Tetris Game Report

Introduction

This report is for implementing a Tetris game using FRP principles and RxJS observables. The game is using TypeScript and SVG to do rendering. The report will provide an overview of the code's structure, design decisions, adherence to FRP, state management, and the usage of observables.

Code Overview

The code implements a Tetris game, featuring the following components:

- Game Logic: The game is implemented using functions and classes, with an emphasis on functional programming style. The code manages Tetris shapes spawning and rotation, collision detection, scoring, and user input.

- SVG Rendering: SVG is used to render the game elements on the canvas. Functions like clearCanvas() and render() handle the rendering of Tetris shapes and the game grid.

- User Input: User input is done by using RxJS observables. Key presses are captured and processed using observables.

- State Management: The game state is managed using immutable data structures. The tick function is responsible for advancing the game state in response to time intervals, while other functions handle user input and Tetris shape position and movement.

- Scoring and Levels: The game keeps track of the player's score, cleared lines, and difficulty level. The speed of Tetris piece movement increases based on the number of cleared lines.

FRP Style and Observable Usage

- Observable Event Handling: User input, such as key presses, is captured and processed using observables. For example, fromKey creates observables for specific key events, and merge combines them to handle multiple inputs concurrently.

- Interval-based Updates: The game's main loop is driven by observable intervals (tick$). It periodically emits values that are used to advance the game state. This approach decouples time-based updates from the rest of the game logic.

- State Management with Scan: The scan operator is used to manage the game state over time. It accumulates state changes, ensuring that the state remains immutable.

- Dynamic Difficulty: The game introduces dynamic difficulty levels based on the number of cleared lines. This is achieved by checking the number of cleared lines in the autoMoveDown$ observable and increasing the speed.

State Management and Purity

- The game state is stored in a State object, and changes are made through pure functions like tick and scan. This ensures that state changes are predictable and do not have side effects.

- Tetris shapes are represented as immutable arrays, and their transformations (rotation and movement) are handled through pure functions. This prevents accidental error.

- The game grid, which represents the positions of landed shapes, is updated immutably in the updateGameGrid function.

- The game's rendering functions render() are also side-effect-free, as they create new SVG elements rather than modifying existing ones.

Additional Features

- High Score: The code includes a high score feature that keeps the player's highest score. The high score is updated and displayed on the screen.

- Game Over Screen: A game over screen is displayed when the game ends, indicating that the player's session has concluded.

- Restart Game: The code can instantly replay the game at any time player want whether the game is end for not.

Conclusion

To conclude, this Tetris game code follows the principles of FRP, using RxJS Observables to manage user input, time-based updates, and state changes. Code is always kept pure and immutable to ensure predictable and reliable behavior. Additional features such as high scores, game overs and instant restarts are help to improve the gaming experience.