Tetris Game by John Banh

Introduction

The Tetris game implementation in focus incorporates a wide array of features and strictly adheres to best coding practices. It emphasizes functional reactive programming (FRP), state management, and the use of observables to implement logic for rotations, wall kicks, ghost blocks, collision detection, among others.

Controls

Move Left/Right: The arrow keys (Left and Right) are used to move the Tetris block horizontally. The event stream generated by these keypresses is treated as an observable sequence. This sequence is then subscribed to, effecting state changes in a pure function, moveBlockLeft or moveBlockRight, that returns a new game state without mutating the original one.

Rotate: The Up-arrow key is used to rotate a block. Like moving blocks, the rotate action also triggers an observable event, which is handled by the pure function rotateBlock.

Soft Drop: Pressing the Down arrow key moves the block downwards at a speed faster than the normal falling speed but slower than a hard drop. This action also uses observables and maintains purity via calling the tick function.

Hard Drop: The Space bar triggers a hard drop, placing the block immediately at the lowest possible position. This action also follows FRP principles, encapsulating the logic in the moveHardDown function, which is called recursively until the block reaches its final position.

Hold Block: The C key allows players to hold a block for future use. Pressing this key triggers an observable stream, handled by the holdBlock pure function.

Features Implemented

Rotations Using the Nintendo Rotation System

The rotateBlock function handles block rotation, using a Set to distinguish block types. Specifically, square blocks and line blocks have different rotation logics to align with the

Nintendo Rotation System (NRS). The function is pure, as it returns a new state object without altering the original state.

Note: Implementing the line block rotation was facilitated by initially focusing on the pivot, then specifically addressing the line block by pivoting using its third index and rotating it back. This approach seamlessly aligns with the NRS.

Wall Kicks

The wall kick logic is also a part of the rotateBlock function. This function employs a list of standard tests and kicks for "I-blocks" to determine if a rotation can occur without collision. The function maintains its purity by performing all checks and operations in a separate functional scope, without altering the existing state.

Ghost Blocks

The getGhostBlockPosition function produces a clone of the falling block to calculate its landing position. It utilizes recursion to move the block downwards until meeting a stopping condition. In alignment with FRP principles, it maintains purity by creating a new ghost block object without affecting the existing game state.

Move Hard Down

In keeping with FRP, the moveHardDown function recursively calls moveBlockDown until the block can no longer move. This approach encapsulates the logic away from the main game loop and ensures purity by not mutating any state.

Hold Block

The holdBlock function enables the holding of a block for future use, ensuring no block is held more than once during its fall. This function too upholds state purity by always returning a new, unaltered state.

Note: To preserve purity, new blocks are generated every time a block is held. This ensures that all blocks start at their appropriate positions without the need to account for previous locations.

Collision System

The canMove and canMoveDown functions are responsible for collision detection. These functions validate cell positions following any proposed moves and confirm that no overlaps

occur. All of this is accomplished without affecting the original game state, thereby maintaining purity.

Bag of Blocks and Random Hashing with Seeds

The game employs the "bag" model for random generation of Tetris blocks. Each bag contains a set sequence of blocks, and seeds are used for random number generation. This ensures both repeatability and purity.

Note: The initial use of .random() was replaced because it introduced impurity through its non-deterministic behaviour.

BehaviorSubject for Level Tracking

A BehaviorSubject is utilised to monitor and update the current game level in real-time. Whenever the game state changes—such as when lines are cleared—the BehaviorSubject emits a new value, thereby triggering appropriate updates like game speed adjustments. This practice is in line with FRP principles, offering a reactive approach to level management.

Functional Reactive Programming and Observable

The architecture employs FRP rigorously. All state changes are represented as a stream of immutable states, adhering to the principles of state purity. Observables take this a notch higher by handling not just simple input but also game ticks and other asynchronous operations. These observables unify different streams of data like user input, game ticks, and level changes into a single, cohesive game loop. This approach results in high modularity and maintainability, core tenets of FRP.

Conclusion

The Tetris game implementation not only maintains state purity but also handles complex features like wall kicks, ghost blocks, and block rotations with functional elegance. The rigorous adherence to FRP principles, coupled with the use of Observables and BehaviorSubject for real-time state and level management, achieves a robust, modular, and highly maintainable codebase.