FIT2102 Assignment 1 Report

Name: Amirul Mohammad Azizol
Student ID: 32619898

Overview

General Approach

The game state is represented as an immutable object of the State type, which includes important properties such as the game board (represented as a grid of 0's and 1's), the current, next, and held blocks, the score and level, and so on. I also declared other various types, including generic types for the lazy sequence. To update the board, I had the updateBoard function which maps a function checking if the coordinates match the block for every square within the row, then mapped again to every row within the grid. I also did a similar thing with filtering and checking the length, in order to clear rows and increment the score / level.

To rotate blocks, I chose to follow the SEGA system. To do this I used a lookup table where the key is the name of the block, and it contains 4 groups of x/y coordinates representing the blocks possible rotations. These constants are at the top by default, so a rotation requires me to calculate the distance travelled from the origin and translate the rotated version of the block accordingly.

Action Interface

This action class emulates the example shown in the FRP Asteroids, and is used for any user interactions or events in the game, such as moving a block or rotating it. Reducers (apply methods) are responsible for updating the game state based on the action applied. This allows for state management without mutation. All of the Actions responsible for features will take a state as a parameter and return a new, slightly modified state based on the old one (using the spread operator).

Design Decisions

Lazy Sequence

A lazy sequence is used to generate random numbers for block selection. Aside from rendering, this is one of the areas where an impure function is used, but random functions are inherently impure. The lazy sequence is pure, which makes it very useful for testing - we can set a constant seed which results in the same sequence every time. However, for gameplay purposes, it must be initialised with a random seed which cannot be the same each time, using the impure Math.random() function.

Higher Order Functions

I made use of various high order functions which either input or output a function. One example of this is my updateBlock(movement) method, which inputs a blockGroup and a movement function, and maps that function to each of the individual blocks. Another example is the createTickObservable function, which creates the tick observables corresponding to difficulty levels, and the createKeyObservable function, which creates a key observable with the function input being an Action factory.

Game Loop with Ticks

The game loop is implemented using observables that emit actions at regular intervals (ticks). Scan is used to accumulate the state of all the merged observables, including tick. The tick rate varies with the game level, making the game faster as the player scores more points. I tried to implement a function that infinitely speeds up, but this was difficult without extensive knowledge of rxjs, or simply an imperative programming approach. I eventually decided to just have 3 levels as predetermined constant tickrate values.

The tickrate is involved in the creation of streams, so it can't really be tracked in the state. I had a function that generates tickstreams from my previous attempt to scale infinitely, just that it needed to be initialized before the game started. I used a special rxjs operator called switchMap to dynamically switch between my 3 levels. If the game ends, I use switchMap to change to a placeholder observable (an rxjs Subject that simply emits void) to override the previous tick observable from occurring as the game has ended.

Additional Features

I elected to add two extra features to tetris, both to make my own testing more convenient and to make the game more fun to play. These are:

- hard drop (press X to instantly drop your block)
- hold block (press H to swap the stored block with your current block)

Both of these features impact the state management in non-trivial ways. For the first feature, I used tail recursion to input repeated down movements until the block stops, and that state is returned.

The second feature basically works as a toggle for the user to "hold" the current block, saving it for later. The game starts with no held block, but once a block is held, the user can keep it there until they press the key again, which will swap the held block and current block and move it to the top. This required additional HTML and attributes in State.

When swapping blocks I also had to ensure that the orientation and position were fixed. I did this by using the rotation lookup table (see "Overview") to get the original orientation. Another requirement in tetris is that you can only hold once before dropping a block, which led me to implement a boolean value in the state. This starts out as false, becomes true once the player holds a block, and becomes false again when any drop is blocked, allowing the player to hold again afterwards.

These features come at the cost of additional complexity, but they were relatively straightforward to implement given that the existing code was pure, functional, and referentially transparent.