Functional Reactive Programming Tetris Game Report

Overview

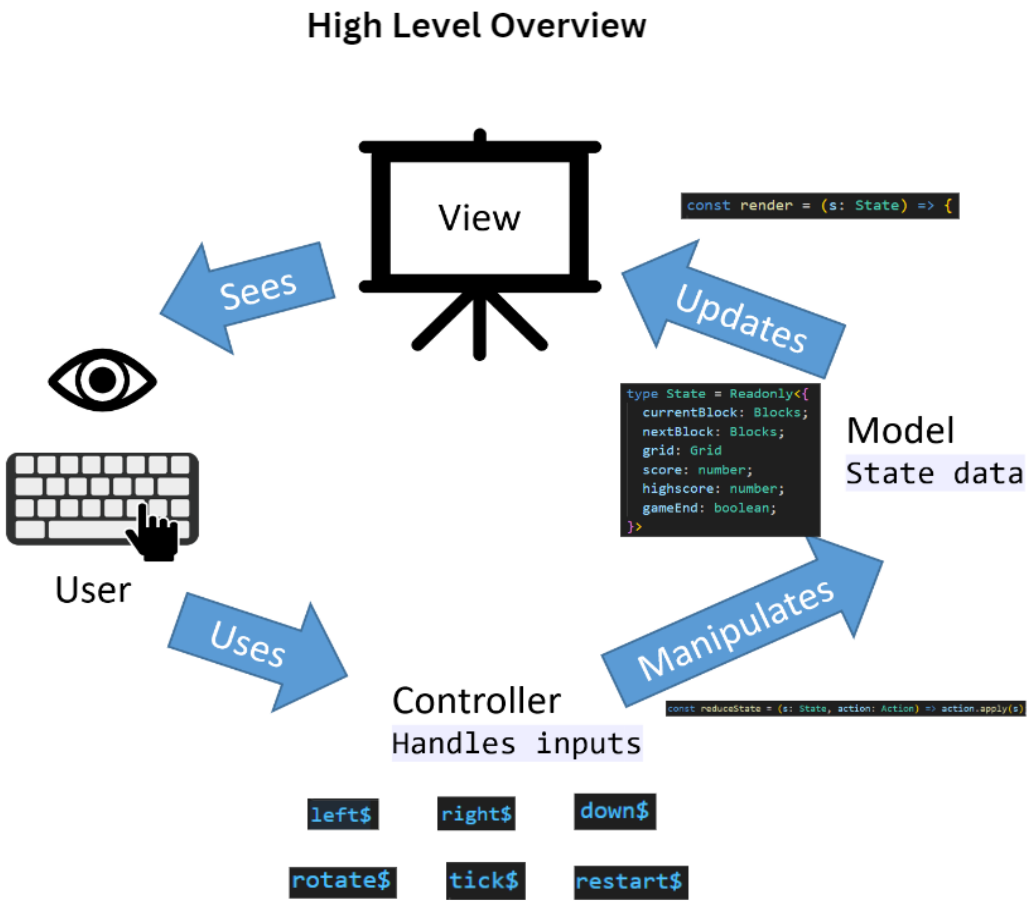
This is a TypeScript implementation of a simplified Tetris game using the principles of Functional Reactive Programming (FRP) with the RxJS library.

Design Decisions

Model-View-Controller Architecture

The following diagram illustrates how I have followed FRP. My code is organized into three main components:

- Model - manages and processes data.
- View - displays the data to the user.
- Controller - handles input.



Observables and Functional Reactive Programming (FRP) Usage

I have used observables streams to manage user actions like keyboard inputs. One interesting thing I've learned is that merge when applied to asynchronous streams will merge the elements in the order that they arrive in the stream. This real-time processing allows them to be seamlessly integrated into our game logic.

The use of Observables aligns with the principles of FRP, where the game's behavior is modeled as reactions to streams of events. Various observable operators like map, scan, filter, and merge help manipulate streams to drive game behavior efficiently.

Main Game Stream

```
const action$: Observable<Action> = merge(tick$, left$, right$, down$, rotate$, restart$)
const state$: Observable<State> = action$.pipe(scan(reduceState, initialState));
const subscription: Subscription = state$.subscribe((s: State) => { render(s) });
```

The main game stream acts as the CPU of our game. It combines multiple input streams (action$), processes state updates (state$), and updates the view (render). Side effects are contained as much as possible.

FRP Approach

FRP simplifies complex event handling and offers a clear way to model game behavior over time. This approach ensures that the game state is a pure function of the input streams, making it more predictable and testable. Loops from an imperative approach are eliminated as well because loops are the source of many bugs.

State Management and Functional Programming (FP)

- Referential Transparency
- Function Purity

Immutability

I did not use any mutable variables in my code to maintain Immutability throughout the game. By keeping our game state immutable, we prevent unintended side effects and make state updates more predictable.

Transducer

Transducers (reduceState) provide composable transformations that allow us to apply a sequence of operations to the game state without mutating it. This functional approach maintains the purity of our state updates. I have also implemented modular functions and a custom type system in my code.

Action Handling

To facilitate state updates, I have implemented an action-based system. Actions triggered by user input or game events are handled by distinct classes. Each action class implements an apply method that takes the current game state as input and returns an updated game state. By encapsulating each action in its own class, we achieve a clear separation of concerns. This makes it easier for me to reason about how each action affects the game's state.

Render Handling

State changes automatically trigger rendering updates, guaranteeing that the game's visual representation stays synchronized with the underlying logic.

Bug Identification and Resolution

I encountered a rendering bug while implementing my code. The issue arose from storing individual cube positions within the 'positions' property of the 'Blocks' object. To render the game, I looped through the grid, but this approach led to problems. Specifically, I had to loop through 'block.positions' for each square to access the respective positions when creating SVG elements. This resulted in creating multiple instances of the same cubes.

The problem became more noticeable when I removed a full row. The cubes belonging to the same block above it would replace its position, making it seem as though the removed row was still present. I attempted to assign unique IDs to each cube for identification but encountered challenges in implementation.

Conclusion

In crafting this Tetris game, I was able to create a fun game and gain a better understanding of FP, Observables and FRP. The advantages of functional programming over imperative are also shown.