Flappy Birb - Design and FP/FRP Implementation

Introduction

This project is a Flappy Birb implementation in TypeScript based on Functional Reactive Programming (FRP) and `RxJS` Observables. The game has a falling bird, which can flap when the space key is hit, and must go through a series of pipes as outlined in an external CSV file. The player begins with three lives and when the player hits something they lose a life, and the game finishes when the player loses all the lives or all the pipes are cleared. The score goes up when the bird manages to go through pipes.

The code is in a pure functional style, and side effects are limited to rendering and event handling. State transitions are specified as reducers and used with `RxJS` operators. One major design choice was to represent the whole game as a sequence of state changes caused by user input (`flap$`), time ticks, and game events (collisions, scoring, or ending). This choice makes the state management predictable, immutable, and composable, as per the FRP principles.

State Management in FP Style

The state is stored in an unchanging `State` object that holds the bird, pipes, pipes awaiting, score, lives, time elapsed and the game-over flag. Pure functions such as `tick` and `checkCollisions` are used to handle each update. New state objects are updated every frame, rather than being mutated.

The `tick$` function uses gravity, moves pipes, creates new pipes based on the CSV schedule, and collisions are checked. Mapping and filtering are used to update arrays, not to mutate them. This fixity renders the logic simple to reason.

The reducers are actions that mutate the state, and are used with the `scan` operator of RxJS. This reducer pattern implies that state changes are not imperatively written, but instead they are declared. Isolation of update logic and rendering provides modularity and purity.

FRP and Observable Usage

The game loop and input are modeled entirely as Observables.

1. **User input**: The spacebar press (`flap$`) is an Observable which sets the velocity of the bird. It does not need any direct DOM listeners, inputs get sent into the same reducer stream as ticks.
2. **Time**: An interval Observable (`tick$`) drives gravity and movement. Time is declarative but not an imperative loop.
3. **Composition**: Tick and flap streams combine into `actions$`, which suffice to reduces the state. Higher-order Observables and `switchMap` are used to restart the game again.
4. **Purity**: The state transitions are pure and the side effects are in render, which renders the SVG view.

Those show how Observables unify time, input, and state under one model.

Ghost Bird Feature

The ghost bird adds to the base game by recording the movement path of the player in each game, and replaying it in subsequent games. The `scan` operator adds to a `BirdPath` sequence each frame the current position of the bird and the time. This path is stored into a session history array when a game has been ended. Ghosts in future runs are drawn in semi-transparent form, in the same direction as previously (with the same elapsed time).

FRP ideas are well illustrated in this feature. Instead of mutating state, the result of every run is collected as a sequence of positions. The restart logic does not delete any past run, only the current run is cleared, thus ghosts are visible across sessions. The `render` is processed independently in the render function where all ghosts are drawn first in the background to be seen behind the main bird. The ghosts are non-interactive, passive, and leave collisions alone, which makes them consistent with the declarative style of FRP. Using ghost birds, instead of using imperative state storage, the design remains functionality-wise simple and pure, and provides some complexity to the gameplay.

Leaderboard Feature

The leaderboard logs and shows the highest scores in the same session of running a number of runs. The player has his final score added to a leaderboard array at the end of each game. This list is then filtered and narrowed down so that only the most important scores are displayed. The leaderboard is kept up

to date in a reactive way, with the result of every completed run streaming into the Observable pipeline.

Drawing the leaderboard can be considered a pure response to changes in states. Whenever the leaderboard is updated, the display is regenerated resulting in the ranked scores being displayed in order. This style has the same principle of purity as game rendering, except that side effects are confined to the real `DOM` update.