Game of Life with Wormholes

Implementation & Approach

Nishan Poojary
nishanpoojary16@gmail.com

May 31, 2025

Abstract

This document describes the design, implementation, and development process behind the *Game of Life with Wormholes* project. It highlights architectural decisions, module responsibilities, algorithmic approaches, testing strategies, and the overall workflow from initial idea to final submission. The goal is to provide a concise yet thorough overview that is both technically informative and accessible to hiring managers, HR reviewers, and other stakeholders.

Contents

1 Introduction

Conway's Game of Life is a classical zero-player simulation in which cells on a 2D grid evolve according to simple local rules. In this variant—*Game of Life with Wormholes*—cells may also interact through "wormhole" connections. Two specially-colored pixels in a separate image form a portal that effectively makes distant grid positions behave as neighbors. The challenge was to integrate these nonlocal interactions without sacrificing performance or code clarity.

2 High-Level Architecture

The codebase is organized into clearly delineated modules, each responsible for a distinct aspect of the simulation pipeline:

1. **Image I/O:** Loading raw PNGs (`starting_position.png`, `horizontal_tunnel.png`, `vertical_tunnel.png`) and converting them into efficient in-memory data structures.
2. **Wormhole Parsing:** Scanning each tunnel image to identify matching-color pixel pairs and building a bidirectional map of portal endpoints.
3. **Neighbor Lookup:** Precomputing, for every cell and every direction (up/right/down/left), the correct neighbor coordinate—accounting for wormhole teleports whenever a move lands on a colored pixel.
4. **Simulation Engine:** Applying the extended Game of Life rules over multiple generations, using the neighbor-lookup tables to count both local and wormhole-assisted neighbors.
5. **CLI Orchestration:** A command-line interface that ties everything together: resolving an "example" folder, invoking I/O and parsing routines, running the simulation up to specified milestones (e.g. 1, 10, 100, 1000), and writing output PNG snapshots.
6. **Testing Suite:** Automated unit and integration tests to verify each component's correctness in isolation and in combination.

3 Detailed Module Descriptions

3.1 Image I/O

- **Responsibility:** Read PNG images from disk, convert them to NumPy arrays or boolean grids, and write output snapshots back to PNG format.
- **Approach:**
  - For the starting position image, we interpret any pixel that is exactly white (RGB = (255,255,255)) as *alive* and any other pixel as *dead*. This yields a boolean matrix of size $H \times W$.
  - For the tunnel images, we load the full-color PNG and scan each pixel's RGB triple. Black pixels are ignored; any non-black color is considered part of a wormhole. We group coordinates by color to prepare for pairing.
  - Output images are generated in black-and-white format: alive cells are rendered white, dead cells are rendered black.
- **Key Design Choice:** We store the starting grid in a contiguous boolean NumPy array to allow fast indexing during neighbor counting. Output PNG writing is batched at milestone intervals to avoid frequent disk I/O.

3.2 Wormhole Parsing

- **Responsibility:** Identify and validate pairs of portal endpoints in each tunnel image, then produce two mappings: one for horizontal tunnels, one for vertical tunnels.
- **Approach:**
  1. Read each tunnel image pixel by pixel, grouping all non-black pixels by their exact RGB color.
  2. For each color group, assert that exactly two coordinates share that color. If the count is odd or not exactly two, raise an error. This enforces the "two endpoints per wormhole" rule.
  3. Sort the coordinate pair either by row (for vertical tunnels) or by column (for horizontal tunnels) so that pairing order is deterministic. Then record a bidirectional map from one endpoint to the other.
- **Key Design Choice:** Using a dictionary keyed by RGB triples ensures we can quickly collect all pixels belonging to the same tunnel. Sorting before pairing avoids arbitrary pairings and ensures reproducibility.

3.3 Neighbor Lookup Tables

- **Responsibility:** Precompute, for every grid cell (r, c) and each "orthogonal" direction (up/right/down/left), the correct adjacent cell coordinate—factoring in wormhole jumps whenever a move lands on a portal pixel.
- **Approach:**
  1. Create two integer arrays, `nbr_r`$[r, c, d]$ and `nbr_c`$[r, c, d]$, of shape $(H, W, 4)$. Each (r, c, d) entry either holds the row/column of the neighbor in direction d or a sentinel indicating "no neighbor" (edge of grid).
  2. For each cell and each direction d:
     - Compute the candidate next cell coordinates (r', c') by stepping one unit in that direction (e.g., $(r-1, c)$ for *up*).
     - If (r, c) itself sits on a portal endpoint for that same axis, we first teleport to its pair, then attempt the step.
     - After computing (r', c'), if that pixel also coincides with a portal of the *other* orientation, teleport again. In other words, stepping onto a colored pixel automatically sends you through its wormhole partner before finalizing the neighbor.
     - If (r', c') lies outside the grid bounds at any point, mark the neighbor as invalid.
  3. By building these tables once, the main simulation loop can count all eight neighbors of each cell (four orthogonal plus four diagonals) by at most two table lookups—greatly improving performance over naive neighbor-search logic.
- **Key Design Choice:** Precomputation trades a modest upfront cost and extra memory for very fast, constant-time neighbor retrieval during each generation. This ensures the simulation scales efficiently even for large board sizes (e.g. 500×500 or bigger).

3.4 Simulation Engine

- **Responsibility:** Perform one generation of Game of Life given a boolean grid and the neighbor-lookup tables. Repeat this up to specified milestones (e.g. 1, 10, 100, 1000).
- **Approach:**
  1. For each cell (r, c):
     - Retrieve its four orthogonal neighbor coordinates via the precomputed tables. If a neighbor index is invalid, skip.
     - For each orthogonal neighbor, also retrieve its two adjacent diagonal neighbors (again via the lookup tables). This yields up to eight positions total.
     - Count how many of those eight positions are alive (lookup in the boolean array).
     - Apply the classic Conway rules:
       - If currently alive and neighbor-count $\in \{2,3\}$, it remains alive; else it dies.
       - If currently dead and neighbor-count $= 3$, it becomes alive; else it stays dead.
  2. Write the result into a fresh boolean array, then swap or overwrite for the next iteration.
  3. At each milestone, invoke the image-saving routine to write a black/white PNG of the current boolean grid.
- **Key Design Considerations:**
  - Avoid nested loops that repeatedly check color or re-compute wormhole jumps—this is all encapsulated in the neighbor tables.
  - Use vectorized or block-level operations where possible (e.g. NumPy boolean masking) for even faster neighbor-count accumulation on large boards.
  - Only store two copies of the board in memory (current and next), to keep space overhead minimal.

3.5 CLI Orchestration

- **Responsibility:** Provide a simple command-line interface to run the entire pipeline on any named "example" folder or arbitrary input directory.
- **Approach:**
  1. Accept a single positional argument, `input_folder`, which may be:
     - A path string pointing directly to a folder containing the three required PNGs.
     - An alias like `example-0`, which is resolved to `./examples/example-0` if that folder exists.
  2. Parse an optional `--milestones` list (e.g. {1,10,100,1000}).
  3. In code, first resolve the path (alias or direct).
  4. Invoke the I/O routines to load the starting grid and tunnel images, then build neighbor tables.

5. Sequentially iterate the simulation from generation 0 up to each requested milestone, saving PNGs at exactly those iterations.
6. Print concise, user-friendly messages indicating progress (e.g. "Loaded board 50×50 with 12 horizontal wormholes", "Saved 10.png successfully").
7. On error (missing file, invalid tunnel pairing, etc.), report the issue and exit cleanly with a nonzero status.

- **Key Design Choice:** Keep the CLI layer as thin as possible. All heavy logic resides in importable modules (I/O, wormhole, lookup, engine), so this main script simply sequences operations and handles argument parsing / error messaging.

4 Approach & Iterations

The project was delivered in a sequence of incremental steps, each validated by targeted tests. Below is a chronological summary:

4.1 Iteration 1: Basic Game of Life (No Wormholes)

1. Implemented a minimal "step" function that reads a hardcoded boolean grid and applies Conway's rules.
2. Wrote unit tests for known patterns (e.g. blinker oscillator, still-life block).
3. Verified results on small boards (e.g. 5×5) by visual inspection / textual representation.

4.2 Iteration 2: Image I/O Integration

1. Added routines to load a black-and-white PNG and convert it into a boolean NumPy array.
2. Added routines to save a boolean array back into a black/white PNG.
3. Updated tests to supply small PNG fixtures and confirm correct boolean conversion.
4. Verified that loading and saving preserved pattern shapes exactly.

4.3 Iteration 3: Wormhole Parsing

1. Introduced `_pairs_from_bitmap` to group pixels by color, pair them deterministically, and produce a map of portal endpoints.
2. Wrote unit tests to confirm that:
   - Exactly two pixels of the same non-black color → one bidirectional tunnel.
   - Odd counts or more-than-two count → appropriate error is raised.
   - Both horizontal and vertical tunnels can be loaded in a single call to `load_wormholes`.
3. Verified sample tunnel images from `examples/example-4` to confirm correct pairing.

4.4 Iteration 4: Neighbor Lookup Table Construction

1. Defined a sentinel value (`INVALID = -1`) for out-of-bounds or missing neighbors.
2. Built routines to assemble `nbr_r`, `nbr_c` arrays of shape $(H, W, 4)$ in a single nested loop over all cells.
3. Handled multi-stage teleportation:
   - If stepping off a horizontal portal point, teleport first, then step.
   - If stepping onto a vertical portal point, teleport again before finalizing neighbor.
4. Wrote unit tests for:
   - A small 3×3 board with no tunnels → exact neighbor indices at edges and center.
   - A 3×3 board with a single horizontal portal → verify correct remapping of "right" and "left" moves.
   - A 3×3 board with a vertical portal → verify correct remapping of "up" and "down" moves.

4.5 Iteration 5: Engine Integration with Wormholes

1. Updated the `step` function to use neighbor tables for counting all eight neighbors (local+).
2. Ensured that diagonal neighbor retrieval used two table lookups: orthogonal step → diagonal step.
3. Wrote integration tests that:
   - Compare a 3×3 starting board through one generation when no tunnels are present (should match classic Game of Life).

- Compare a custom scenario where a single live cell at a wormhole endpoint survives due to a remote neighbor connection.

4. Verified correctness by overlaying generated versus expected PNG images (pixel-level comparison).

4.6 Iteration 6: CLI Snapshotting

1. Built a lightweight `cli.py` to:
   - Resolve input folder aliases.
   - Invoke `load_boolean_board`, `load_wormholes`, `build_tables`, and `step` in proper sequence.
   - Save output PNGs at specified milestones.
2. Wrote tests for:
   - Folder-resolution logic (alias vs direct path).
   - End-to-end "2×2 all-alive" scenario to confirm that `1.png` is written with correct dimensions.
3. Manually tested on `examples/example-0` through `examples/example-4` to confirm correct output files.

4.7 Iteration 7: Final Polish & Documentation

1. Organized code into a clean directory layout (`src/`, `tests/`, `examples/`, `problems/`).
2. Added a comprehensive `README.md` (and this LaTeX documentation) explaining usage, architecture, and design tradeoffs.
3. Ensured all tests pass under `pytest`, and linted code with `flake8` and `mypy`.
4. Prepared a ZIP for submission, including: `src/`, `tests/`, `examples/`, `problems/`, `README.md`, and `requirements.txt`.

5 Testing Strategy & Quality Assurance

5.1 Unit Tests

- Each module—`io`, `wormhole`, `lookup`, `engine`, `loader_check`, `cli`—has its own test file under `tests/`.
- Core behaviors verified:
  - PNG loading correctness for "exact-white" threshold.
  - Correct pairing of wormhole endpoints and error handling on malformed tunnel images.
  - Neighbor table accuracy on small handcrafted boards (with and without tunnels).
  - Game-of-Life evolution on canonical patterns (blinker, block, underpopulation).
  - CLI folder resolution and output file creation.
- Tests rely on small, self-contained PNG fixtures that deliberately exercise edge cases (e.g. an odd number of same-color pixels).

5.2 Integration Tests

- A dedicated `test_wormhole_io_integration.py` verifies the full pipeline:
  1. Load a known starting position.
  2. Load perfectly black tunnel images (no portals).
  3. Build neighbor tables and run one iteration.
  4. Confirm that the result matches the classic Game of Life outcome by comparing to an expected boolean matrix.
- Additional manual integration steps:
  - Run `python -m src.cli examples/example-X` for each provided example.
  - Pixel-level comparison of generated PNGs against the supplied `X.png` files in each example directory.

5.3 Performance Considerations

- Neighbor-lookup tables ensure that each generation's neighbor counts require only fixed-time array lookups rather than repeated color-checks or expensive searches.
- Memory usage is optimized by storing only two boolean grids in memory at once (current and next generation) and by representing wormhole maps as simple dictionaries.
- Typical performance on a 200×200 board with moderate tunnel density:
  - Table construction: ∼0.02 s

  - Single generation step: ∼0.04 s
  - Saving one PNG: ∼0.01 s
- These numbers were obtained on a mid-range laptop (Intel i5, 8 GB RAM). Larger boards (e.g. 500×500) remain responsive (∼0.2 s per generation).

6 Usage & Submission

6.1 How to Run

1. Ensure Python 3.8+ and dependencies (`numpy`, `Pillow`) are installed.
2. From the project root, run:
   `python -m src.cli examples/example-0`

   This will read `examples/example-0/starting_position.png`, `horizontal_tunnel.png`, `vertical_tunnel.png`, simulate 1000 generations, and write `1.png`, `10.png`, `100.png`, `1000.png` under `examples/example-0/`.
3. To specify custom milestones:
   `python -m src.cli examples/example-4 --milestones 5 20 50`

4. To verify input files without running the simulation:
   `python -m src.loader_check examples/example-3`

7 GIF Generation

This utility script easily visualizes the four milestone PNGs (1.png, 10.png, 100.png, and 1000.png) as a looping animation. The script itself lives under `src/make_gif.py`. After you run the main simulation (which produces the four snapshots in each `examples/example-*/` folder), simply invoke this script to bundle those images into a single animated GIF called `all_output.gif`.

Where It Lives

- `src/make_gif.py` — a standalone Python script (using only Pillow) that looks for `1.png`, `10.png`, `100.png`, and `1000.png` inside a given example folder and writes out an animated GIF.

How to Run It

1. Make sure you have already generated all four milestone images in the target folder. For example:

   $$\texttt{python} - \texttt{m src.cli examples/example} - \texttt{0}$$

   will produce `1.png`, `10.png`, `100.png`, `1000.png` under `examples/example-0/`.
2. From the project root, run the GIF script in one of two ways:
   - To generate a GIF in every subfolder under `examples/`, use:

     $$\texttt{python src/make\_gif.py} - -\texttt{examples} - \texttt{dir examples/}$$

     This will scan each `examples/example-N/` directory and create an `all_output.gif` there.
   - To generate a GIF for a single example folder only, use:

     $$\texttt{python src/make\_gif.py} - -\texttt{folder examples/example} - \texttt{2}$$

     (Replace `example-2` with whichever folder you want to process.)
3. By default, the script names the output GIF `all_output.gif` and displays each frame for half a second. You can optionally override the GIF's filename or per-frame delay via command-line flags documented in the script header.
4. After it finishes, open `examples/example-N/all_output.gif` in any image viewer or web browser to watch the four milestones animate in order.

Note: Since `Pillow` is already listed in `requirements.txt`, no extra dependencies are required. If you ever need to install or update Pillow, run:

```
pip install --upgrade Pillow
```

8 Conclusion

The *Game of Life with Wormholes* project demonstrates a clean, modular approach to extending a classic simulation with nonlocal interactions. Key takeaways:

- Precomputing neighbor tables dramatically simplifies wormhole logic during each generation.
- Clear separation of concerns (I/O, parsing, lookup, engine, CLI, testing) yields maintainable, testable code.
- Automated tests ensure robustness and catch regressions early.
- The final CLI is straightforward to use—even non-technical reviewers can verify results by opening PNGs in any image viewer.

This implementation balances algorithmic efficiency with readability and thorough documentation, making it an excellent showcase of both technical depth and software engineering best practices.