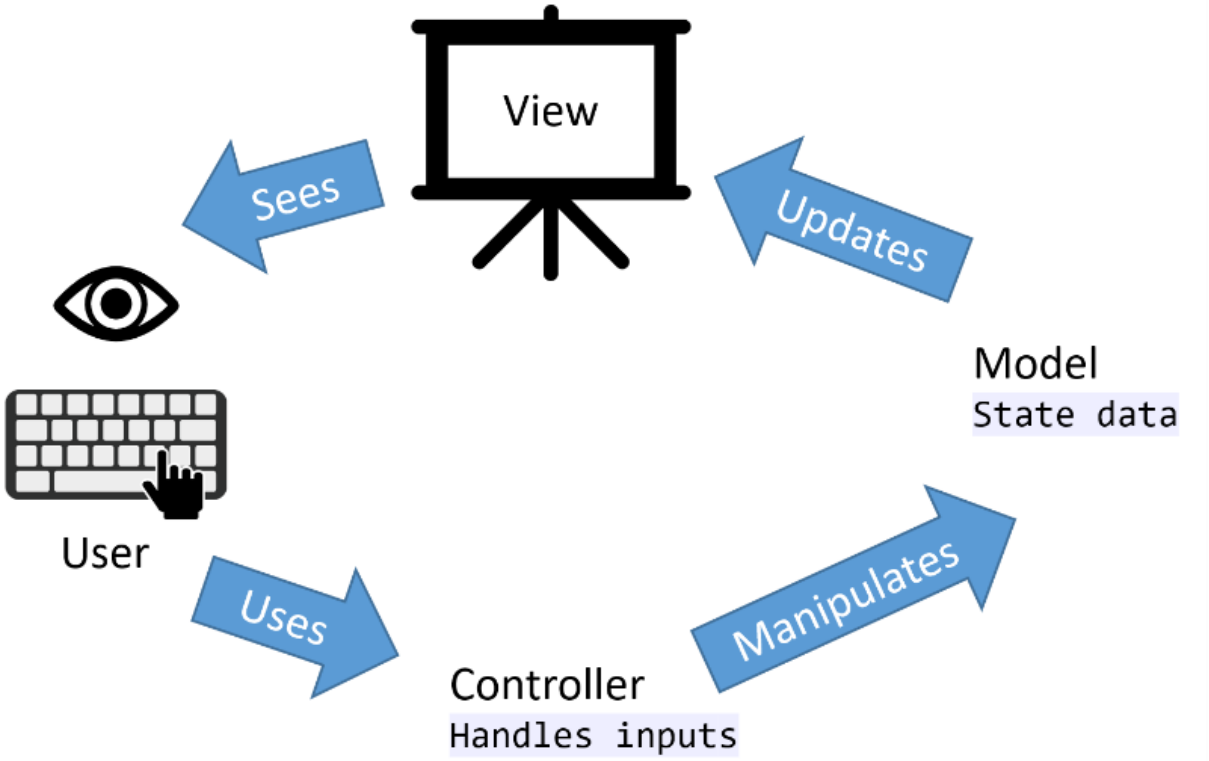
Report for Tetris Game (With Advanced Features)

I implemented the Tetris Game based on the MVC model introduced in the course note:



Controller

Controllers are used to manipulate the gameState. They control the gameState by using Observable operator scan, which allows us to capture this state transformation inside the stream, using a pure function(creates a new output state object with whatever change is required) to transform the state. **This is how the state is managed throughout the game while maintaining purity.**

The controllers for this game implementation include:

Tick()
Handle Current Tetris descending
User Input - *None*

Debuff()
Handle Debuff of the game
Game Level increases when debuffTime becomes 0, which can be delayed by clearing rows.
User Input - *None*

MoveLeft()
Handle Tetris Move Left by default coordinate

User Input - ArrowLeft

MoveRight()
Handle Tetris Move Right by default coordinate
User Input - ArrowRight

MoveDown()
Handle Tetris Move Down by default coordinate
User Input - ArrowDown

Restart()
Trigger game restart (Only allowed to restart when game over)
User Input - Space

ClearLastRow() - Power-ups
Handle power-ups of the game (Only allowed to used not game over)
Clear the last row of tetris, where score and others not affected
User Input - Enter

Rotate()
Handle Tetris rotate **(Nintendo rotation system)**
User Input - ArrowUp

Highlights of the controllers is:

- We use **DEFAULT_TICK_COUNT_THRESHOLD** and **gameLevel** to control the speed of descending tetris rather than directly use the tick observable
- If rowCleared reach a certain threshold => level up => game speed increase
- Game over when at least one column of blocks stack to the top of the grid
- Add power-up quota if clear more than a certain amount of rows at the same time
- Delay debuff time according to number of rows cleared

Model
State is used to update the view through function **updateView()**

Highlights of the game state is:

```
game: number[][];  // game grid
xOffset: number; // x offset of the current Tetris piece
yOffset: number; // y offset of the current Tetris piece
currTetris: TetrisPiece; // current tetris piece
nextTetris: TetrisPiece; // next tetris piece
```

game
We use an 2D array to store the state of the game grid, where each block is an element in the array, which naturally represents the game grid in Tetris (inherently two-dimensional), making it intuitive to work with. Also, it is efficient and straightforward to access and manipulate the blocks.

xOffset, yOffset
We use offset to calculate the position of current moving tetris, which is easy to manipulate with the 2D array game grid. (ex. xOffset = 0, yOffset = 1 indicates the current tetris is at the first element of the second row in gam grid)

currTetris, nextTetris
TetrisPiece consists of blocks, hence I also use 2D arrays to represent it. State of the game stores **both the current tetris and preview tetris.** We store different tetris as constants.

View
View is updated to render what the user will see based on the model.

Highlights of the view is:

- We render the game grid and preview grid with blocks at the start of the game, use show() and hide() to control them (show the block when the grid element is 1, hide otherwise)
- We render the current tetris based on state.game and offset of current tetris
- We also use other attribute in state to update the render of the game board(levelText, debuffTime, etc)

User
Users use input to use the controllers, this is implemented using observables, where we have input streams which in turn map to the respective controllers.

main
All the Model, View, and Controller are tied together with a main game stream **action$.**

The Observable is used to handle **Time-Based Operations, State Management(**The scan operator accumulates state changes over time**)**, **Combining Multiple Sources, User Input, Real-time Updates.**

The observable also allows us to keep well separated state management (model), its input and manipulation (control) and the visuals (view).

Highlights of the **main** is:

tick$: This Observable emits values at regular intervals, simulating a game tick. It's used for tetris descending calculations.

deBuff$: Another interval-based Observable that emits values at regular intervals. It's used for debuff calculations.

Functional Programming
Throughout the whole game, we maintain Functional Programming style by using techniques such as **Immutability, Pure Functions, Higher-Order Functions, Lazy Evaluation, Currying and Partial Application.**

These FP techniques help make FRP code more modular, maintainable, and testable, which is well-suited for handling asynchronous behaviour in applications like this game.