):
In my Tetris implementation, I've adjusted the tick rate by changing the interval from 500ms to 10ms, significantly increasing the game's speed. To manage this faster tick rate while maintaining gameplay

balance, I've incorporated Functional Reactive Programming (FRP) principles. I introduced the currentTickTime variable to keep track of time elapsed between ticks. When currentTickTime reaches a certain threshold (in my code, set to 80), I trigger the apply method of the Tick class, which updates the game state as usual and resets currentTickTime to 0. This approach allows me to dynamically control the effective tick rate by manipulating currentTickTime while keeping gameplay smooth and responsive. Additionally, I've linked the currentLevel to the currentTickTime formula, currentTickTime amount to reach =(Number constant)/(currentLevel), in my code number constant is 80. So, the higher my currentLevel get the lower my currentTickTime amount to reach, which makes the block moves relatively faster when my level increases.

-Shape Rotation:

In doing rotation I implement the super rotation system. First I set a center point for every shape, which is the arrayBlocks[1](Second block element of every shapes), then I find the relative distance for every block x,y coordinates. Relative distance= centerBlock(x,y) – demandedBlock(x,y), then I use Matrix multiplication constant ([[0,1][-1,0]] * relative distance(x,y))+centerBlock(x,y)= new rotate Coordinate. I have develop a handleCollision() method for rotation which will return the state before rotation if the rotation happens to collide with another block or grid boundary.

-Hard drop:

To implement the hard drop feature in my Tetris game while adhering to the immutability constraint of Functional Reactive Programming (FRP), I opted for a recursive approach instead of using loops. When the player initiates a hard drop, the shape descends by 20 units at a time until it collides with another block or the grid boundaries. To maintain immutability, I keep track of the initial shape position and remove it before completing the hard drop. This recursive technique ensures that the game state remains immutable throughout the hard drop process, aligning with FRP principles and providing a seamless player experience.

Conclusion:

The Tetris game implementation demonstrates the effectiveness of FRP and observables in creating a robust and maintainable application. The code adheres to functional programming principles, emphasizing immutability and functional purity. By using observables beyond simple input handling, we achieve a declarative and reactive architecture that makes the code concise and straightforward to

understand. The design decisions are motivated by the desire for a clean, predictable, and maintainable codebase.