FIT2102 Report

Code architecture follows the MVC pattern, which is a design pattern that separates an application into three interconnected components. Model, View and Controller.

The application uses pure functions, which always return the same output for input, as opposed to directly modifying data. This makes the system's behaviour more predictable and less likely to produce weird side effects. We use small pure functions to increase testability, modularity and code maintenance.

1. Game is initialised with a default state (model in MVC):

Decided to use a 2d array to represent the game board/tetromino element as it would make rendering/collision detection/rotation/grid clears/etc far simpler and more extensible. I could have added the “coordinate” as part of each grid element, but chose to defer this to a HOF (traverseMatrix in utils) to decrease space usage at the cost of a function which lends itself more to imperative style coding. An alternative implementation was to use a single array to store x and y coordinates and perform linear search for elements, however that is far less efficient in runtime.

Rotation follows the Nintendo System and rotates clockwise. An alternative option was to hard code the tetromino variants, which makes it “faster” and more flexible (can adjust pivot point), but uses more space and is less clean code wise.

Falling tetromino elements contain an id (current number), which makes it easy to retrieve the element by id and also maintain a consistent expected block type through hashing (utils) function.

1. Observables (Controller in MVC)

Space: Restarts game

Rotate: Rotates Tetromino

Drop: “Drops” the Tetromino

Move: Moves the Tetromino up down and left

Tick: A set time period indicating when a Tetromino descends

Observables are based on the observer pattern (publisher subscriber), where if a stream of data is changed the subscriber (reduceState) is notified of the change and will adapt accordingly.

Tick speed is managed by rxjs tap, which performs side effects on the observable stream. The tick speed also uses a special kind of observable called a behaviorsubject, which allows its value to be modified.

1. “reduceState” will update the Gamestate based on the observable (input) and current state

dropTetromino - drops the Tetromino

rotateTetromino - rotates the tetromino

moveTetromino - move the tetromino

Space - restarts game

s.gameEnd - do nothing

default / tick - Handle the block falling down

isValidPosition (utils.ts) takes in the coordinate of a tetromino, shape and the current grid state to determine whether the intended position is valid or not. I.e if it off the screen or the shape coincides with another block in the grid state. This function is used in many aspects throughout the game, such as to verify the drop position, tetromino movement, rotation and when the tetromino should be added to the grid.

I would like to highlight the “createTetrominoAction” function which is a hof that abstracts movement and rotation behaviour of the current tetromino. Resulting in cleaner, more maintainable code.

1. “Subscribe” will check the state for changes and update the “view” (in MVC) based on game state.

The entire grid state is checked and rendered/updated, also removing any redundant blocks. The HTML Blocks have an id consisting of row and column, making it simple to manipulate the canvas based on the grid state. This approach is highly extensible and can support more features without requiring additional adjustments.

An alternative is to have more of an event driven rendering. Only rerendering the components which have changed based on gamestate, which would be highly efficient in runtime but very painful to manage state wise.

“rxjs” follows the Observer design pattern (commonly known as publisher subscriber). Where changes in game state will result in an update in view.

Additional Features

Used AI to generate comments and for ideas

Bomb:

The bomb object is rendered differently from a normal tetromino block. To make the code more modular, I created a higher order function called createObj which takes in a radius argument to make it optionally round. I also extracted the creation of elements to two functions called createCube and createCircle.

Added bomb type to the tetromino mapping constant. Where it can be “randomly” selected.

The bomb object is treated in the same manner as a normal tetrimino, except when it is “added” to the grid it will trigger an alternative function, where it will detonate and clear all cubes in a 1 unit radius if there are any. As there are only two different types of elements that need to be handled (cube and circle) I used a ternary function to select between the logic. However if more types were to be added to the game I would extract this handling logic into its own function to increase modularity and code maintenance.

Drop Feature:

Initially implemented a function that finds the lowest y coordinate that a tetromino can be validly placed in, however after implementing this idea I found that the tetromino could phase through objects and thus scrapped this idea.

In an early iteration of my current approach, I implemented a while loop which iterates through the grid, top to bottom and finds the first invalid position. It will then break and return the last valid position and thus place the tetromino in that position. This approach ensures that the tetromino would not phase through any elements. As while loops are forbidden in the assignment, this was converted to use an inbuilt javascript function (reduce).

Unit Tests:

Tests individually test units/components to ensure they work as intended. With functional programming, functions have predictable outcomes as they are pure and do not have side effects. Data is also immutable, where new data structures are created rather than altering existing ones, ensuring that data remains consistent throughout the application lifecycle.

A unit test for a functional code can be written with confidence that, if the test passes, the function under test behaves correctly under the given conditions. Writing tests is particularly useful in the world of software development, as when you are adding more and more features and changing the codebase, the unit tests can catch potential bugs you are adding to the code.