FIT 2102 ASSIGNMENT 1 REPORT – TETRIS

Name: Tan Chun Ling

Student ID: 33402973

TABLE OF CONTENTS

A. Game features

Description

In the game of Tetris, players strive to clear lines by strategically manoeuvring various geometric shapes, known as tetrominoes, as they descend onto the game board. When players successfully complete lines, those lines vanish, earning them points and creating opportunities to fill the newly opened spaces. The game continues until the accumulating lines near the top of the game board, at which point the game concludes.

Game mechanics

	Description	
Block Type	1. Type 0 (yellow) 2. Type I (blue) 3. Type L (pink) 4. Type J (dark blue)	5. Type T (orange) 6. Type S (green) 7. Type Z (Red)
User Input	Arrow Up ↑ : Rotate clockwise Arrow Left ← : Move left Arrow Right →: Move right Arrow Down ↓ : Soft drop	Space bar ␣: Hard drop Key Z: Rotate anti-clockwise Key R: Restart (you can restart any time if you want) press for 2 seconds!
Rotate System	Arika Super Rotation System: including wall kick and floor kick (see D section).	
Scoring System	Original BPS scoring system	
Game state	Score: Initial score: 0 - Earn points by executing a hard drop of the tetromino or by successfully clearing rows. Level: Initial level: 0 - Advance to the next level upon clearing 10 rows. The highest achievable level is 5, and each level enhances the speed at which tetrominos descend. Highscore: Initial highscore: 0 - Your top score will be saved after each round. Strive to surpass your own highest score! Power Ups: (see D section) Initial power up: 5 - Remove the last row. Save your Tetris!	

B. Code Architecture

File separations

	Description
MAIN.TS	This file contains the main code logic and the core game loop.
TYPES.TS	This file contains the common types and type aliases.
UTIL.TS	Util functions
STATE.TS	State processing and transformation
VIEW.TS	Rendering
OBSERVABLE.TS	Function to create observable streams

Action Management

Here are the actions I am employing within my code. I modify the state exclusively in response to user input-driven actions, adhering to the principles of the Model-View-Controller (MVC) architecture and also parametric polymorphism (see FRP section).

	Justification
Tetro	Create random tetromino based on the rng stream.
Tick	Emit one tick every tick rate, the tick rate could be changed depends on the player's level.
Move	Move the tetromino, based on the parameter movement.
MoveMax	Used when hard drop by player, recursively move tetromino to the desired position and add points to the score.
Rotate	Rotate based on the direction. See Pos class for more information in rotate direction.
Restart	Restart when the player press key R.
PowerUp	Remove the last row when player press P.

C. FRP IN MY CODE

PURE FUNCTION AND REFERENTIALLY TRANSPARENT

	Description
Definition	A *pure function*: - has no *side effects*: i.e. it has no effects other than to create a return value; - always produces the same result for the same input.
Why	Pure functions ensure consistent and predictable outcomes by solely relying on their input, devoid of side effects or external dependencies.
Example	handleCollision (state.ts) This function manages collision logic, generating a new state through computations reliant solely on the input state and the new tetromino. It operates without any side effects, exclusively yielding results based on its inputs.
Justification	The majority of my functions exclusively depend on their input to generate new return values, refraining from altering external object values. I prioritize using deep copies to create fresh states instead of modifying existing ones, ensuring the functions consistently yield identical outcomes for identical inputs.

PARAMETRIC POLYMORPHISM

	Description
Definition	Parametric polymorphism refers to a programming concept where functions or types can work with multiple data types in a generic and reusable manner, without specifying the specific types in advance.
Why	I use parametric polymorphism to create more versatile and reusable code that can handle various data types without requiring separate implementations for each type.
Example	reduceState (state.ts) The "reduceState" function demonstrates adherence to the principle of parametric polymorphism by applying a wide range of "Action" instances, accommodating diverse actions within the same function structure.

REUSABLE CODE

	Description
Definition	Reusable code involves functions, classes, and modules that package distinct behaviours, enabling their use across various sections of your game logic without requiring extensive alterations.
Why	Using reusable code enhances development through modular organization, simplifying game creation, maintenance, and adaptation by leveraging existing logic and minimizing redundancy.
Example	moveTetro (state.ts): This function is utilized to relocate the current tetromino to a new position, and it is repurposed across various instances like tick, handle collision, and the MoveMax action class, effectively preventing redundant code and enhancing code manageability.
Justification	I made lots of functions in my code that can be reused whenever I need them again. This helps me avoid writing the same code over and over. For example, I took out collisionX and collisionY from the handle collision part and used them again when I was working on the rotate action. This way, things run smoother and I don't have to repeat stuff.

OBSERVABLE

	Description
Definition	An observable is a data stream that emits values over time and allows you to react to those values using various operators and subscriptions.
Why	Observables are used to handle asynchronous and event-driven programming more effectively by providing a consistent way to manage and respond to data streams.
Example	Create Observable: interval, fromEvent, merge Observable Operators: map, filter, scan, exhaustMap (see D section) Observable method: pipe, subscribe

D. ADDITIONAL

I've incorporated several additional requirements into my code. While there might still be some undiscovered bugs, I've successfully implemented a few small enhancements to my game.

BACKGROUND AND TEMPLATE

I've introduced a background video and transformed the template into a sleek dark grey theme. The inspiration for this template draws from the renowned game 'tetris.io.' The following files showcase the modifications I've made:

File	Description
style.css	- Adding #background video and #backgroundOverlay - Change the colour of the theme - Add restart button
index.html	- Add restart button - Add video and overlay

PREVIEW THE POSITION AFTER DROPPING

Previewing the tetromino's position after a drop has become a common feature in modern Tetris games. I incorporated a similar concept into my code, leveraging the code structure used for tetromino hard drop.

Function	Description
moveAmount	- Check the desired movement
State	- Add the tetromino to the current state
previewHardDrop (inside render)	- Append block to the svg

WALL KICK AND FLOOR KICK

In order to address the problem of certain orientations being unrottable, I implemented a coding approach inspired by wall kicks and floor kicks. Drawing from the Arika Super Rotation System, I adopted its rotation principles, which are an extension of the Super Rotation System with additional functionalities like wall kicks and floor kicks. Notably, in this system, the I block is capable of undergoing a floor kick even when adjacent blocks are present—unlike conventional Tetris, where the I block couldn't be rotated in such circumstances.

Kick	Description
Wall Kick	Used for *J*, *L*, *S*, *T* and *Z* tetrominoes
Floor Kick	Used for *I* tetromino

POWER UP

To assist players in overcoming situations where the game grid is filled with full rows, I've incorporated an additional gameplay feature known as "Power Up." This concept aligns with the principles of Functional Reactive Programming (FRP), as Power Up is implemented as a subclass of the broader "Action" category in my design. Players are rewarded with power ups as they progress through levels, and I've also included extra power ups at the beginning of each game, granting players an initial allocation of five power ups for use within a single game.

Function	Description
removingRow	- The class I employed for checking full rows also follows the "Don't Repeat Yourself" (DRY) principle
PowerUp	- Create Power Up action to return a new state after applying this action

RESTART

I used some observables that are not covered in the class, to implement the game's restart feature. By holding the "R" key for 2 seconds, players can initiate a game restart at any moment. This functionality is reminiscent of Tetris.io, a well-known Tetris game that also requires players to hold a key for several seconds to restart the game.

Observable	Description
exhauseMap	A function that returns an Observable containing projected Observables of each item of the source, ignoring projected Observables that start before their preceding Observable has completed.

E. Bugs and Improvement

Instant Replay

I attempted to implement instant replay using a Replay Subject, but unfortunately, it didn't yield the desired results. The issue stems from the fact that the Replay Subject emits all values from its buffer simultaneously, disregarding the actual chronological sequence. Moreover, I encountered challenges in unsubscribing the source$, leading to the persistence of the source$ stream even during replay, which was unintended.

** I have commented the Instant Replay code in my main file **