# FIT2102 A1 - Functional Reactive Programming

*Report by*: William Keeble (33095043) - wkee0002@student.monash.edu

---

## Introduction

When developing Tetris in FRP, there are a few main concepts that our game engine will rely on. As we cannot have mutability, and must ensure that all of our functions (that do not update the view) have no side effects - are referentially transparent.

To conform to these requirements, we must develop a framework that utilises *state management* in order to update our data throughout the game. Additionally, we use observables from the RxJS library to handle all user input and some dynamic functionality of the game. Hence, we update the view so the user has visual feedback, based on the current data held in our state at each game tick.

## Observables

- We use Observable streams for our movement inputs (A, D, S, Space, W, R) - each of these executes a new Action class object.
- Movement streams are merged into a main game loop - from here we reduce our state.
- The game loop is initialised each time the user presses the “New Game” button - keeping a separation between each game.
- The high score is managed between each new game through the UI. This could be improved by keeping a state outside of each game with the high score, instead of making the check when we update the UI. However, this would increase the complexity considerably so was not necessary for this program.

## Action Classes

- We have Action class objects for key game actions.
- These include Move, Rotate, NewGame and HardDown for user-executed streams, and Tick and AddPiece for objects that are automatically created based on the state of the game.

## State Management

- Action class objects are used to “reduce” the state. Each of these objects manipulate the state (purely - returning a new copy) through their “apply” method.
- Many of the main calculations are done in the Tick class, which is on a fixed increment of 10ms.
- We could improve our state management by keeping single responsibility within the state. If we separate our State object into smaller objects (more than the Piece), it will be much easier to keep our code clean and readable. This would also resolve the issue with high scores, in which it does not always update correctly. However, I felt this extension was too complex.

# Pieces and Cubes

- We have Piece and Cube objects - Pieces are made of Cubes (and other data).
- I made this decision based mainly on the rotation implementation, by being able to keep track of the rotation state outside of the State object itself.
- For both Piece and Cube objects, we use a Type (instead of a class), because we do not need them to be objects that manipulate data themselves. They are the data that we are passing around - this is why we have Actions as classes.
- Random pieces are chosen using an Observable stream which generates a “potential” piece every 1ms, and we only pull from this stream each Tick that we need.
- One improvement that could be made to the random choice is our starting piece - it is currently generated as the same “I” piece every time. By making this random as well, our gameplay would be more interesting.

# Scoring

- For a user to score, they must “delete” rows.
- In our program, we recognise when the current piece has dropped, and if we find any full rows, cubes in those rows are sent to our “exit” array - deleted on our next game tick, and our score is updated depending on how many rows are deleted.
- Our high score is updated after each game is ended - this is held as data in the UI and only updated if the new score is higher. We could make an improvement here through the game state management mentioned above.

# Rotation and Wall Kicks (Additional Feature 1)

- I chose the Super Rotation System (SRS) for my rotations.
- Because of this choice, I decided to implement Wall Kicks as an additional feature.
- Basic rotations are done through their respective Action classes - clockwise or anticlockwise is specified here. We rotate each cube around the “rotation centre” - the cube stored as the first in the Piece’s cube array
- .A “wall kick” in SRS occurs when the user tries to rotate a piece into a space in which it will collide with other cubes/the boundary.
- When this happens, we run through a series of “tests” which are potential offsets for the final position in which the piece can be in a valid position.
- These tests are stored in static data, and referenced in the Rotate action class. This offset data is a known paradigm, and shown below.
- The chosen tests are based on the difference between the starting rotation index and the final rotation index. Note that the current rotation index is initialised at 0, and is updated through the Piece data in the state.

**J, L, S, T, Z Tetromino Offset Data**

| | Offset 1 | Offset 2 | Offset 3 | Offset 4 | Offset 5 |
|---|---|---|---|---|---|
| 0 | true rotation | true rotation | true rotation | true rotation | true rotation |
| 1 | true rotation | ( 1, 0) | ( 1,-1) | ( 0, 2) | ( 1, 2) |
| 2 | true rotation | true rotation | true rotation | true rotation | true rotation |
| 3 | true rotation | (-1, 0) | (-1,-1) | ( 0, 2) | (-1, 2) |

**I Tetromino Offset Data**

| | Offset 1 | Offset 2 | Offset 3 | Offset 4 | Offset 5 |
|---|---|---|---|---|---|
| 0 | true rotation | (-1, 0) | ( 2, 0) | (-1, 0) | ( 2, 0) |
| 1 | (-1, 0) | true rotation | true rotation | ( 0, 1) | ( 0,-2) |
| 2 | (-1, 1) | ( 1, 1) | (-2, 1) | ( 1, 0) | (-2, 0) |
| 3 | ( 0, 1) | ( 0, 1) | ( 0, 1) | ( 0,-1) | ( 0, 2) |

## Hard Drop (Additional Feature 2)

- Another additional feature I decided to implement was the “hard drop”.
- When a user presses the space key, the current piece is dropped to the lowest point possible in the board.
- This calculation is a simple one, being the smallest difference for each cube in the piece, with all the cubes below it.
- The preview for this was the same calculation, and updated in the view by a collection of preview cubes that are rendered without colour, to show the user where they are going to drop.

## Conclusion and Improvements

When considering the overall implementation of Tetris, there are a few improvements that can be made. The overall extensibility of the code could be improved, especially regarding the state management between game rounds.

Currently, the workarounds for calculating high scores across the user’s sessions could use improvement, and implementing a more concrete structure for managing the session state (as opposed to the state of each game) would largely help alleviate this issue.

---

*983 words (600 words + 2 additional features)*