Features of game:

Game has 10x20 grid, with grid scalability.

All 4 possible squares on Tetris.

Descending blocks can be translated smoothly.

Blocks collide and stack correctly.

Full row blocks are cleared, with score, levels and high score updates.

Rotations included for all blocks (SRS rotation system)

Drop down action. (Additional feature)

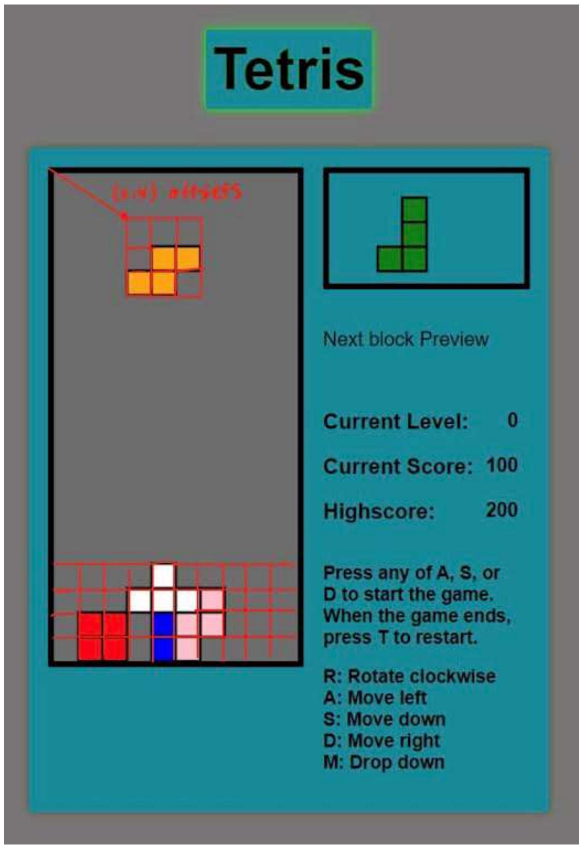
Next Shape preview, and random shape choosing.

Game restart using state management.

Blocks speed increases every level up.

Design decisions:

My states keep an array of unit blocks for the active Tetromino, and the same structure for the entire game grid. Each unit block has their own coordinates with respect to the game grid, and the state keeps an offset pair of coordinates for the active Tetromino.



Current Level: 0

Current Score: 100

Highscore: 200

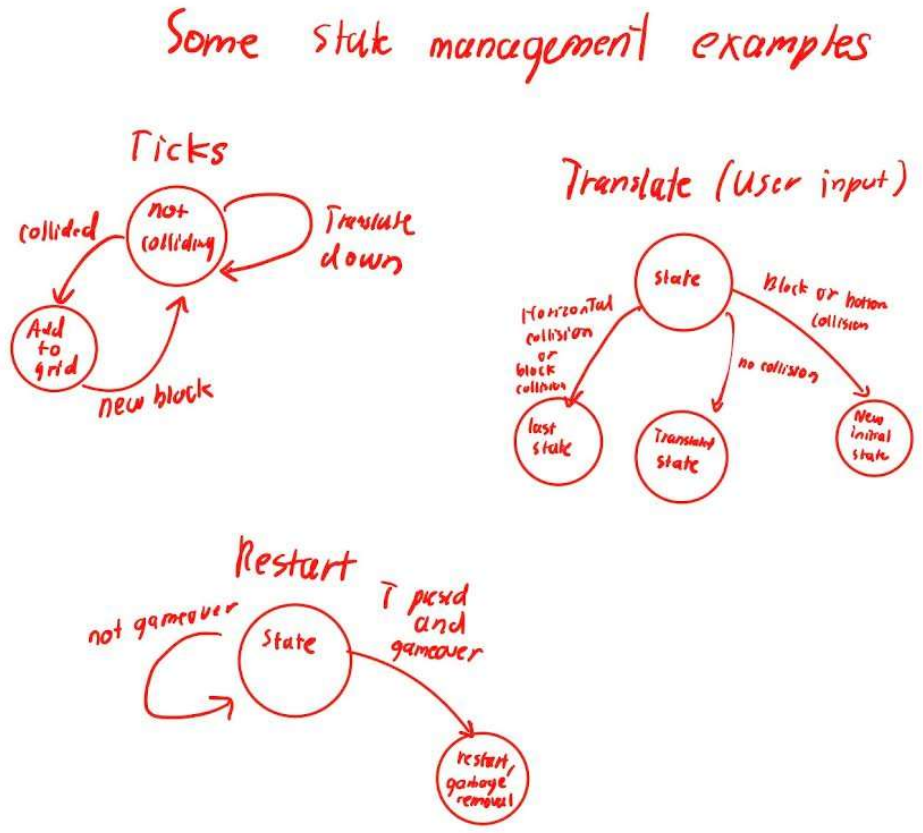
Press any of A, S, or D to start the game. When the game ends, press T to restart.

R: Rotate clockwise
A: Move left
S: Move down
D: Move right
M: Drop down

Used simple immutable array structure so list methods such as map and filter are used to return new arrays of transformed blocks for code purity. For clearing rows, I have remedied this disadvantage by a pure function that converts an array of blocks into a matrix of blocks.

My game design uses class Action instances to handle state management.

All state actions are implemented using a class with an apply method that will return new state instances, this is so we can compartmentalise each action as a separate instance, so we can scan through the state stream and use their respectively apply methods onto the state stream, this allows debugging separate actions much easier, and creating new action instances during code development.



I used an observable stream of actions, game ticks, etc which can be merged, flattened, filtered, etc for more interesting features like smooth handling and asynchronous movement.

FRP + FP style for state management and observables

I used FP + FRP concepts to manage state purely with minimal side effects, all functions return state instances rather than mutation. Spread operator used to duplicate other properties of state, while returning new, modified properties to return new, state instances to maintain purity, this is to avoid side effects such as global or local variable mutations to increase code scalability and maintain referential transparency.

All variables inside functions are declared as constant, and all array types are read only to maintain immutability, this is so debugging is easier when handling asynchronous state transformations. This makes adding new features to the game much easier as it would not affect the functionality of previously added transformations, so the code is scalable and different transformations have independent code behaviour.

HOF concept, combined with currying is mostly used to avoid code repetition, such as extracting leftmost/rightmost blocks in an active Tetromino.

An index extractor HOF is used, which allows me to specify particular x/y axis to create more reusable functions with minimal code repetitions (avoids creating hard-coded functions for similar behaviour) combined with currying for reusability, and to maintain code granularity.

A HOF random number generator is used as an infinite lazy sequence, to be able to extract the next number in a sequence to display the next Tetromino preview.

Most game features including clearing rows, matrix conversion use function/method chaining to handle multiple transformations of state, composing only of filter, map for purity. This is advantageous given that combining pure functions would still produce pure transformations.

More interesting usage of observable

Game ticks: A separate game tick with a custom game Subject observer is used to keep a record of new interval ticks, and distinctUntilChanged rxjs operator is used to detect level changes. SwitchMap is used to flatten new interval subject ticks and ignore previous inner observables.

Asynchronous movement: mergeMap is used to flatten multiple key presses as an inner observable, combined with takeUntil operator to notify when the key is released, this allows smoother gameplay.