For definitions of elements in the game, file structure, game rules and instructions of playing, please refer to README.MD

The Tetris-like game was developed using a functional reactive programming (FRP) style with TypeScript and RxJS. The design decisions and functional programming techniques used are discussed below:

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 events, which are also Observables, and updates the game state accordingly. This design decision was made to centralize the game's logic and control flow, making it easier to understand, debug, and maintain.

FRP Style & Observables:

1. `fromEvent` and `filter`: The game uses the `fromEvent` operator to create Observables from keyboard events and the `filter` operator to process only the relevant keys. This design choice allows for easy addition or removal of controls and reduces unnecessary processing of irrelevant keys.

2. `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 to update the game state. This decision was made to handle multiple asynchronous events in a unified, sequential manner.

State Management & Purity:

The game state is managed using the `scan` operator, which acts like a reduce function for Observables. Each time an event is emitted, `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 ensures the state's immutability, a fundamental principle in functional programming.

Purity and Side Effects:

The code is designed to ensure that side effects are isolated and controlled. This is evident in the game$ Observable's tap operator, which is used to update the score and high score BehaviorSubjects. The tap operator is specifically designed for side effect operations, and it does not affect the main data flow. This approach ensures that side effects do not interfere with the game's core logic, thus maintaining the purity of functions.
In the game observable, we need to have certain side effects such as using the tap operator is used to update score/highScore. Observables for score/highScore can be subscribed to and calling their next methods will notify all subscribers of the new value. However, in this case, these side effects are necessary for the game to function correctly. They are isolated as much as possible to specific parts of the code (the tap operator and the next method of BehaviorSubject) to minimize the impact on the rest of the program.

Advanced Observable Usage:

1. `takeWhile` and `repeat`: The `takeWhile` operator is used to end the game when the `gameEnd` condition is met. After the game ends, the `repeat` operator restarts the game after a delay. This design choice allows for a simple and declarative way to control the game's lifecycle.

2. `withLatestFrom`: This operator is used to combine the game state with the high score, which is managed by a separate BehaviorSubject. This allows for the high score to be updated and displayed alongside the game state.

Functional Techniques:

1. Currying: The `hasObjectCollided` function is curried, allowing it to be partially applied with the direction first and then used later with the object and old blocks. This technique increases the function's reusability and improves code clarity.

2. Higher-order functions: The `createMoveBlockAction` function is a higher-order function that returns a function based on the given direction and boundary check function. This design choice promotes code reusability and reduces redundancy.

Code Modularity and Reusability:

The code is highly modular and reusable, thanks to the use of functional programming techniques. For instance, the move function is a higher-order function that returns a new function based on the given direction. This function is then used to create the moveBlockLeft and moveBlockRight functions. This approach reduces code duplication and improves code maintainability. It also demonstrates the power of functional programming in creating modular and reusable code.

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.