FIT2102: Tetris Report

Name: Kai Le Aw

ID: 32202490

This is a report about Tetris implementation by using Functional Reactive Programming techniques to enable programmers to code functional programming and be able to observe reactive programming concepts as it is part of Observable that allows users to communicate with systems in ease. It simplifies our coding structure for us to use combinators to handle event listeners. Thus, we can perform both asynchronous and synchronous operations to handle user inputs and sources at once.

Several typescript files created to organize the programming parts by differentiating their responsibilities:

- main.ts is main function call
- state.ts is to handle changes in state
- type.ts is a collection of types and constants
- view.ts is to handle display

FRP and Observables help to maintain code purity by dividing pure data and data transformation that will be applicable to SVG elements. The scan function has applied reduceState onto the latest state, this will create a new state without modifying the previous state.

Readonly Array helps to avoid changes and maintain data purity. It has been applied on several array types known as TetroProperties (Block is generated randomly based on RNG class functions. A Date() calls getMilleSeconds() and placed into hash and scale function, it applies to every block generated to get purity) and States. Lastly, impurity of data happens in updateView as it changes the boards for display purposes in SVG.

The game starts off with the main function call, it imports rxjs and its operators to start. The usage of Observable starts by having a source$ observable to receive multiple input Observables and merged in a single stream by using FRP merge.

The scan function will apply the reduceState onto state.The reduceState has assigned actions to every key input to perform conditional checks and apply changes and return as a new state.

After observables are captured and dealt with by reduceState. The subscribe function will handle side effects and update views to let players see if their moves are smart choices (appropriate placement block to clear rows). The updated state that is remaining pure is passed to updateView function to apply appropriate changes onto SVG and preview elements.

Majority classes and functions are created and stored in the State file. This is convenient as the state will apply many functions to output an updated state element to return to subscribe.

Ticks per fall action	Level
5	1
1	5

The tick$ source is an Observable that is a time interval function and being used to let block falls until it encounters obstacles or boundaries. An interesting part is the timer has been changed by me to enable tick$ fall faster. The main purpose is to implement the speed difficulty challenges. Level increase also led to faster fallen speed and this is implemented by the state having a counter to keep track of how many times tick$ received only allows the block to fall. Therefore, I remain purity by remaining tick$ interval value and handle the speed falling for blocks without modification of variables.

Keep$ Observable is an additional feature implemented to allow the user to reserve a block in his pocket. It creates additional elements (reservable and inventory) in initialState to let users store a block. Reservable is always true unless KeyC is captured by Observable and turns false. It will be reset only after a block is placed. This is done so that users cannot keep swapping between current and inventory to get back at top of the gameboard. This is pure as the data is preserved as it only changes in the next state.