Monash University

FIT2102 Assignment 1 Report

Tetris

Kok Tim Ming

32619138

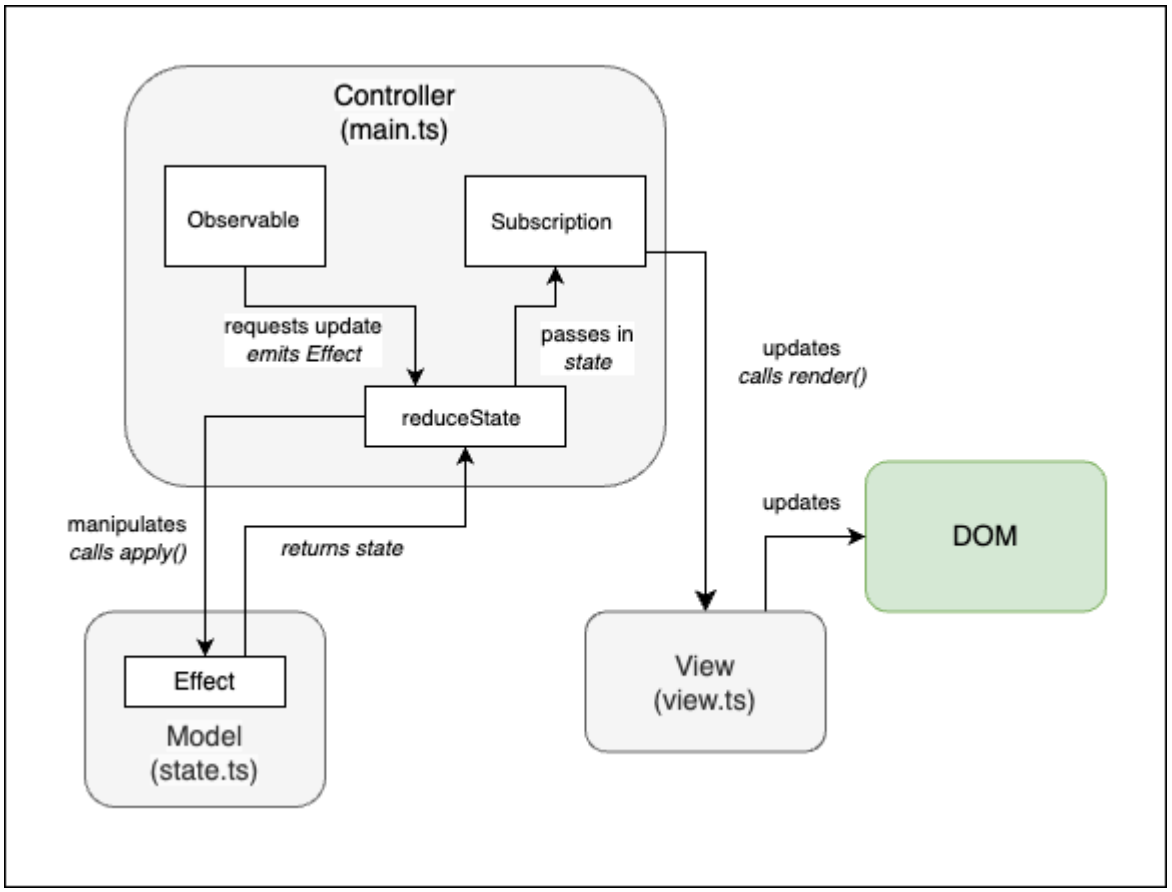
This report outlines the development of Tetris in RxJS, focusing on the utilization of functional reactive programming principles, observables, state management, and design decisions. It provides brief insights into the pipeline and implementation of key game features, such as game tick, rotation, 7-bag system, wall-kick, controls and more.

Table of Contents

Pipeline **3**
Observables **3**
interval (Game-tick) 3
fromEvent (Controls) 4
fromKeyNoRepeat 4
fromHold 4
Pausing 4
State Management & Functional Programming **4**
Design Decisions **5**
Framework 5
Maintenance 5
Functionality **5**
Game-tick 5
Locking 5
Lock Delay 6
Ghost & Hard Drop 6
Hold 6
Rotation & Wallkick 6
Additional Features **7**
Tetris 7-bag 7
Pause 7
Wall-kick 7

Pipeline

Image below demonstrates a high-level overview of the data flow through the application.



Observables

Two types of top-level observables interval (game-tick) and fromEvent (controls) were used. fromEvent has two wrapper functions, fromKeyNoRepeat and fromHold. Each emission from the observables (except pause$) maps out a specified Effect for the state, which is applied (Effect.apply() interface) to the game state with a state transducer (reduceState).

interval (Game-tick)

Used for game-tick. Emits a Tick on every interval to the Model (state.ts).

fromEvent (Controls)

fromKeyNoRepeat

A wrapper for fromEvent that emits specified non-repeatable keystrokes. This was created to be used for rotations and Hard Drops, as regular repeatable keystrokes would be undesirable.

fromHold

A wrapper for fromEvent (and fromKeyNoRepeat), which utilises timer, startWith and takeUntil to implement DAS/autorepeat (or not) controls. This was created for movement keys (DAS) and SoftDrop (autorepeat, no delay). Inner observables from the timer (by map) are merged into output observable with mergeAll().

Pausing

Game pausing is done with switchMap() in rxjs. Each specified key actuation toggles between paused and !paused which is accumulated with scan() in the observable pause$. The emissions of pause$ captured by combineLatestWith() are used to toggle the Tick interval with switchMap(), where the time elapsed is purely and manually accumulated within the observable with scan() to adhere to FRP principles. Manual accumulation of time elapsed is required to preserve the elapsed time, because switchMap resets the interval on each toggle.
While pausing could be implemented with states, observables were preferred for many reasons:

i) Cleaner & more declarative code (SoC, should be the Controller's job in MVC)
ii) Future reusability
iii) No need for extra state logic (less intra-file coupling)

State Management & Functional Programming

The game state is kept immutable throughout with const and readonly types. States are updated by classes that implement the interface Effect, which is emitted by observables. Effect enforces the implementation of the pure apply() method: it takes in a state and returns a new state with the desired effect defined by the implementing class through polymorphism, eliminating the need for using multiple "instanceof" checks. State updates are then asynchronously accumulated with the state transducer and the result is passed on to another Effect.

Besides having the state being immutable, all functions relating to state (besides view.ts) are also pure and are ensured to have a return value, i.e., all are expected to have deterministic behaviours, have no side effects and are referentially transparent. Enforcing return values also promotes functional composition.

FP helpers were created to leverage FP's nature of brevity and purity: pipe(), flip(), ifElse(), not(). While function composition (e.g. a(b(c()))) was possible, the combination of pipe(), other FP helpers and curried functions was favoured for better readability (callback hell). In the code, the functionality of pipe() and other FP helpers was leveraged by splitting/decomposing functions into smaller ones, allowing the apply() function to concisely indicate the potential applied effects.

Design Decisions

Framework

A matrix (or Grid) was used to facilitate game logic, as tetris in its nature is discretely defined (i.e., a cell represents a mino). Thus, when tetrominoes are paired with a locator (Pos), relative rotations and translations to the playfield can be discretely calculated with precision to map the index of the tetromino's individual minoes in the playfield.

Each cell contains an object of filled {boolean} and color {RGBA}, to allow future special tetrominos to constitute of multiple colors instead of just one.

Maintenance

The project files were separated into game logic, state management, and rendering, following MVC architecture to promote code maintainability and development. Classes like Pos, Grid, PlayField, Tetromino and TetrominoBagFactory were preferred over traditional javascript objects for readability. Methods of each Class in state.ts were made to be modular, with each method handling only related responsibilities to keep the *apply()* method concise.

Functionality

The following key functionalities are achieved with the pure apply() method in Effect interface.

Game-tick

The game makes time-based calculations for gravity & locking (not frames) based on elapsed time provided by the observable to Tick (state.ts).

Locking

In reference to framework, the playfield accumulates minoes by merging its own Grid with Tetromino's Grid. Merging happens when a tetromino is locked.

Lock Delay

The game employs lock delay, which can be *move resetted* (rotation, translation) and *step resetted* (gravity).

Ghost & Hard Drop

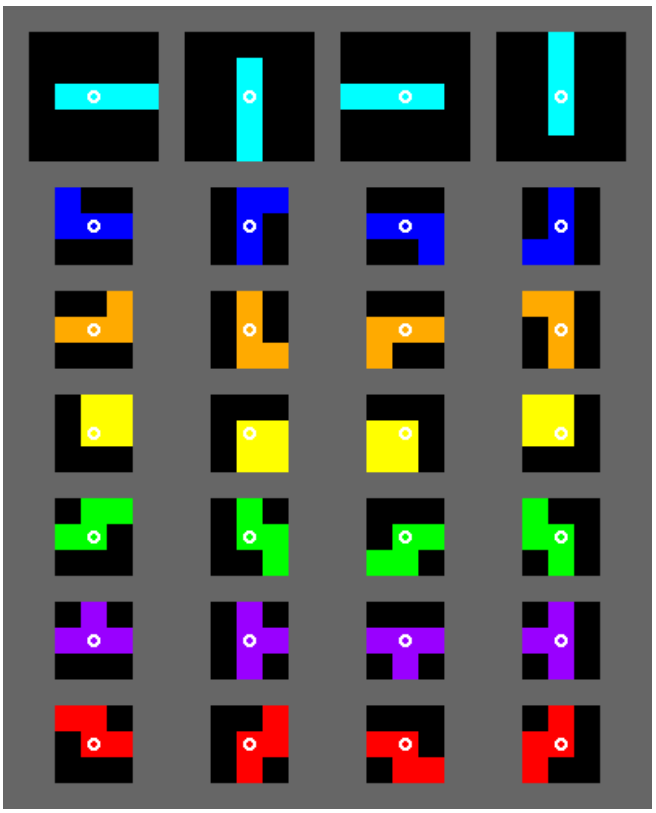
Ghost is calculated by recursively translating downward until collision occurs. Hard drop is achieved by translating to ghost position.

Hold

Achieved by swapping out active/next tetromino in the state.

Rotation & Wallkick

The game employs the SRS “true rotation” system by TTC (image below), utilising tetromino offset data based on rotation state transitions to integrate wall kicks alongside rotations. This is achieved by combining Pos translations retrieved from the offset data table and matrix rotations for each rotation request.



(Source)

Additional Features

Tetris 7-bag

Implemented with lazy sequences. Each next() call returns a immutable lazy sequence closure containing the updated object values without any side effects. When the pointer of the object reaches the end of the array, a new seed is generated by hashing the previous seed, which is then passed into the pure shuffle() util function to generate a new tetris bag.

Pause

Explained in Observables.

Wall-kick

Explained in Functionality.