FIT2102 Programming Paradigms – S2 2025

Report for Assignment 1: Flappy Birb

The game is a single immutable State stream produced by folding pure reducers over time and inputs. main.ts bootstraps: fetch CSV (pipe schedule), wait for first click, then subscribe to state$ and render. observable.ts composes tick, keyboard, and a small dynamic reducer bus into one reducer stream folded by scan → State. All DOM/SVG side effects live in view.ts; all game logic is pure in state.ts. The result is one source of truth, deterministic updates, and a strict purity boundary.

# Design Decisions & Justification

***ReplaySubject for ghosts***   
Each completed run is a ReplaySubject<number> that records the bird’s Y each unpaused tick. We complete the subject on game end and keep it in a runs accumulator. On restart, we zip each stored subject with the current unpaused tick stream, emitting frame-accurate ghost positions in lockstep with gameplay. This design (i) keeps global state minimal (only the current ghost frame lives in State), (ii) makes replay deterministic and pause-aware, and (iii) treats history as a *time series* that composes naturally with operators, instead of doing imperative array indexing.

We use ReplaySubject because it lets us treat each ghost run as a true time stream rather than a static array. This means the bird’s Y-positions are recorded frame by frame only when the pause-aware tick emits, so pauses never create “fake frames.” When a run ends, the subject is completed and can later be zipped with new ticks to replay deterministically in sync with live gameplay. This design keeps the global state lean (only storing current ghost positions), avoids manual array indexing, and leverages FRP operators for clean, composable, and pause-safe ghost replays.

***Pause as data + paused-aware time.***  
Pause is a pure boolean in State. Two guardrails enforce it: tick(s) early-returns when s.paused, and a derived playTick$ filters ticks while paused. Because recording and replay are *driven by playTick$*, there’s no drift: physics, pipes, scoring, and ghosts all freeze consistently. This separation (state flag vs. time gating) keeps the model simple, avoids hidden timers, and preserves determinism.

***Deterministic restart choreography.***  
Restart on Space (when ended) emits a short sequence of reducers: reset to initialState reusing the parsed pipes (fair, reproducible), apply an immediate flap (snappy restart), clear ghost actors, then for each stored run: activate → frame updates via zip(replaySubject, playTick$) → deactivate. We inject that sequence through a dynamic reducer bus (extraReducers$) instead of nested subscriptions, preserving the single scan fold.

# State Management & Purity

state.ts holds only pure reducers: moveBird, movePipes, checkPipes, scoring, collisions (with a seeded RNG helper for deterministic “random” bounce), and tick (which composes those reducers and increments the tick counter). Because effects are confined to view.ts, we get referential transparency, easier reasoning/testing, and reproducible behavior (vital for ghosts and restart determinism).

* Observable Beyond Simple Input (why it’s valuable)
* Time as a first-class stream (ticks, pause-aware ticks).
* Data-driven scheduling (CSV → initial pipe positions without timers).
* Streamed history (ReplaySubject) re-played as time, not arrays read in loops.
* Reducer for one-off reactive “scripts” (restart + ghost choreography) that still flow through the single state fold.