FIT2102: Assignment 1 Report

Emily Samantha Zarry (32558945)

Code Summary and Highlights

- SVG canvas elements are set up for the game and get updated in each game tick to display the current state of the game
- The game state, which holds information about the current score, high score, level, moving shape, next shape, and the block-filled grid is initialised and gets updated through the state transducer by detecting the appropriate action being taken
- The updates are implemented through observable streams, which capture key events for moving, rotating, and restarting, and the game clock
- The code also handles the logic for collision detection, shape movement, row clearing, and level progression, all using functional programming where variables remain unmutated
- To implement increasing speed when leveling up, the gameClock$ observable is made to have a quick interval, while in the tick function, we only update the state if the interval counter is divisible by the fall rate. The fall rate is adjusted according to the level, making the speed of the game proportional to the current level.

Design Decisions and Justification

- Separation of concerns
    - Functions and components in the code adhere to the Single Responsibility Principle, each having specific roles and responsibilities. This ensures code modularity and makes the logic easier to understand.
    - The overall architecture is separated into a Model-View-Controller pattern. This allows a clear separation of tasks: the model handles game state and logic, the view handles rendering, and the controller handles user input.
- Observable streams
    - Observable streams are used because they provide a composable way to handle multiple data streams and different types of real-time events and apply changes throughout the game.
- Immutable state
    - Having an immutable state simplifies state management and prevents unintended side effects. In a complex game, this allows consistent behaviour and easier debugging.

Functional Reactive Programming Style and Usage of Observables

The code follows the Functional Reactive Programming style by modeling events and behaviors as observable streams. Some notable aspects of FRP in the code:

- Event streams
    - Key event observables are used to track user input. This allows the code to react to user input in a declarative and functional manner.
- Merging streams
    - The merge function is used to combine the observables into one so the transducer can correctly update the state as the game progresses.
    - The zip function is used to merge the random number streams used to generate the random shapes
- State management through observables
    - The game state is updated by subscribing to the merged observable streams (source$). Each event emitted results in a new state, making state management reactive.

State Management and Maintaining Purity

The code maintains purity by following functional programming principles. The key aspects include:

- Immutability
    - The game state is continually updated without mutating variables by using spread operators (e.g. {...s}) to create new objects and modify necessary fields. This prevents accidental mutation of the state.
- Pure reduce function
    - The reduceState transducer is a pure reduce function that takes the current state and an action and returns a new state with appropriate changes. It does not perform side effects. This ensures that state changes in the game are predictable.
- Pure helper functions
    - Functions such as tick, moveShape, isCollision, exceedsTop, handleFilledRows, rotateShape, etc. all use RxJs functions that maintain purity
    - Some functions that were used include: map, reduce, filter, some

Maintaining purity in state management allows for code predictability as it will produce consistent output for every input, as well as ease testability and debugging.

Description of the usage of observable beyond simple input

- Game loop
    - The game loop is implemented as an observable (gameClock$) that emits events at regular intervals. This emitted event represents one tick in the game where the active shape will fall until it hits the bottom of the grid or collides with a fixed block.
- Generating random numbers
    - The game uses a pseudo-random number generator. Observables (xRandom$, shapeIndexRandom$, rotationIndexRandom$) are used to

generate random numbers based on the game clock. This ensures random elements are available in each tick of the game.

- Signaling game over
    - The gameOverSignal$ observable is emitted when the end-game condition is met. The observable is simply a new Subject() which allows it to be called manually as opposed to other observables in the code which is either emitted consistently (gameClock$) or waits for a key event.
    - This observable acts as the source observable for gameOver$, which returns a new GameOver() action. When this action is passed to the state transducer, the state is updated to display the game over message and prompts the user to restart by pressing the R key.

Additional Features

Saving moving shape

- This feature allows players to save the current moving shape by pressing the W key.
- It is implemented by creating an observable based on the W key press. When emitted, it creates a new SaveShape object and when passed into the reduceState function, the updated state that is returned will have the current moving shape saved as the saved shape, and vice versa.