FIT2102 Assignment 1 Report

❖ Introduction & Summary of Workings

This report describes the design of a Flappy Bird game built with Functional Reactive Programming (FRP) in TypeScript and RxJS. The core is a reactive game loop that processes user inputs and ticks as observable streams, composed into a single stream of game states rendered to the screen. Pure, testable game logic is separated from impure operations like rendering and input, ensuring predictability and decoupling from the view layer. Key features include declarative event handling, immutable state management, and FRP-driven time-based mechanics such as the ghost replay system.

❖ High-Level Design Decisions & Justification

The primary design goal was to adhere strictly to **FRP principles** to manage the complexity of asynchronous events and state changes inherent in a game.

The application is built on a **unidirectional data flow model**, chosen for its predictability. This design prevents complex state updates and race conditions by ensuring data flows through a clear, linear pipeline:

`Inputs (Events) → Action Streams → State Reducer (scan) → State Stream → View (Render)`

Instead of a single, large function to handle all types of state changes, the logic for each change is **encapsulated into its own class** (e.g., *Tick*, *Flap*).

- Each action object contains an `apply` method that knows how to transform the game state.
- This approach organizes the code functionally, treating state transformations as self-contained units that can be passed as data through RxJS streams.
- It improves modularity and makes the system easier to extend.

All **core game logic** physics calculations, collision detection, and state transitions are implemented as pure functions in `state.ts`.

- These functions operate only on their inputs to produce new data, without any side effects.
- This *pure functional core* is the cornerstone of the architecture.
- Purity leads to **referential transparency**, making the logic easy to reason about and test in isolation.

By composing these small, verifiable functions, we can build complex game logic that remains robust and predictable.

❖ State Management and FRP Style

The entire game state is contained within a single, **immutable `State` object**, managed reactively.

The heart of this system is the **`scan` operator**, which implements the reducer pattern for streams:

- It subscribes to a merged stream of all possible actions.
- For each action, it calls a pure reducer function that takes the current state and the action, then computes a completely new state object.

This is where the principle of **immutability** is enforced; the reducer never modifies the state it receives. As a result:

- The state is always a predictable result of the history of actions.
- Bugs from mutable state are prevented.

To ensure consistency, the game state stream is **shared across all parts of the application using `shareReplay(1)`**.

- Every component receives the exact same state updates.

- The state is computed only once, regardless of how many parts of the application need it.

❖ Advanced Observable Usage

RxJS observables were used to implement **complex, time-based game features** in a declarative style.

- **Pipe spawning** is handled as a pure function of the game's clock.
    - On every tick, a pure function filters the pipe schedule based on the current `gameTime`.
    - This deterministically identifies which pipes should appear, without using any impure timers.
- **Ghost bird feature**:
    - Replays previous runs in real time by using `combineLatest` to compose three distinct streams:
        1. The history of all completed runs
        2. A stream of the current animation frame number
        3. The main game state
    - This declarative approach synchronizes the ghost replay perfectly with the current game's clock.
    - No imperative logic is required, showing how complex, time-based relationships can be elegantly modeled with streams.

❖ Design Trade-offs and Alternatives

A **traditional OOP approach** might involve a `Bird` class with methods that mutate its own state.

- This leads to state being distributed and hard to track.

The centralized, **immutable state of the FRP model** was chosen instead for its predictability and testability.

For the **ghost feature**, another alternative would have been:

- Store past runs in an array
- Use an imperative game loop to manually sync the replay

However, the **chosen stream-based approach** is more declarative and automatically handles synchronization, avoiding manual logic that is often prone to errors.

❖ Advanced Features

1. Pause/Resume with Countdown

The pause mechanism is integrated directly into the **reactive state machine**, avoiding side effects such as `setTimeout` that would break purity.

- When the player presses **'P'**, a `Pause` action toggles the `isPaused` flag and records the trigger time.
- At the same moment, the state is updated with a **countdown value** representing the three-second delay before resuming play.

On every subsequent **Tick**:

- The reducer (`reduceState`) calculates the remaining countdown by comparing the current clock to the recorded pause time.

- While the countdown is active:
    - The reducer prevents the main `gameTime` from advancing.
    - Pipe updates, collisions, and scoring are suppressed.
- Once the countdown reaches zero:
    - The reducer automatically clears the flag.
    - Normal gameplay resumes.

This **declarative approach** guarantees deterministic behaviour:

- If you replay the exact same sequence of inputs, the countdown will behave identically.
- It also simplifies testing, since applying a controlled sequence of Tick actions to a paused state reproduces the exact timing logic without reliance on real-world timers.

2. Power-Up System

Power-ups such as **shrink** and **slow down** are modelled as **scheduled events** rather than random spawns.

- On every **Tick**, the reducer checks the deterministic spawn schedule.
- It immutably appends new power-ups to `state.powerUps`.
- Each power-up is represented with metadata, including its type, spawn time, and duration.

When the bird collides with a power-up:

- The reducer sets an **active flag**.

- It records an **endTime** based on the current `gameTime`.
- From that point onward, effects are not applied imperatively but are derived directly from the state on each tick.

Examples:

- `updateBirdPosition` checks `shrinkActive` to dynamically adjust the bird's radius during collision checks.
- Pipe movement speed is computed with a `slowDownMultiplier` that smoothly interpolates back to normal speed as the `endTime` approaches.

This design makes all effects a **pure function of the State**.

- There are no mutable timers or hidden callbacks in the system.
- As a result, power-ups are reproducible: running the same sequence of actions always yields the same effect timings.
- This demonstrates the FRP principle that the **present state is fully determined by history**.

3. Comprehensive Unit Testing

Because the functional core is completely pure, the test strategy is straightforward yet powerful. Using Vitest, unit tests cover:

- Physics: gravity and flapping in updateBirdPosition and flapBird.
- Collisions: overlap detection in checkBirdPipeCollision across different bird and pipe positions.
- Scoring and Game Flow: reducer transitions on Tick, Flap, Restart, and Pause.
- Ghost Replay: verifying that calculateGhostPositions correctly reconstructs trajectories for past runs.

- Power-Up Decay: ensuring that multipliers and radius adjustments correctly deactivate at the recorded endTime.

Each test constructs a sample immutable State, applies an Action, and verifies the new state. No DOM mocking is required, as rendering is isolated to view.ts. This provides broad coverage of mechanics, ensures deterministic behaviour across runs, and gives confidence that both normal and advanced features behave consistently under the FRP model.

❖ Conclusion

Building this game using an FRP paradigm resulted in a declarative, modular, and robust codebase. By treating all events as streams and managing state immutably, the system is easy to reason about, test, and extend, demonstrating the practical benefits of FRP for managing complex, interactive applications.