FIT2102 Programming paradigms

Assignment 1 Report

Name: Khor Ying Shan

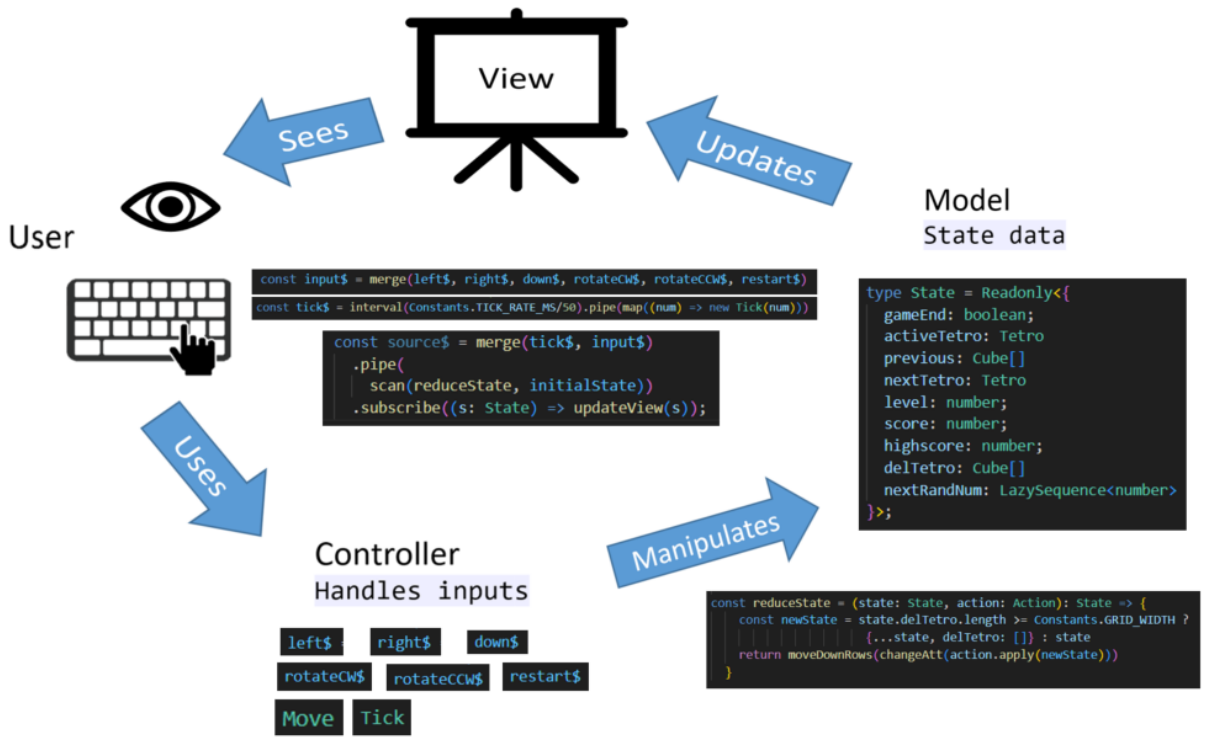
Student ID: 33410488

Introduction

This is a report that is intended to demonstrate my theoretical understanding of functional reactive programming(FRP), and highlight design decisions of my Assignment 1.

Model View Controller (MVC) Architecture

The diagram demonstrates my adherence to Functional Reactive Programming (FRP) principles in structuring my Tetris game. It showcases the separation of state management, input handling, data manipulation, and visualization, all interconnected through the use of Observables. This approach draws inspiration from FRP Asteroids materials.



```
const input$ = merge(left$, right$, down$, rotateCW$, rotateCCW$, restart$)
```

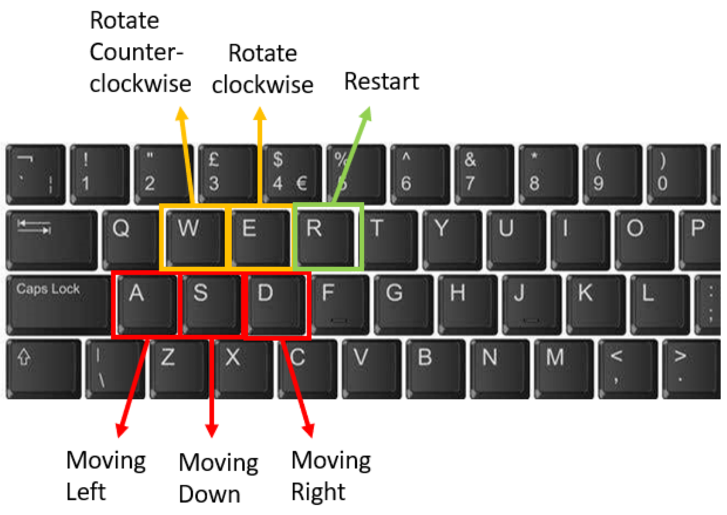
```
const tick$ = interval(Constants.TICK_RATE_MS/50).pipe(map((num) => new Tick(num)))
```

```
const source$ = merge(tick$, input$)
  .pipe(
    scan(reduceState, initialState))
  .subscribe((s: State) => updateView(s));
```

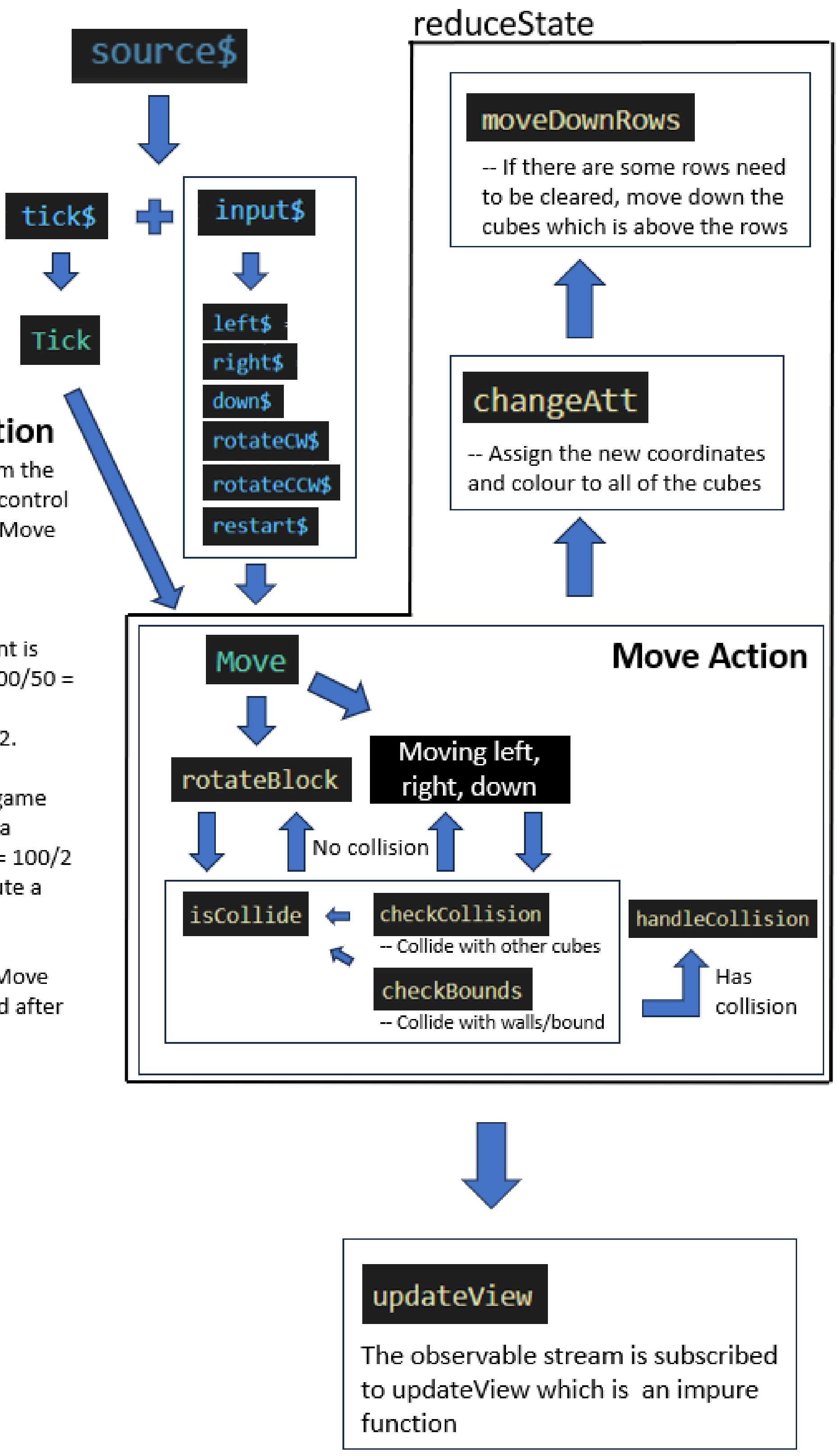
```
type State = Readonly<{
  gameEnd: boolean;
  activeTetro: Tetro
  previous: Cube[]
  nextTetro: Tetro
  level: number;
  score: number;
  highscore: number;
  delTetro: Cube[]
  nextRandNum: LazySequence<number>
}>;
```

```
const reduceState = (state: State, action: Action): State => {
    const newState = state.delTetro.length >= Constants.GRID_WIDTH ?
                     {...state, delTetro: []} : state
    return moveDownRows(changeAtt(action.apply(newState)))
  }
```

Basic Game Instruction



Flow of design


reduceState
source$
tick$
input$
Tick
left$
right$
down$
rotateCW$
rotateCCW$
restart$
moveDownRows
-- If there are some rows need to be cleared, move down the cubes which is above the rows
changeAtt
-- Assign the new coordinates and colour to all of the cubes
Move
Move Action
rotateBlock
Moving left, right, down
No collision
isCollide
checkCollision
-- Collide with other cubes
checkBounds
-- Collide with walls/bound
handleCollision
Has collision
updateView
The observable stream is subscribed to updateView which is an impure function

Tick Action

Use the tick count from the observable stream to control the time to execute a Move action

For example:
Initially every tick count is executed after each 500/50 = 100ms.
And the game level is 2.

In Tick, based on the game level, if a tick count is a multiple of 100/level = 100/2 = 50, then it will execute a Move action.

In other words, each Move action will be executed after each 50ms

Observables & FRP

In this game, I efficiently utilize observables to encapsulate standard asynchronous actions.

I've established two observables:

- One for keyboard events(keypress) listening to specific keys (KeyA, KeyD, KeyS, KeyR, KeyE, KeyW) to control tetromino movement. These individual observables are then combined into a single observable stream called input$ using the merge() method.

- An interval observable, tick$, is generated by interval(), which produces a continuous stream of increasing numbers at a 100ms interval. This interval stream can be further manipulated through piped transformations and is later used to regulate the emission of Move actions within the game.

The keyboardEvent and interval observables are combined into a single observable stream, passed through a pure function called 'reduceState', and then subscribed to perform state updates and tetromino movements, by setting the attributes. The observables remain pure because the scan operator captures state transformations using a pure function to create new output state objects with the necessary changes.

Additionally, the observables promote the creation of reusable data processing pipelines. Once it defines a sequence of operations on an observable stream, it can be reused in multiple places within the game.

FRP manages the game state in this game. Interfaces such as 'State', 'Tetro' and 'Cube' encapsulate object behaviours, while a common 'Actions' interface defines actions with a pure 'apply' method, which takes a previous state and produces a new state.

We begin by defining a 'State' interface with Readonly members and store the initial state in a const variable conforming to this interface.

Within the state reduction process, each action modifies the state using pure functions and returns a new state after application. These functions transform the state purely, always yielding an unchanged state or a new modified state.

There is an example: How I implement handling collision

- Combined all existing cubes from active tetromino and previous cubes via concat() to produce a new array.

- Find the rows of cubes that are going to be cleared with a curry function that uses map() and filter() to get the cubes that are in the same rows, and later on select the rows with full cubes in the row with filter(). The curry function avoids passing the same variable multiple times, and helps to create a higher order function.
- Get the cubes that should be stayed and also the cubes to be cleared via curry function again, involving reduce(), filter(), concat(), to avoid side effects.
- Check if any cube reaches the top of the y bound, to determine should the game end via reduce().
- Then, create a new state with updated details instead of modifying the original state in order to maintain the purity of the function.
- When creating a new tetromino, a random number is used. It is created by a lazy evaluation, creating a lazy sequence of random numbers with an initial seed value as an argument. This quality ensures their purity by adhering to referential transparency, which signifies that when provided with the same input, they consistently generate the same output.

These functions and operators maintain purity by producing new outputs instead of altering input data. They improve code by promoting composability and reusability, enabling the decomposition of complex tasks into smaller, reusable building blocks of code.