

Parallelizing Geometry Dash-Style Physics

Testing whether multithreaded collision calculations improve a single game simulation in Haskell

TEAM MEMBER(S)	UNI	LANGUAGE
Jaxson Robinson	jnr2154	Haskell

1. Motivation

Players sometimes suggest that *Geometry Dash* would run faster if its physics were multithreaded. A common counterargument is that player movement and collisions depend on the results of earlier calculations, so parallel execution could introduce ordering problems or simply add overhead. I want to test that tradeoff experimentally rather than assume that more threads will help.

Research question: How much, if at all, can parallel collision detection speed up the physics-update loop of a Geometry Dash-inspired game, while producing the same results as a sequential implementation? I will measure both collision-only speedup and **end-to-end speedup per simulation tick**.

2. Algorithm to parallelize

I will implement a deterministic, simplified two-dimensional physics simulator, **not** a reimplement or reverse-engineering of Geometry Dash itself. It will feature one automatically moving player, gravity, jumping, solid rectangular platforms, and hazardous obstacles such as spikes (using simple geometric hitboxes). Each fixed-time simulation tick will:

- **Update player motion:** apply the recorded jump input and gravity to compute the proposed next position and velocity.
- **Find and test obstacles:** identify nearby collision candidates, then run independent geometric overlap tests against the proposed player position.
- **Resolve outcomes:** combine collision results in a fixed order and update the player state (landing, continuing, or dying/restarting).

The dependence between successive ticks, and the final collision response, will remain sequential. The independently evaluated obstacle tests within one tick are the main target for parallelism.

3. Parallel Haskell approach

The baseline will evaluate each obstacle test sequentially. The parallel version will use `Control.Parallel.Strategies`, initially `parListChunk` with `rdeepseq`, to evaluate chunks of collision checks on multiple cores. Both versions will share the same motion equations, collision logic, and deterministic result-combination code; only the evaluation strategy will change.

Each check will be a pure function of an immutable player state and obstacle. This avoids shared mutable writes and race conditions in collision detection. If several obstacles overlap, the results will be combined using stable obstacle IDs and an explicit priority rule, rather than whichever worker finishes first. I will tune chunk size to study the tradeoff between parallel work and scheduling overhead.

4. Input data and workload design

No external game files are required. I will generate reproducible, seed-controlled levels containing ground segments, platforms, and obstacles, as well as predetermined jump-input sequences. A few small hand-designed levels will test familiar cube-style movement; generated levels will vary object density and how many obstacles are close enough to be collision candidates. If time permits, I may also test other gamemodes (ball, ship, etc.). The simulation will run at 240 ticks per second to match the original game's TPS.

I will test at least three workloads: **sparse** (few nearby colliders), **moderate** (more densely populated sections), and **stress-test** (unusually many simultaneous candidates). Including the sparse case is important: in realistic single-player situations there may be too little independent work to justify parallelization. The stress case is meant to reveal the upper potential of the algorithm, not to represent typical gameplay.

5. Benchmarking and correctness

I will compile optimized Haskell executables with `-O2 -threaded` and run the parallel version with varying GHC runtime capabilities, such as `+RTS -N1/-N2/-N4/-N8` where supported. Each implementation will process the same seeded levels and input traces for a fixed number of ticks. I will force results to avoid accidentally timing unevaluated lazy computations, repeat trials, and record:

- **Time:** total time for the complete simulation; time attributable to collision checking; and average time per tick.
- **Scaling:** sequential time divided by parallel time, across thread counts, obstacle densities, and chunk sizes.
- **Correctness:** equality of player states, deaths, and collision decisions on the same input trace; failures count as bugs, not speedups.

I will also compare workloads with and without a simple spatial candidate filter. This will show whether a useful sequential optimization leaves so little collision work that parallelism becomes unnecessary.

Crucially, parallelizing many independent level replays will **not** be reported as speeding up one game instance.

6. Scope, challenges, and expected outcome

The project is intentionally limited to **one player, fixed-time motion, static geometry, and basic collisions**. I will exclude rendering, audio, level-editor effects, moving platforms, portals, and an exact reproduction of RobTop's engine. The main challenges are obtaining enough work per tick for threads to help, controlling Haskell evaluation overhead, and preserving deterministic collision behavior. If parallelization is slower on ordinary levels, that is still an informative result.

The deliverable will be a working Haskell program with sequential and parallel modes, reproducible level generation, correctness tests, performance plots/tables, and a short discussion of when parallelism helps or hurts. *The conclusions will apply to this Geometry Dash-inspired physics model; they will not claim to directly measure the actual game engine's speedup.*

7. Implementation plan

Stage 1: Build and test sequential movement/collision logic.

Stage 2: add Strategy-based collision evaluation and check deterministic equivalence.

Stage 3: benchmark different densities, thread counts, and chunk sizes, then analyze the limits on speedup.