FIT2102 Assignment 1 Design Brief

Glorison Lai
30587220

Design Goal

This project implements a version of Tetris in a functional paradigm. The functional paradigm involves the use of pure functions to manipulate immutable variables to simulate state, and isolates functions that perform side effects, such as rendering. Typescript allows us to enforce this by providing strict typing, and constant variables.

Events

The main observable is a single RxJS Interval Observable, that acts as a global clock. All other observables are synced to this global clock, allowing for a single source of truth, and constant game updates. In each “tick”, we map the tick number, and any associated player input, that is mapped to an action. This is then piped to a scanned tick function with an initial game state, that creates a new game state.
This new game state is then rendered onto the canvas elements.
This loop remains the same regardless of the current state of the game, however, the player inputs are mapped to different functions, depending on the game state. For example, while the game is in progress, player inputs are mapped to movement actions, while in the Game Over state, player inputs are mapped to restarting the game. Having the state define what pure functions player inputs get mapped to makes streamlines implementation, and makes unit testing these functions simple.

State

The state of the game is contained in a single object, and is used to create the next updated state every tick. This state contains the currently placed tiles on the board, the player’s current tile, the preview tile, as well as the state of the game and the player’s scores. Keeping the state as a single object makes writing pure functions easy, as functions can be given the entire state of the game to create a new one, while keeping the original state safe from modification.
The state is initially passed into the scan operation, which accumulates operations on the same state type, while also passing the updated state to the next function in the pipe. This keeps the scan function pure, as every state the scan function returns does not mutate the state the scan function is accumulating..
The main drawback to keeping state in a single object is its size, particularly having to store the entire board inside the state object. In order to retain purity, every function that must return a new state, even if it does not use the board in the function, must create a new copy of the board. This may lead to slow downs and lag for larger boards and states, however for this implementation, the game experience was not severely affected.

Game Logic

Game logic was kept relatively conservative. The main game objects are blocks, which take up a single unit of space on the board, and tiles, which are the player pieces, and are a collection of blocks in the shape of traditional Tetris pieces, that have not yet been placed onto the board.
The collection of tiles are defined in a single array, with an associated union type for its elements, and is used to index an object storing the tile shapes and rotations. The tile shapes are kept in a separate dictionary and accessed via a function to avoid having to store the shape in the game state, and unnecessarily bloating the state.
When a game is in progress, every tick, the current player piece is automatically moved down one space, and a new state is created.
This new state is then used to map any player input that moves the current player piece, creating another new state.
Each state is then tested for validity, and may be used as the next state.
If the player state is invalid, that is the resultant player movement causes the piece to overlap with a block, or go outside the board, then the automatic state is tested. If that state is also invalid, that is the tile cannot move down one more block space, then the player's current tile is placed onto the board, and the player's preview piece becomes the player's current piece.
Finally, before moving onto the render step and the next iteration, any filled rows are cleared, and accumulated into the player's score. Every 10 rows, the tick rate for block movement increases, effectively increasing the game speed and difficulty.
If a tile that is first generated cannot be placed onto the board, then the state moves into Game Over, and awaits player input to start a new game.

Rendering

Rendering is the only side-effect in this implementation. This is kept isolated from the rest of the state in the subscribe method body.
The game renders in two separate canvases, and prefers drawing shapes instead of creating new SVG elements. This makes the game run smoother, as using the Canvas element to draw shapes does not clutter the DOM.
Before starting the game, we first find all of the DOM elements
Rendering is performed by first clearing out the canvases every frame. This ensures that what is drawn to the screen purely reflects the current state of the game, to mimic purity.
The most common rendering functions used are *createBlock* and *createTile*. *createBlock* draws a single box onto the canvas, and takes as input X and Y positions, as well as the type of block. The type of block is mapped onto a object that contains what colour to fill the block. *createTile* calls *createBlock*, in the shape of the Tetris tile.
Rendering also responds to state changes, such as the Game Over state, in which case it shows the Game Over screen. On player input, the render step will hide this screen.