Yenny Fransisca Halim 33448930

FIT2102 Report – Tetris

Design decision:

The code is designed with purity, modularity, and reactive design principles. It adheres to the FRP and use the observables to handle asynchronous events and manage state. Hence, the code is scalable, robust, and maintainable.

The **"switchMap"** operator was chosen to manage the internal observables for the ticking rates dynamically instead of "mergeMap". This is because mergeMap could lead to multiple active inner observables, which something that was not suitable for this specific use case.

- All the utility and reducer functions are coded in pure which make it easier for reasoning and follow the **Functional Programming style**. It also takes care of the specific game actions such as blocks rotate, updating the grid, color, etc.

- **Use FRP helps** in streamlining state change and to update the UI.

- **Separated the pure functions and impure functions** which any side effects or DOM manipulation are isolated from the pure code.

- **The state management** maintain immutability because it utilizes the Readonly<> built in function and subsequent state are derived from initial state. Each function either returns a new state or a modified copy of the state instead of altering the original state. It also prevents side-effects. Functions like reduceState, reduceKey, and reduceTick are pure and it depends only on their input arguments for the next game state.

FRP

Observables

- Merge: the events (keyboard and ticks) are combined by using merge into one observable streams. It is then subscribed to update the state.

```
const tickRateSubject$ = new BehaviorSubject<number>(initialState.tickRate);
```

- **tickRateSubject$** is serves as a centralized, reactive way to manage the tick speed of the game (to update).
- It uses to manage the tick rate of the game (for difficulty) when the level reaches level 5 and above (based on my implementation), it will increase the block speed dynamically with the help of **BehaviourSubject** from RxJs library because BehaviourSubject is a specialized type of subject that can both emit data and register multiple subscribers.

- It uses **“scan” operation** to accumulate the changes in state as events circulate within the system. It also manages my state. It takes the current state and action, then returns a new state without any side-effect.

- **Reactivity:** the observable streams allow the game to be inherently reactive. The action$ observable will react to the state change due to a keypress or a “tick” and it will update the game state accordingly. Then, the new state is rendered to the canvas.
- Recursive with observable for dropTheBlock function to find and compute the final position of the falling block until collision is detected.

- **distinctUntilChanged()** operator is used in the code to filter out the emitted values from the source observables to pass through. It only allows different emitted value to pass through. Hence, if the value is the same, the second emission will not pass through this operator, and it is to prevent unnecessary recalculation or side effect downstream.

- **RNG stream** are used in the code to generate random number for generate random block by using hash() to hash the number and scale() to get the random number from hash.

Advance $ Additional Feature:

Some basic additional feature like **drop the block** when the user press “space” has been implemented in this game by using observables similar with the movement (left, right, down). dropTheBlock function is a recursive function to find its lowest point that the block can land.

Wall kick has been implemented in this assignment to provide a solution for rotate problem of block rotations near the grid or walls, or other blocks. The purpose of this implementation is to enhance the gameplay experience by making the rotation flexible and intuitive.
- When rotate, it pivots around a “pivot” point which I chose the middle (length of the block / 2) .
- If the collision would give a collision, the wall kick feature will kick the block either to the left or right side and see if it can achieve a collision-free rotation.
- If none of the solution (kick left or right) solve the problem, it returns the original position. Else, the collision-free state is returned.

Ghost Piece feature implementation using the FRP and state management. This feature is to improve gameplay experience by letting the user know where the current block will land.
- **State management:** the coordinates of the ghost piece are stored in ghostPoint which is a part of immutable state in our game.
- **Reactivity:** every time the game state change, whether it through rotating the active tetromino, moving it, or drop it, the ghost position will recalculate (calculateGhostPiece function) and refreshed it by using updateGhostPlace function.

- **No side effects** because the calculateGhostPiece and updateGhostPlace function does not depends on external variables which is in line with **FRP**.