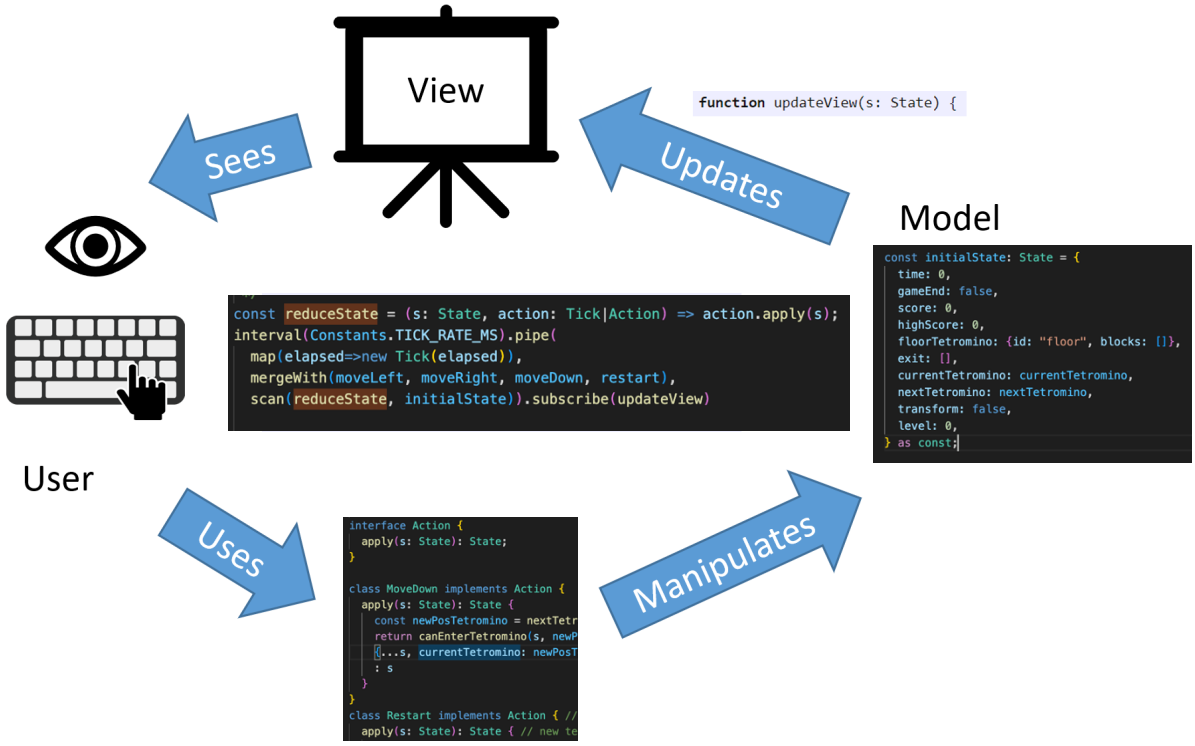
FIT2102 ASSIGNMENT 1 – REPORT

BY: Cheah Jit Yung 32837011

This report will provide a basic overview of the game in sections:

1. State Management using Observables
2. Design Rationale

State Management using Observables



*The game code has been inspired by the asteroids example.

State Management in 4 parts

1. A constant stream of observables stream which acts as a game clock, ticking every 500ms.

   This is achieved by using interval. Then, the stream of number is converted into a stream of Tick Action. In addition, the game's time is kept tracked by having time as an attribute for state and requiring it to be passed in by construction.

2. User Input handling

```
/** User input */
const observeKey = <T>(eventName:Event, k:Key, result:()=>T)=>
    fromEvent<KeyboardEvent>(document,eventName)
      .pipe(
        filter(({code})=>code === k),
        filter(({repeat})=>!repeat),
        map(result))

const
  moveLeft = observeKey('keydown','KeyA',()=>new MoveLeft()),
  moveRight = observeKey('keydown','KeyD',()=>new MoveRight()),
  moveDown = observeKey('keydown','KeyS',()=>new MoveDown()),
  restart = observeKey('keydown','KeyR',()=>new Restart())
```

   Basically, the code above uses observables to keep track of key presses. It filters out keys matching the keypress, whilst ensuring there are no repeated key for better user engagement (no long presses). In the end, the specified action for that key is mapped to it.

3. Applying actions to the state based on user's input by reducing it.

   Then, all these action streams are merged into one observable stream. This allows action to be completed asynchronously. Hence, along with Tick actions are Move Actions.

4. The view is modeled based on the game state.

   The updateView function will be the function containing side effects, where it manipulates the object view. Any action applied to the state will be visualized using this function.

Hence, all these ensures that the impure code will be contained within updateView, which is important to avoid bugs if side effects happens at multiple areas, such as being applied twice. In addition, side effects are also avoided by not using let keyword, array push method while also avoids imperative programming.
This section will highlight the main design behind the game state.

Design Rationale

Tetromino Storage:

There are three Tetromino in the game:

- currentTetromino : Tetromino controlled by user
- floorTetromino : the stack of Tetromino on the bottom of the canvas, can't be moved
- previewTetromino : the next Tetromino to be controlled by user after stacking

This is done instead of using a matrix because it would be cleaner for manipulating State.

For example:

```
*/
function shiftDownAll (s:State){
  if (!(canEnterTetromino(s, nextTetrominoPos(s.currentTetromino, "down"))) && !canEnterTetromino(s, s.nextTetromino)){ // if game e
    s =  {...s, gameEnd: true, exit: [...s.exit, ...s.currentTetromino.blocks, ...s.nextTetromino.blocks, ...s.floorTetromino.blocks
  }
  else if (!(canEnterTetromino(s, nextTetrominoPos(s.currentTetromino, "down")))){ // if current tetromino cannot move down, pull fr
    s = ({ ...s,
      currentTetromino: s.nextTetromino,
      nextTetromino: createTetromino(s.time),
      floorTetromino: {...s.floorTetromino, blocks: [...s.floorTetromino.blocks, ...s.currentTetromino.blocks]},  // basically forms
      transform: true,
    })
  }
  else{ // move down
    s = ({ ...s,
      currentTetromino: nextTetrominoPos(s.currentTetromino, "down"),
      transform: false
    })
    s = removeFullRows(s)
  }

  s = <State>{...s, tetromino:canEnterTetromino(s, nextTetrominoPos(s.floorTetromino, "down"))? nextTetrominoPos(s.floorTetromino, "
  return s
```

The function above is ran at every tick (a sub-function for Tick) used to process shifting down of Tetromino. In the function, there are three processes being handled:

1. gameEnd
2. stacking – just concatenate currentTetromino.blocks to floor.blocks.
3. move down – update the position for each block within the Tetromino.

Tetromino movement:

It is also easier to handle Tetromino movement. Tetromino can be moved if there are no existing Tetromino at that same place, and it is within the canvas. Which, again, can be done by just updating Tetromino blocks with a new position using map and reduce functions.

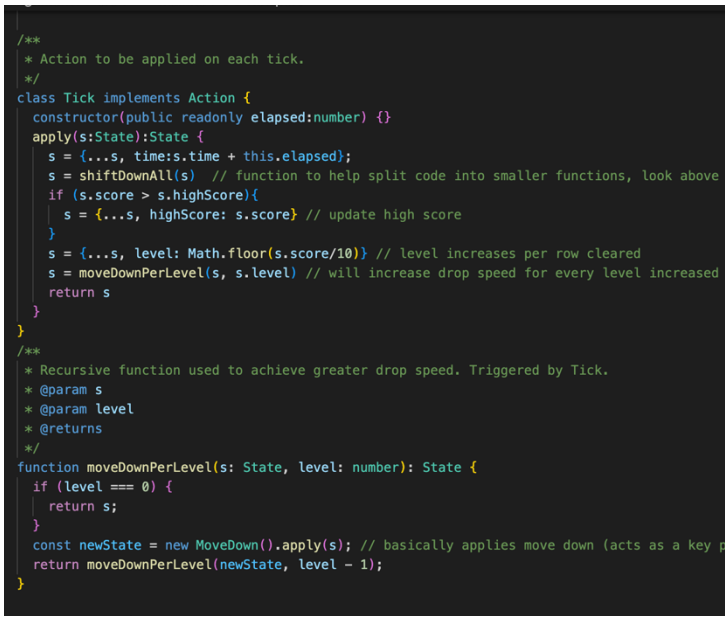
Row Elimination:

As opposed to the other two, row elimination is easier to be done using matrix.
However, I have chosen to prioritize the other two features as they are basic requirement for the game.
Nevertheless, row elimination is still achievable by using array mapping and filtering.

Restart:

A restart action is added which is triggered upon keypress of ‘R’ button. To achieve the restart function, a Boolean condition is added to check if the game has ended, which if it has, then only the state is being manipulated. This process minimizes complexity, which is why it is chosen.

Increasing difficulty:



```
/**
 * Action to be applied on each tick.
 */
class Tick implements Action {
  constructor(public readonly elapsed:number) {}
  apply(s:State):State {
    s = {...s, time:s.time + this.elapsed};
    s = shiftDownAll(s)  // function to help split code into smaller functions, look above
    if (s.score > s.highScore){
      s = {...s, highScore: s.score} // update high score
    }
    s = {...s, level: Math.floor(s.score/10)} // level increases per row cleared
    s = moveDownPerLevel(s, s.level) // will increase drop speed for every level increased
    return s
  }
}
/**
 * Recursive function used to achieve greater drop speed. Triggered by Tick.
 * @param s
 * @param level
 * @returns
 */
function moveDownPerLevel(s: State, level: number): State {
  if (level === 0) {
    return s;
  }
  const newState = new MoveDown().apply(s); // basically applies move down (acts as a key p
  return moveDownPerLevel(newState, level - 1);
}
```

Instead of changing the tick rate and observable stream of the game, which is crucial for the stability of the game, restart is implemented through recursion. Basically, the block controlled by user will move down multiple times. This is done by reusing the MoveDown action class’.