Flappy Birb — FRP Design Document

FRP architecture (Model–View–Intent)

• Intent (inputs as streams): keyboard events via fromEvent create streams for flap (Space), restart (R), and pause (P). A fixed-rate tick stream is produced via interval.

• Model (pure state machine): all game logic is a single pure reducer step(s,e) folded with scan into a stream of State. The state is immutable (Readonly<…>) and updated only by returning new values (no mutation).

• View (single impure boundary): one renderer function consumes State and mutates the DOM (SVG) accordingly. All other code is pure.

FP fundamentals applied

• Purity: step(s,e) returns a new State (or deliberately returns the same object while paused), with no hidden mutation or side effects. Rendering is the sole impure boundary.

• Immutability: State, Pipe, Bird, and GhostSample are Readonly. Collections are transformed with map/filter and spread copies.

• Referential Transparency: Deterministic transformations—same inputs → same outputs. Where variation is desirable (bounces, flaps), we use a seeded LCG RNG carried inside the model.

• Higher-Order Functions: createRngStreamFromSource, binarySearchLastLE, and the reducer pattern express reusable, composable behaviour.

• Algebraic Data Types (ADTs): Event is a tagged union (EvTick | EvFlap | EvTogglePause); step exhaustively handles each case.

FRP design pattern

• Single time base: interval(T) provides the discrete simulation clock.

• Everything is data: inputs are events; CSV pipe specs are data; ghost recordings are arrays of samples; the RNG seed is data inside the model.

• Single fold: merge(tick$, flap$, pause$) → scan(step, initial) → State.

• Effects last: only the subscription performs DOM work via render.

Streams & operators

• Time & pausing: tick$ = paused$.pipe(switchMap(p => p ? EMPTY : interval(TICK))). switchMap cleanly gates time; pausing cancels ticks rather than buffering, avoiding drift.

• Input shaping: flaps are debounced from OS key repeat and mapped to a velocity impulse. A tiny LCG is used to generate deterministic pseudo-random bounce magnitudes (pure + testable).

• Ghost session loop: runs are composed with switchScan. Each run emits States until gameEnd; the final array of GhostSamples is appended to the prior runs and fed into the next run. This makes "ghosts" a first-class stream concern rather than external mutable storage.

Operator choices:

– switchMap (pause) over mergeMap avoids parallel time sources.

– switchScan (session) instead of manual Subjects; restarts remain referentially transparent.

– scan (not reduce) so every intermediate State is available for rendering.

CSV scheduling & pipes

I parse the CSV once and schedule spawns against simulated time (a scan of dt) instead of wall-clock timers. This keeps the simulation deterministic, makes pausing trivial (ticks are gated off), and ensures replay behaves identically on every machine. Spawns are emitted by comparing timeMs ≤ t inside a fold and flattened with mergeMap, which handles "zero or many" spawns per tick without extra Subjects. Pipes then advance by dt (frame-rate independent) and are culled when fully off-screen to avoid DOM growth. I rejected per-pipe setTimeout/intervals because they fight pause/resume, drift under load, and complicate teardown.

Collisions, scoring, and lives

Score increments exactly once per pipe when the bird's x passes the pipe's trailing edge. On the first collision outside invulnerability frames, a life is lost and a direction-aware bounce is applied: hitting the top half (or top wall) sets a positive vy (downwards); bottom half/wall sets a negative vy (upwards). Random magnitude comes from the LCG so runs are reproducible and unit testable.

Ghost birds (all runs)

All completed runs' traces are retained (ghostPrev: GhostSample[][]). During a new run, every ghost image steps by binary search into its trace at current t and is hidden as soon as its trace ends—so ghosts disappear when their run "dies". Ghosts are semi-transparent and non-interactive. The composition across runs is formed by switchScan at the stream level. This approach satisfies the rubric's emphasis on an **observable-based** replay rather than relying solely on an array of saved game states.

Pause (final approach and rationale)

Implemented approach (Asteroids style):

• P emits EvTogglePause; the reducer flips paused.

• While paused === true, step returns the same State object (no allocations), freezing physics, time t and spawn progression, i-frames, and ghost recording/playback.

• The tick stream still emits on cadence, preserving a stable, deterministic simulation clock and simple control flow.

Why better than gating time with switchMap(EMPTY)?

• Avoids timer cancellation/re-creation semantics and potential drift.

• Keeps all pause semantics in one place (the model), which matches the unit's "pure state machine" emphasis.

• Easier to test: "paused" becomes a pure input to step.

Alternative design choices + justification

1) Pause by stream gating (previous version): Gate ticks with paused$.pipe(switchMap(p => p ? EMPTY : interval(T))). This conserves CPU and avoids emitting unchanged states, but moves pause semantics out of the model and couples control flow to scheduling. We intentionally moved to model-level pause to align with Asteroids-style FRP and keep determinism/self-containment in step.

2) External spawn scheduling: Schedule EvSpawn events in a separate stream (gameTime$ + scan). Works, but splits logic across streams. We consolidated scheduling inside step so the single fold fully describes the state machine.

3) Ambient randomness (Math.random) vs model RNG: Ambient randomness is simpler but breaks referential transparency. We carried an LCG seed in State, making flaps/bounces reproducible and testable.

AI Use Declaration — Design Document

I used ChatGPT (GPT-5 Thinking) to help plan and iterate the Design Report: organised headings to match the rubric, tightened wording to ~600–800 words, expanded FP/FRP rationale (pure reducer via scan, session switchScan, CSV scheduling, deterministic RNG), updated the pause section to the reducer-centric (Asteroids-style) approach, and drafted the "Alternative Design Decisions" discussion. (~12 iterations.)

I modified the outputs by rewriting in my own words, cross-checking every claim against my code, adding/omitting details to reflect my implementation, fixing terminology to match the unit, and editing for clarity and brevity. I understand the document and can explain all design choices.