Please refer to README.MD for file structure description.

High-Level Overview and Design Decisions:

The game's core logic is encapsulated within the `game$` Observable. This Observable merges keyboard input and game tick observables to let those manipulate the game's state as the user wants. Since game$ is an observable, we have a steady stream of its state and as such, we can retrieve values from it and manipulate accordingly in realtime to make the game progress.

Definitions:

Cube is a 1x1 object

Block is aa collection of made up of 4 Cubes in any shape

Rotation system used is a **Super Rotation System** (SRS) where it is possible to rotate a block 90 degrees clockwise or anti-clockwise

FRP Style & Observables:

1. **merge** and **scan**: The merge operator is used to combine the game tick and keyboard events into a single Observable, which is then processed by the scan operator based on different age events to update the game state. This decision was made to handle multiple asynchronous events in a unified, sequential manner. Important to note that `scan` invokes a pure function (in this case, gameActions[event]) with the previous state and the current event to generate a new state. This approach conforms to the state's immutability,

2. A **score$** observable was created using **BehaviorSubject** to be able to retrieve the current score of the game state. We then keep on updating it with the current game score in the **game$**'s subscribe handler. We need the current score at all times because our **tick$** observable is adjusted based on the score of the game to change game speed. This **score$** is safe to use. Due to the pure nature of state manipulation in FRP, we know we have a correct source of truth. Thus we are able to increase game speed after a specific score is reached to increase difficulty.

3. **switchMap** is used to map the **score$** observable to an observable created using the **interval** function. When a new score is emitted by **score$**, the **switchMap** operator subscribes to a new inner observable created by the **interval** function.

Functional Techniques:

1. The **move** function uses a technique called partial application/currying. Initially, it takes **moveLogic** as an argument and returns a new function that awaits a block as its input. This approach allows us to create new, specialized movement functions such as **moveBlockDown**, **moveBlockLeft**, and **moveBlockRight**. These are created by partially applying **moveLogic** and **boundaryCheck** to a higher-order function called **createMoveBlockAction**. The result is a series of functions that can move a block down, left, or right with the appropriate boundary checks. This method offers flexibility for creating new movement functions, reusability of the same logic for different blocks, and improved readability. If in the future we need a different move logic, perhaps for a diagonal or upward movement, this setup allows easy incorporation of such changes.

2. Similar techniques have been employed for rotation-related functions. Currently, we use clockwise and anti-clockwise rotations. If we wish to introduce new rotation logic in the future, it can easily be provided to the rotate function. Additionally, we've created a higher-order function (HOF), **createRotateBlockAction**, to handle situations common to any rotation such as collisions. This function simply needs to be given a rotation logic to perform its task, which can be flexibly provided later. This HOF is also responsible for taking in common parameters such as the block to rotate and previously placed blocks, and doing common actions such as collisions. That way we can keep these abstract from the dedicated rotation functions for clockwise and anticlockwise

3. **createCube** is a HOF that allow us to partially apply properties like color and the svg element to create. Later we can provide any position value to this partially created function and render accordingly. This greatly promote reusability as is used in **renderBlock** and **renderOldBlock** functions.

Purity & immutability:

Throughout the codebase, Typescript annotations have been used to make **State**, **CubePosition** and **BlockPosition** readonly entities. This assisted in ensuring we do not directly mutate them when writing code.

Side notes:

At the moment, we are generating random numbers for generating the random blocks in the game. While this may be considered impure due to having different output for same input, this is an acceptable practice because otherwise we shall have the same tetris shape every time.

However, for testing purposes, we may want to provide a way to include a seed value to get consistent outcome.

Additional feature (Holding Block):

The feature is such that it allows the player to hold a block and swap it with the current block. This will allow the player to save a block for a more opportune time. This can be used in the current loop of the game only.

So, in the context of the game we are implementing, the current block that is falling can be held. Once a block is held, it remains in the hold area until the player decides to swap it with the current block.

At the moment, the player needs to remember which block is in the hold area. This is because the hold area is not displayed on the game board for raising skill gap.

The currently falling block is vanished if the held block is summoned and placed already.

At a technical level, this feature has been implemented by introducing a **holdBlock** property of **State**. When a user presses the button associated with hold, the currently falling block is set as the holdback and a new Block is generated. Later when the key is pressed again, the **holdBlock** is set as the **currentBlock**