FIT 2102 Assignment 1 report

Overview

The project is about making a “Flappy birb” game by using Functional Reactive Programming (FRP) with RxJS. The game loop is modeled as a set of pure reducer streams that are merged and folded into state.

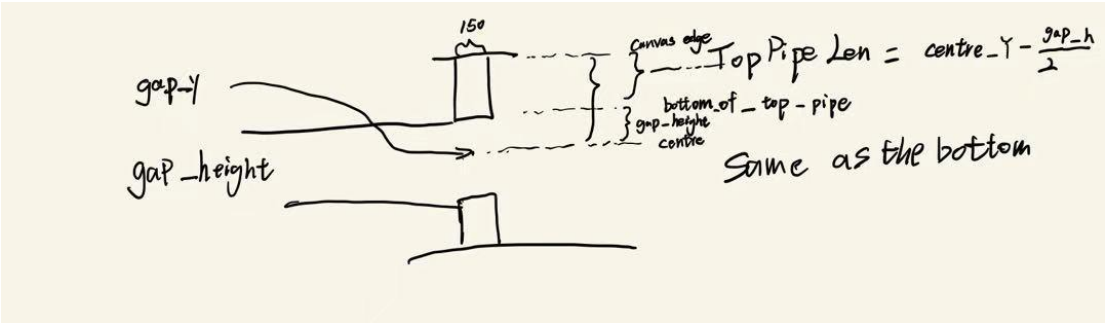
Minimum requirement:

Flapping bird and physics:

The main idea comes from the week 4 applied exercise, a tick stream integrates velocity, and a physical stream integrates position under gravity, and a Space key stream sets an upward velocity. I did this as two reducers, so their composition order does not matter; the scan fold gives the resolved position each frame. Fixed timestep (tick$ at TICK_RATE_MS) is a stable, predictable motion. Besides, flap is just (s) => s', no DOM access or mutation.

Spawning pipes:

For generating pipes, the first thing we need to make sure of is how to generate pipes based on the data provided by the CSV file. Here is the graph showing how to get the pipe data based on the CSV file data.



I parse the CSV once into a schedule and convert it into an Observable that emits at the specified times. For each row, a timer(time*1000) emits a reducer that appends a pipe with spawnedAt = Date.now(). Spawning is time-based, not frame-count-based, so it’s stable across tick rates and timer will help me calculate accurate time so pipes can generate on time.

Collisions, lives, score:

Collision checking is based on whether the bird overlaps with two pipes and top and bottom bounds. After collision, the bird will get a velocity which is opposite the direction of

the collision (if the bird hits the bottom, then the direction will be up. If the bird hits the top, then the direction will be down). A branchless “hit transition” reduces lives only on the 0 to 1 collision edge, avoiding repeated decrements while stuck. In addition, the magnitude of velocity will be generate by rng function to get a random number (the range I set is [10,30]). The score is obtained by counting the pipe centre left of the bird's x. The whole step is a pure (State) => State. No DOM or mutable globals. Collision (contact), lives (game rule), and bounce (response) are computed together per tick but remain simple expressions with no if chains.

Additional Feature: adding ghost birds via observables.

During each run, the physics reducer appends {x,y} frames(but actually, only y is necessary for the later position ghost checking) to state.tape when the current round is over(the bird lost all the lives or won the game), Pushing {frames, deathScore} into an archive subject. On the next run, all previous tapes will be replayed.

Problem:

The ghost bird will be hidden when current passedCount exceeds the ghost's deathScore, which will cause that if the ghost died midway to the next pipe centre, it keeps visible until you pass that next centre.

How state is managed throughout the game while maintaining purity and why

State is a single immutable object (State) that represents the whole game. Each reducer has the shape (s: State) => State (in my code, I defined const tick = (s: State) => s;
export type Reducer = typeof tick; to reduce occurance of the code)
and returns a new object using spreads/array copies—no in-place mutation. All the game state is stored in State type inside the state.ts file. The game reads state$ and sets SVG attributes. No gameplay logic is computed in the view.

the usage of Observable beyond simple input

I did not just use Observables for DOM events, but also for sturecture time, scheduleing, randomness and cross-run memory. Each use is a stream of pure reducers which keeps logic declarative.