Huzaima Amir
32717016

FIT2102 ASSIGNMENT 1

Reflection Report: Process, Challenges, and Insights

Introduction

For this assignment, I implemented a simplified version of Flappy Bird using TypeScript, RxJS, and SVG for rendering. The main goal was to apply functional reactive programming (FRP) principles to structure the game around immutable state and event streams, rather than relying on imperative updates. My implementation achieved all the minimum requirements: bird movement with gravity and flapping, pipes generated from CSV data, collision detection with the ground, ceiling, and pipes, a life system with three attempts, score tracking, and a victory condition when all pipes are cleared. I also managed to complete the restart functionality from the full requirements, and my game runs smoothly, I believe.

Core Logic

The core of the game is managed through RxJS observables. I used three event streams: tick$ for timed updates, flap$ for player input via the spacebar, and restart$ for restarting after a game ends. These streams were merged and processed with the scan operator, which folds over events to evolve the state. This structure ensured that the state was always derived from the previous state and events, keeping the game loop purely functional.
The tick function is central to the design. Each tick calculates the bird's vertical velocity by applying gravity, caps it at a maximum, and updates its position. The function also moves pipes leftwards across the screen, spawns new ones based on the CSV timing data, and increments the score when the bird successfully passes a pipe. Collision detection was implemented through helper functions: one for boundaries and one for pipes. If a collision occurs, a life is deducted, and the bird is bounced with a randomized velocity. The game ends either when all three lives are lost or when all pipes are cleared, in which case a victory is declared.

Adherence to FRP

An important part of the assignment was adhering to FRP principles. I avoided the use of mutable variables such as let, ensuring that all state transitions returned new objects without side effects. Rendering was separated into a pure state to view pipeline: the state is updated independently, and then the render function reflects it by updating the SVG canvas. This clear distinction between model and view simplified reasoning about the program and made the logic modular.

Challenges

I also attempted the ghost bird feature, which would replay previous runs as “ghosts” to follow alongside the player. I explored using a ReplaySubject to capture flap events and generate ghost streams. However, my approach led to overlapping ghost streams that quickly became unmanageable, breaking the purity of the implementation and cluttering the game. While I could not complete this feature, I learned valuable lessons about the challenges of stream composition and managing multiple concurrent observable sequences. The main challenges I encountered included tuning the physics constants to make the bird movement feel natural, ensuring that lives only decremented once per collision rather than all at once, and structuring the code so that state management remained pure. Each of these required careful debugging and reinforced my understanding of immutability and FRP design.

Conclusion

In conclusion, my implementation fulfilled the required functionality and demonstrated an FRP approach to game design. Although I was not able to successfully add the ghost bird feature, the process deepened my understanding of RxJS and highlighted the importance of pure state transitions and clear separation of concerns. Overall, I believe in this assignment I successfully met the learning objectives.