FIT2102 ASSIGNMENT 1

Functional Reactive Programming

GAN SHIJIE | 33585717

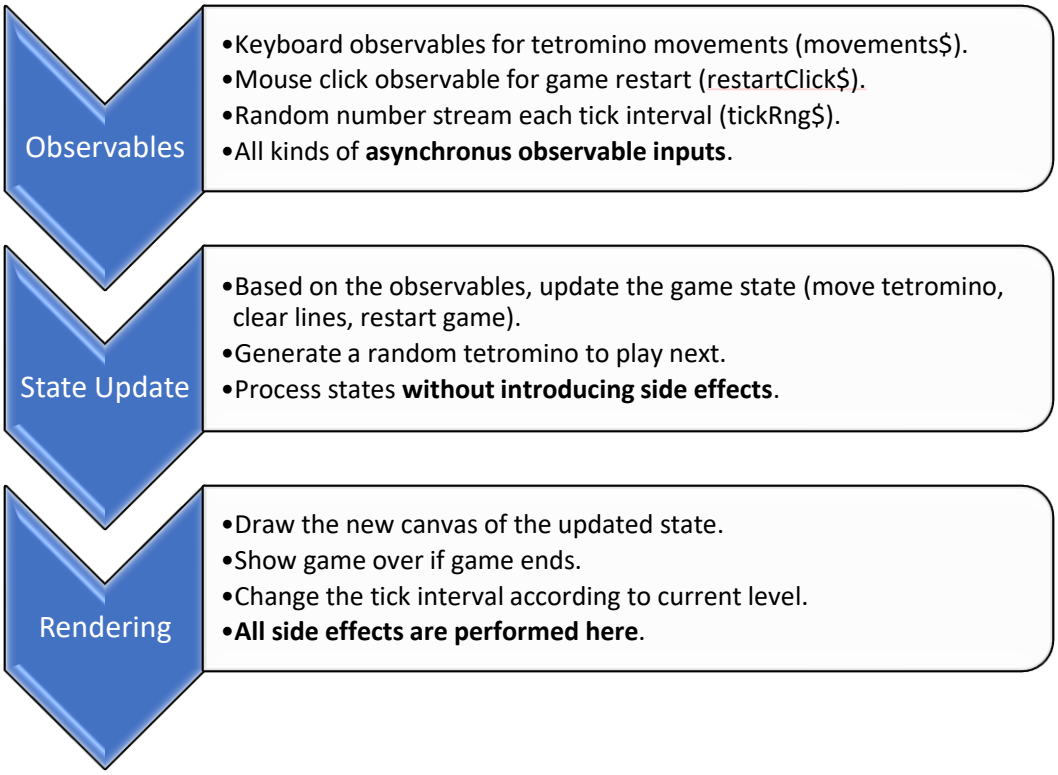
MONASH UNIVERSITY MALAYSIA

1. Overview of Functional Reactive Programming in Tetris

In this Tetris game project, Functional Reactive Programming (FRP) principles was applied to create a responsive and maintainable gaming experience. Key aspects of FRP in this project include the use of observables, maintaining state purity, and expressing game logic declaratively.

Now, let's explore how these principles were put into action in both basic implementations and additional features of the game.

2. Tetris Game Basic Implementations



Game State

- The game state is represented as a "Readonly" object with various properties that describe the current state of the game. These properties include the current tetromino in play, the game score, a two-dimensional grid representing the 10x20 game board, and several other attributes.
- To ensure immutability, the game board is declared as a "ReadonlyArray<ReadonlyArray<number>>" type. This guarantees that every aspect of the game state remains deeply immutable. The same principle applies to the shape of a tetromino. With that, state purity will always be maintained.
- During gameplay, the game state can only be updated by returning a new game state. Mutating the existing game state is prohibited. Updating and returning a new state is achieved through object destructuring methods.

Tetromino Movement

- Basic tetromino movements like left, right, down, and rotation are implemented as keyboard observables. The downward movement every tick is controlled by the interval observable "tick$". These inputs are merged and piped to handle asynchronous user commands.

- To handle possible collisions, the game incorporates tetromino shapes and the Super Rotation System (SRS). Helper functions, such as “isCollision” and “isOutOfBounds”, have been introduced. Valid movements are executed by functions like “moveTetromino” and “rotateTetromino”. These functions do not cause impurity or mutation, instead, they return a new state. The use of separate functions and higher-order functions like “filter” and “some” replaces traditional loops, ensuring a declarative functional programming approach.

Random Tetromino

- To generate random tetrominoes, random numbers are required. To maintain program purity, a stream of random numbers “tickRng$” is created from the “tick$” source using utility functions. A constant seed is introduced for hashing and scaling to eliminate impurities. Using the same seed guarantees that the generated stream of numbers remains consistent.
- As random numbers are generated in sync with each tick, this design ensures a high degree of randomness in the game. New tetrominoes are spawned at different time intervals depending on the game, contributing to the randomness while preserving purity and determinism.

Restart

- The restart feature is triggered by a mouse click observable associated with a dedicated restart button on the canvas.
- Restarting the game simply reverts the state to its initial state, whether the game is over or not. Since a constant seed is used, restarting the game always results in the same initial state, ensuring state purity throughout execution. One exception is the high score attribute, which is carried forward to subsequent games to track the highest score achieved.
- It’s important to note that restarting the game does not involve refreshing the webpage, which would yield a different initial state due to the use of “Math.random()” for seed generation on each file execution.

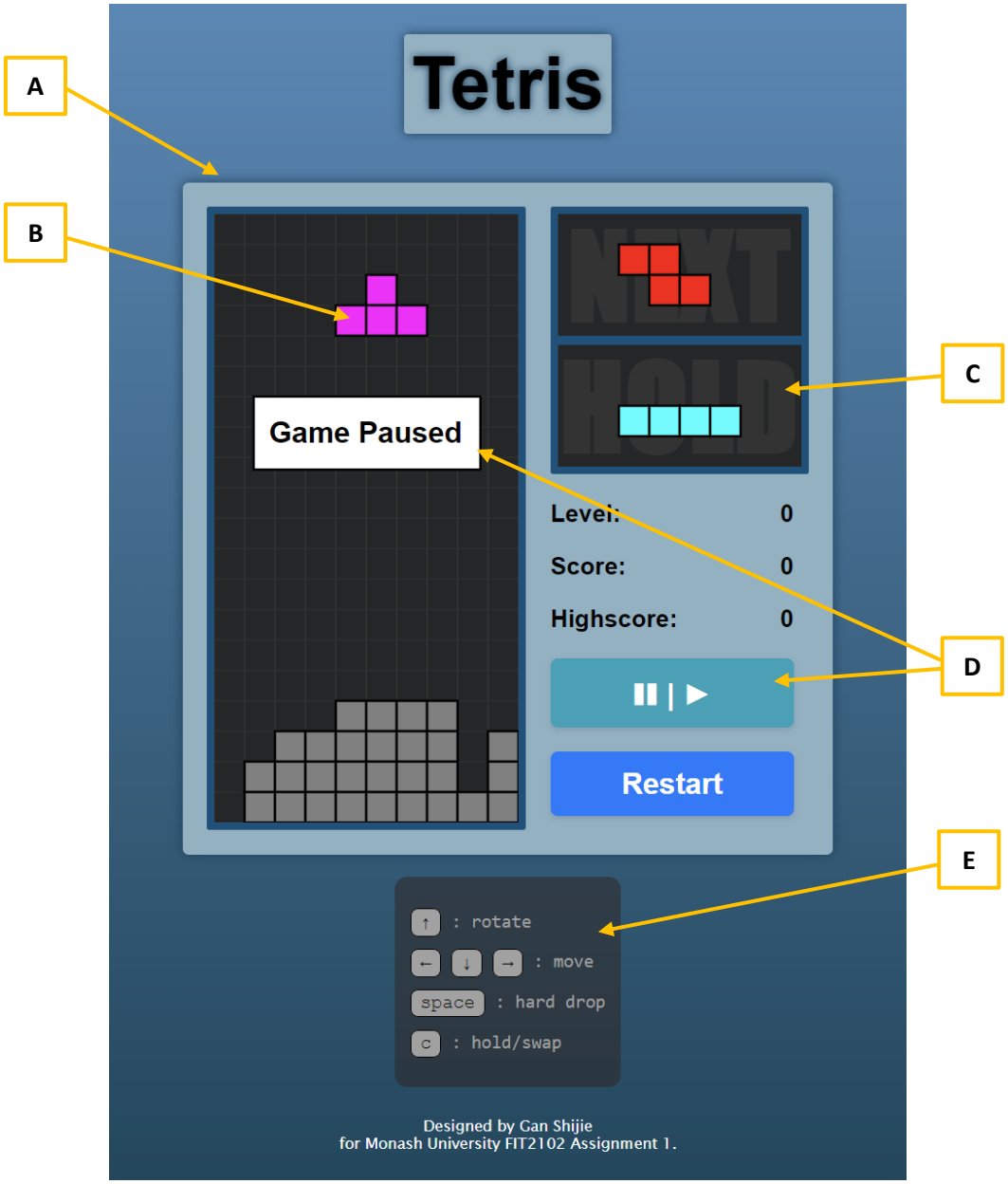
Scoring System and Difficulty Increment

- The scoring system is integrated into the game state. When lines are cleared, the current score increases accordingly, and the high score is updated if exceeded. Additionally, the game’s level increases by one for every ten lines cleared.
- To adjust the game’s difficulty, the tick rate is increased with each level. A “BehaviourSubject” observable is employed to monitor the current game level. By subscribing to this observable using “map” and “switchMap” functions, the tick rate dynamically adapts to the current level, exemplifying functional reactive programming (FRP) principles.
- To ensure that restarting the game will restart the tick rate, a Boolean “gameRestart” is added to the state attribute, to indicate the process of restarting. If the game is restarted, the BehaviourSubject is leveraged to reset the game’s tick rate.

Observable Handling and Rendering

- Observable streams are efficiently managed using the pipe and scan functions, adhering to the principles of functional reactive programming (FRP). A dedicated function, “handleInput”, is employed to process user inputs. This function identifies the type of input and triggers the appropriate state updates by invoking specific functions. The underlying algorithms exclusively return new game states, preserving immutability and preventing unintended side effects.
- The pivotal rendering phase is where side effects take place and where visual changes are presented to the player. The previous game state is systematically cleared from the canvas, ensuring a clean slate for the new state. The rendering process utilises four succinctly tailored rendering functions, resulting in code that is exceptionally clear and comprehensible.

3. Tetris Game Additional Features



A) Enhanced UI Design

- An appealing user interface visualization is a crucial aspect of any enjoyable gaming experience.
- In the files index.html and style.css, extensive edits have been made to introduce new elements and a harmonious colour scheme that elevates the visual aesthetics of the game. Careful attention has been given to colour coordination, ensuring that it complements the overall theme.
- Notably, gridlines have been incorporated onto the game board using the function “generateGridLines”, contributing to a more engaging and immersive gameplay environment. If you examine the preview sections closely, you’ll also notice the addition of “NEXT” and “HOLD” labels in the background, enhancing clarity.
- Game instructions are thoughtfully incorporated into the game’s canvas as well, providing players with clear guidance on how to play.

- Beyond static design elements, dynamic visual effects are triggered when hovering over the pause or restart buttons. These subtle yet effective animations enhance user interactivity and provide a polished feel to the interface.
- Collectively, these design enhancements aim to immerse players in the gaming experience, making it visually appealing and enjoyable.

B) Tetromino Colours Enhancement

- Elevating the Tetris gaming experience is all about embracing vibrant and distinctive tetromino colours.
- Each tetromino shape is thoughtfully associated with a unique and eye-catching colour. This colour assignment is managed by adding a read-only attribute to each tetromino object, based on its shape. While not overly complex, this task involves maintaining colour data for each tetromino shape and ensuring the correct colours are rendered during gameplay. The process of handling colour data and rendering is carefully integrated into each state transition and update.
- To maximize the visual impact and maintain efficient memory usage, a thoughtful approach is employed. Tetrominoes displayed in preview sections retain their original colours, allowing players to easily identify upcoming pieces. In contrast, tetrominoes that have settled onto the gameboard are uniformly coloured in grey. This consistent grey hue signals to the player that these pieces have found their place in the stack, enhancing gameplay clarity.

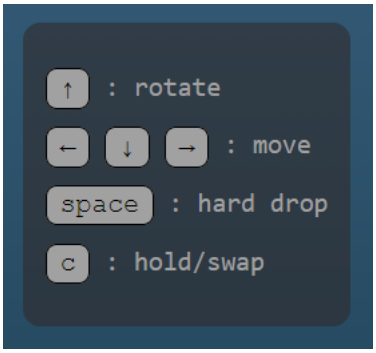
C) Hold Feature

- The addition of the hold feature enhances the gameplay by allowing players to strategically store or swap tetrominoes.
- The hold feature is implemented through a new observable called “hold$”, which listens for user input when the “C” key is pressed. This observable is seamlessly merged with other observables in the game’s state processing pipeline. Implementing the hold feature involves the careful tracking and management of the current and held tetrominoes, along with their state transitions. To facilitate this, a dedicated attribute for the held tetromino is introduced within the game state.
- A noteworthy consideration in the implementation of this observable is the initial state of the game when there is no held tetromino available. In this scenario, triggering the hold feature for the first time requires simultaneous updates to the attributes “currentTetromino”, “nextTetromino”, and “holdTetromino” of the game state. Generating the “nextTetromino” requires a random number, which should be obtained from the previously defined random number stream, “tickRng$”, to maintain state purity and adhere to the functional reactive programming paradigm.
- To address this requirement, the “hold$” observable is zipped together with the “tickRng$” observable to create a new observable, “holdRng$.” This ensures that when the hold feature is triggered for the first time in the game, the “nextTetromino” is generated from the stream of random numbers.
- To maintain code modularity and a declarative coding style, several new small functions, such as “holdState”, “isHold”, and “renderHoldTetromino” are introduced to handle different aspects of the hold feature. This modular approach enhances code readability and maintainability, making it easier to understand and extend the game’s functionality.
- Not to mention that the function “resetPosition” is created to reset both the orientation and position of the tetrominoes being swapped.

D) Pause/Resume Feature

- The pause feature allows players to temporarily halt the game, providing flexibility and control during gameplay.
- The implementation of the pause feature involves the creation of several essential HTML elements, such as the pause button and a corresponding pause message. An observable is created to detect mouse clicks on the dedicated pause button, ensuring a responsive pause/resume mechanism.
- The introduction of the Boolean attribute "gamePause" serves as an important component of the system. It accurately reflects whether the game is currently paused, guiding the state processing and rendering accordingly. When paused, the canvas will display a "Game Paused" message.
- To guarantee a comprehensive game halt in all ways, including aspects like blocking user input and preventing tetromino descent, the "gamePause" Boolean is integrated into every facet of state processing that requires synchronization with the pause feature.
- A dedicated function, "isPause", is thoughtfully implemented to assess mouse click events, determining their execution based on the ongoing game logic. To enhance the user experience, the "blur" function is cleverly applied to buttons, preventing accidental activation of the pause and restart functions when players press the spacebar. This is a small yet significant detail that enhances gameplay control.

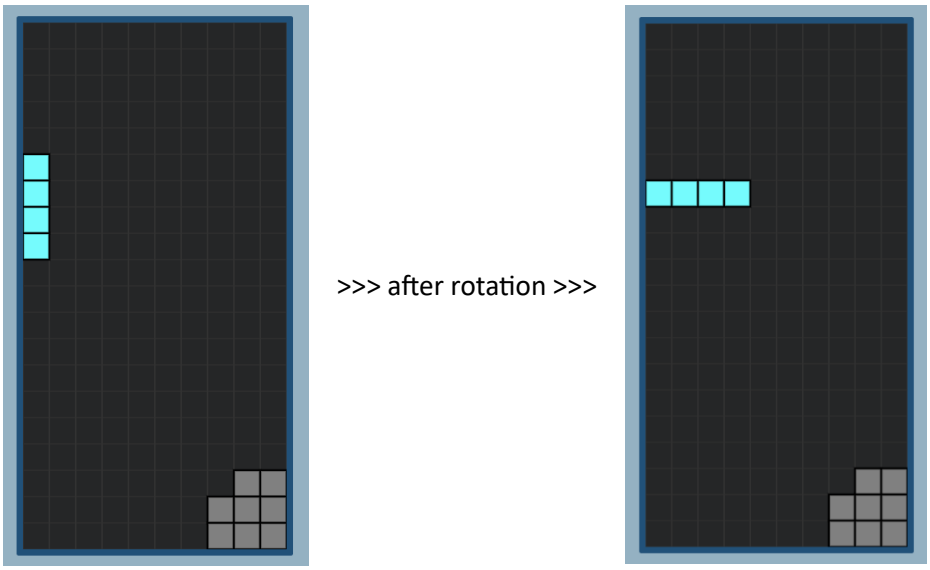
E) Game Control Keys



- In a bid to enhance the game's user experience and make controls more intuitive, the movement controls were switched from the less intuitive WASD keys to the more instinctive arrow keys.
- This adjustment may appear minor in isolation, but it significantly elevates the overall user experience. Arrow keys align more naturally with player expectations, resulting in smoother and more enjoyable gameplay. However, a potential issue arises, arrow keys can also trigger default actions on a webpage, such as scrolling.
- To mitigate this, the "preventDefault" function was applied to the arrow keys. This simple but effective measure ensures that pressing arrow keys won't inadvertently trigger unintended actions on the webpage. The same precaution was taken with the spacebar key.
- This attention to detail exemplifies the thoroughness of the design process, aiming not only to create an entertaining game but also to provide a polished and hassle-free user experience.

F) Wall Kick Feature using Super Rotation System (SRS)

Case 1: Obstructed by wall.



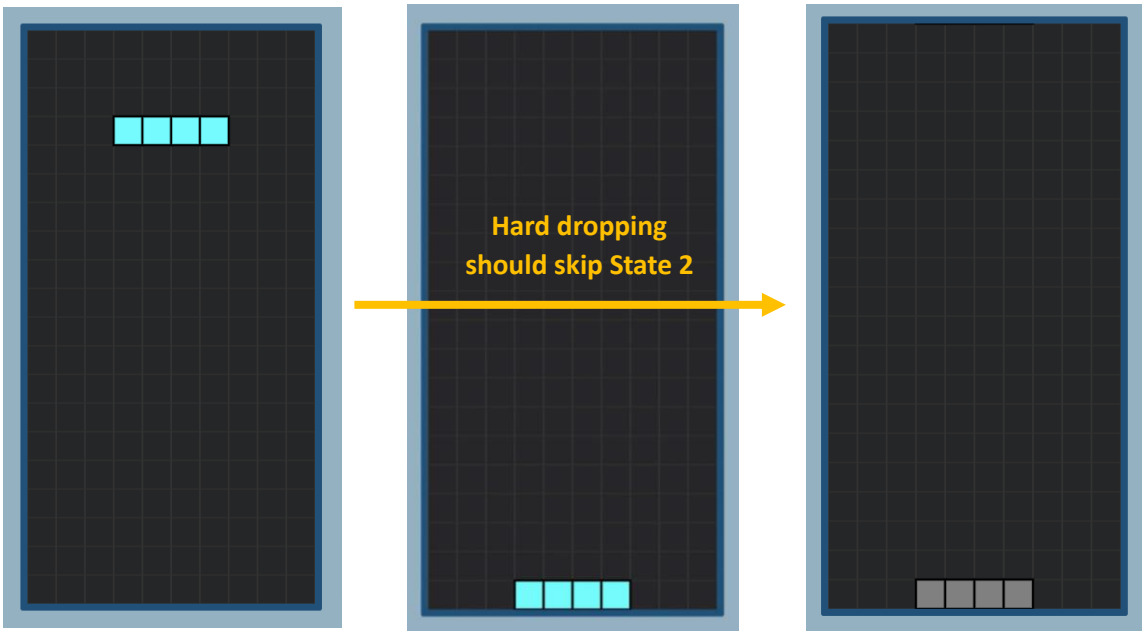
Case 2: Obstructed by floor.

- A critical enhancement to the Tetris game's mechanics was the incorporation of the wall kick feature. This feature significantly improves gameplay by allowing tetrominoes to rotate and adapt when they encounter obstructions, be it a wall, the floor, or other existing blocks.
- The wall kick feature was implemented in strict accordance with the Super Rotation System (SRS), a well-established standard in Tetris game design. The reference from https://tetris.fandom.com/wiki/SRS was followed meticulously to ensure accuracy and adherence to established conventions.
- In this design, players are limited to rotating tetrominoes by 90 degrees clockwise. Consequently, the wall kick rotation system implemented in the game supports only 90-degree rotations. The implementation details can be found in the "wallKickData" dictionary within the

codebase. This dictionary contains the orientation and wall kick rotation logic for each tetromino shape and orientation. It was extensively tested during gameplay to verify its functionality.

- To facilitate this feature's success, several crucial elements were addressed:
  - Wall kick data was created for each orientation of every tetromino shape.
  - The rotation functionality, specifically within the "rotateTetromino" function, was thoroughly reworked. Now, for each rotation, the game checks for the validity of a basic rotation and attempts a wall kick if necessary.
  - To maintain modular functionality and adhere to a declarative coding style, a separate function called "attemptWallKick" was created to control the kick movement. Higher-order functions, such as "find", were used to align with the FRP paradigm.
  - A new attribute, "orientation", was introduced to the tetromino object to accurately track its current orientation. This attribute plays a crucial role in determining when and how a tetromino performs a wall kick attempt. It is updated whenever a rotation occurs.

G) Hard Drop Feature



State 1 (original position) State 2 (lowest position) State 3 (settled)

- The hard drop feature is a fundamental and crucial game control in Tetris.
- The drop feature is executed using a recursive function aptly named "dropOneStepRecursive", which is contained within the "dropTetromino" function. The fundamental logic behind this recursion is to move the tetromino to the lowest available position on the game board.
- However, during implementation, it became evident that this approach did not fully adhere to the hard drop logic in Tetris. It effectively moved the tetromino to the lowest position but did not settle it in place (only transitioned from State 1 to State 2). In this configuration, players still had to wait for a tick event to finalise the tetromino's placement. This created a slight delay and did not provide the immediate hard drop experience where the tetromino instantly reaches the bottom, and the next tetromino spawns (transitioning from State 1 to State 3).
- To address this issue and ensure a seamless hard drop experience, the observable "drop$" created to capture space bar input was zipped together with the existing "tickRng$" to

generate a new zipped observable called “dropRng$”. This innovative approach allows for the simultaneous execution of the drop and tick actions, aligning the gameplay logic more closely with the expected hard drop behaviour.

- The result is a smoother and more satisfying gameplay experience where players can efficiently perform hard drops, significantly enhancing their control over the game.

H) Unit Test

- While unit tests are undeniably a critical component of software development, this project’s unit tests are relatively limited in scope. Due to time constraints and the prioritisation of other development tasks such as code restructuring, readability enhancement, adherence to declarative coding style, and applying functional programming paradigms, only a few unit tests were constructed.
- The available unit tests were crafted to assess the correctness of the codebase. Although only a few tests were developed, they were designed to scrutinize critical functionalities.
- It’s acknowledged that comprehensive unit testing is pivotal for maintaining code quality and stability. Given more time and resources, a more exhaustive suite of tests would have been created to provide a robust validation of the application’s behaviour.