FIT2102 Documentation Report

Kevin Liu

Student ID: 29685699

Introduction

FIT2102 Weeks 1 to 6 introduced functional reactive programming in Typescript. This was further explored in the first assignment, Tetris. To reinforce students' learning, code was strictly to be implemented in pure code and functional reactive programming; a programming paradigm that strictly deals with asynchronous data or event streams such as mouse clicks or typing. This report aims to further expand upon this programming paradigm, and exemplify the solidified understanding of FRP knowledge garnered across the weeks.

In this replication of Tetris, the game has been implemented with extensive functionality. Of which include, square blocks moving down from the top beginning at a height above the canvas to mimic the game on https://tetr.io. This was coupled with blocks being able to be controlled by the user using 'WASD' and 'Space', 'W' being highlighting the paramount importance of the rotation function, and 'Space' being the hard drop feature. This implementation seamlessly works with game ending on full columns alongside, scores, stacking, row elimination, random next shape based on FIT2102 Week 4's tutorial implementation of the RNG function, difficulty increase with every 2 lines cleared (for ease of use) as well as restart and all block types and rotation.

Design Decisions and Justifications

The use of FRP allows purity and side effects to be avoided. This clear segregation between logic and effects on the DOM allows for easier testing and debugging, as the game's logic does not play a direct influence in manipulating the DOM. This game state is also maintained in a single state object which includes the fall blocks, list of blocks, score and level allowing easier management and updating of the game through the game's progression. This coupled with observables for event handling to create a stream of game actions based on 'WASD' and 'Space' as well as time intervals allows for a more declarative approach to handling events and helps manage the asynchronous nature of user input and ticks.

State Management

State Management is quintessential in the development of game progression, as it includes the position of falling blocks, arrangement of the blocks, score, high score and level. The state of the game encapsulated by s of type State holds all the necessary information to render the game at any given moment. This state is updated via pure functions determined by the input. Exemplified through the **tick** function, **s** is taken as input and returns a new state object with updated properties, checking for collisions, and so forth without input mutation or external variables.

Observables for Event Handling

Expanding upon observables for event handling, it is evident RxJs observables are used to handle keyboard inputs and game times. Accentuated by the use of the **input$** observable, merged streams of keyboard events in a single stream of game actions reveal the underlying utility of observables in the case of event handling. Similarly, **source$** couples **input$** and **tick$** observables and potentiates the stream of game actions to apply them to the game state. The **scan** operator accumulates the actions and applies them correspondingly to the game state akin to the **reduce** method for arrays ensuring game actions are applied in the order they are received as well as consistency of the game state.

By using pure functions for state updates and observables for event handling, the game logic ensures determinism and voids the game of side effects. Allowing the code to be understood, tested and debugged. The approach and considerations taken into stride in the development of the game ensures game determinism, no side effects and is simplistic in understanding, testing and debugging.