Trait-AC

Multi-Trait Cellular Automata

A Generalized Approach for Modeling

Complex Multi-Agent Systems

Technical Documentation

Théo Deygas

January 29, 2026

This project began as a school assignment completed with a classmate. However, the topic was open-ended regarding automata, and my contributions go well beyond what was required. For this reason, this document contains only my own work and does not include my colleague's contributions, which I do not claim as my own. I consider this a personal project rather than a school project.

A demonstration video is provided to help illustrate the project.
The video demonstrates the case study presented in Chapter 8 (Case Study: The Energy-Charge-Phase Model).

link to the video

Contents

Part I

Simulation Library

Chapter 1

Introduction

1.1 What is a cellular automaton?

A cellular automaton is a mathematical model used to simulate complex systems. Imagine a grid, like a giant chessboard, where each square (called a "cell") can be in a certain state. At each step of the simulation, the state of each cell is updated according to precise rules that take into account the state of neighboring cells.

The most famous example is John Conway's "Game of Life": each cell is either alive or dead, and its future state depends on the number of living neighbors it has. Despite very simple rules, this system can produce extraordinarily complex behaviors: structures that move, reproduce, or even perform calculations.

1.2 Why Trait-AC?

Classical cellular automata constitute a fundamental tool for modeling many phenomena, particularly when each cell can be described by a **discrete state** (for example, alive or dead, or a small number of possible states). These models have enabled the study of rich dynamics from simple local rules.

Trait-AC follows in this tradition while expanding the modeling framework. Instead of limiting each cell to a single state, Trait-AC allows each cell to possess **multiple traits**, each represented by a **floating-point value (real number)**. These traits can evolve over time according to local rules, either independently or in a coupled manner.

For example, in an urban population simulation, a cell (representing an individual or household) can be characterized by several continuous traits:

- **Income**: floating-point value between 0 and 100,000
- **Average age**: floating-point value between 0 and 100
- **Education level**: normalized floating-point value between 0 and 1
- **Life satisfaction**: floating-point value between -1 and 1

The main advantage of this approach lies in the possibility of **making these traits interact**. Some traits can influence the evolution of other traits (for example, education level influencing income), while others can remain independent. This flexibility allows for building more expressive and modular models.

Thanks to this multi-trait and continuous representation, Trait-AC opens the door to simulating various phenomena such as:

- the coevolution of social or economic characteristics,
- the diffusion of behaviors or cultures,

- epidemic dynamics integrating different immunity levels,
- or systems where multiple local factors interact simultaneously.

Trait-AC does not replace classical cellular automata, but generalizes them, proposing a framework capable of representing more complex systems while preserving the fundamental principle of local interactions.

1.3 Two important parts of Trait-AC

1.3.1 A sophisticated movement system

Unlike traditional cellular automata where cells are fixed, Trait-AC allows "agents" (non-empty cells) to move across the grid. This opens the door to simulations where individuals can migrate to more favorable areas, flee overcrowding, or group together with similar agents.

The major technical challenge is conflict management: what happens when two agents want to occupy the same cell? Trait-AC implements a four-phase conflict resolution algorithm that guarantees consistent behavior while remaining performant through parallelization.

1.3.2 An extensible architecture through macros

Adding a new update rule or movement behavior requires only a single line of code thanks to the Rust macro system. This ease of extension does not sacrifice performance: the compiler generates optimized code for each rule and movement.

Chapter 2

Design Philosophy

2.1 Performance without compromise

Cellular automata simulation is inherently computationally expensive. A grid of 3000×3000 cells contains 9 million cells, and each cell must be updated at each time step. If the simulation runs for 1000 time steps, this represents 9 billion operations.

To achieve acceptable performance, Trait-AC employs several optimization strategies:

Parallelization with Rayon Work is automatically distributed across all processor cores. Grid rows are processed in parallel during rule application, and movement phases use atomic operations to avoid expensive locks.

Cache-optimized data structure Instead of storing traits cell by cell (which would scatter memory accesses), Trait-AC uses a "Structure of Arrays" (SoA) layout. All values of trait 0 are stored consecutively, then all values of trait 1, etc. This allows the processor to efficiently load data into cache when processing a given trait.

Optimized path for static movement When static movement is chosen, all movement logic is bypassed via a simple memory buffer swap.

2.2 Extensibility through macros

One of Trait-AC's design goals was to enable simple addition of new **rules** and new **movements**, without complicating the architecture or introducing significant runtime overhead. These two mechanisms play a central role in the simulation and are called repeatedly at very large scale.

For both rules and movements, Trait-AC relies on an **enumeration** (`enum`) that groups all available variants. Each variant of this enumeration corresponds to a specific behavior (a rule or a type of movement). These enumerations constitute a form of static "catalog" of behaviors supported by the library.

Rust macros are used to automatically generate, at compile time:

- the **enumeration** containing all rule or movement variants,
- the conversion functions between **textual names** (used in configuration files) and enumeration variants,
- the accessor functions that associate each variant with its corresponding **implementation function**.

Once the configuration is loaded, selected rules and movements are stored as **simple function pointers**, identical in their operation for both systems. The behavior choice is therefore made only once during initialization, then reused efficiently at each simulation iteration.

Adding a new rule or movement follows exactly the same process: simply implement the corresponding function and add a line in the macro. The rest of the code (enumeration, conversions, dispatch) is generated automatically, ensuring consistency between rules and movements while maintaining good performance in critical loops.

Chapter 3

General Architecture

3.1 Module Overview

The library is organized into five distinct modules, each with a clear responsibility:

Module	File	Responsibility
Grid	`grid.rs`	Main data structure containing the state of all cells
Neighborhood	`neighborhood.rs`	Definition of neighborhoods used for rules and movements
Rules	`rules.rs`	Trait update logic at each time step
Movement	`movement.rs`	Agent movement system with conflict resolution
Utils	`utils.rs`	Utility functions for display and debugging

Table 3.1: Library module organization

3.2 Data Flow in the Simulation

Each time step of the simulation proceeds in two distinct phases:

Phase 1 — Rule Application For each non-empty cell in the grid, new trait values are calculated based on current values and those of neighbors. Results are written to a temporary grid to prevent updates from affecting ongoing calculations (double buffering technique).

Phase 2 — Movement Application Agents decide where they want to move based on their environment. A conflict resolution algorithm determines who can actually move. The final grid is constructed by moving agent data to their new positions.

This decoupling between rules and movements allows great flexibility: one can simulate classic automata (rules only, static movement), particle systems (movement only, trivial rules), or complex hybrid models.

Chapter 4

Grid Module: The Central Data Structure

4.1 Concept

The Grid module defines the structure that contains the entire state of the simulation. It is the heart of the library: all other operations read from or write to this structure.

A Trait-AC grid is not simply a 2D array. It must efficiently manage:

- Potentially very large dimensions (several thousand cells per side)
- Multiple traits per cell (up to 9 in the default configuration)
- Empty cells (sparse grids where only a fraction is occupied)

4.2 Memory Representation

The `Grid` structure contains the following fields:

Field	**Type**	**Description**
`width`	`usize`	Grid width in number of cells
`height`	`usize`	Grid height in number of cells
`num_cells`	`usize`	Total number of cells (width × height)
`num_traits`	`usize`	Number of traits per cell
`data`	`Vec<f32>`	Values of all traits, stored contiguously
`is_empty`	`Vec<bool>`	Vector indicating which cells are empty

Table 4.1: Grid structure fields

The choice of a "Structure of Arrays" layout for the `data` field deserves explanation. In a naive "Array of Structures" approach, data would be stored as follows: for each cell, store all its traits consecutively, then move to the next cell. The problem is that when processing a particular trait (for example, calculating the average of trait 0 in the neighborhood), the processor must skip over other traits, wasting memory bandwidth.

Trait-AC uses the opposite layout: all values of trait 0 are stored first, then all values of trait 1, and so on. Thus, when the processor loads a cache line, it contains only data useful for the current calculation. To access trait t of the cell at index idx, the formula is:

$$\texttt{data}[t \times \texttt{num\_cells} + idx]$$

4.3 Creating a Grid

The library offers two constructors depending on needs:

The simple constructor `new(width, height, num_traits)` creates a fully filled grid where each trait is randomly initialized between 0 and 1. This is useful for quick tests or simulations where all cells are active.

The advanced constructor `new_with_density(width, height, fill_percentage, num_traits, trait_ranges)` offers fine control. The `fill_percentage` parameter determines what proportion of cells will be occupied (between 0.0 and 1.0), while `trait_ranges` specifies the minimum and maximum bounds for each trait.

For example, to create a 100×100 grid where 30% of cells are occupied, with an "age" trait between 0 and 100 and a "wealth" trait between 0 and 50,000:

Listing 4.1: Rust excerpt

```rust
let trait_ranges = vec![(0.0, 100.0), (0.0, 50000.0)];
let grid = Grid::new_with_density(100, 100, 0.3, 2, &trait_ranges);
```

Chapter 5

Neighborhood Module: Defining the Neighborhood

5.1 Concept

In a cellular automaton, a cell's behavior depends on its "neighborhood" (the surrounding cells). But the exact definition of the neighborhood can vary considerably depending on the type of simulation.

The **Moore** neighborhood includes the 8 adjacent cells (horizontal, vertical, and diagonal). This is the classic Game of Life neighborhood.

The **Von Neumann** neighborhood includes only the 4 directly adjacent cells (up, down, left, right), without diagonals. It produces more "orthogonal" behaviors.

Some simulations require larger neighborhoods (radius 2, 3 or more) or custom shapes (cross, ring, etc.).

5.2 Mask Representation

Trait-AC represents neighborhoods using a mask: a small matrix where 1 indicates that a cell is part of the neighborhood and 0 indicates that it is excluded. The center of the mask corresponds to the cell whose update is being calculated.

For a standard Moore neighborhood (3×3 with all neighbors), the mask is a 3×3 matrix filled with 1s. The central cell at position $(1, 1)$ represents the evaluated cell.

For a Von Neumann neighborhood, only the cardinal positions (up, down, left, right) and the center are 1, the corners are 0.

For simulations requiring greater range, one can use 5×5, 7×7 or larger masks, precisely defining which positions influence the calculation.

For neighborhoods without a clear center, the cell of the central square (or pair) with the lowest indices (row, column) will be designated as the center.

5.3 Two Independent Neighborhoods

Trait-AC uses two distinct neighborhoods in its configuration:

The **trait neighborhood** is used during rule application to determine which cells influence trait updates. For example, to calculate an average, the values of traits in this neighborhood are considered.

The **movement neighborhood** is used during the movement phase to determine which cells the agent can "see" to make its movement decision.

This separation allows for interesting behaviors. For example, an agent could calculate its traits based on a small local neighborhood (3×3) but make movement decisions by observing a larger area (7×7), simulating an ability to "see far" but only be influenced by immediate neighbors.

Chapter 6

Rules Module: Update Rules

6.1 Concept

Rules define how traits evolve over time. At each time step, for each non-empty cell, each trait is recalculated according to its associated rule.

A rule is a pure function that receives as input the index of the trait to calculate, the cell's position (row and column), the neighborhood to consider, and the current complete state of the grid. It returns the new trait value as a floating-point number.

This abstraction allows implementing very varied behaviors: simple rules like neighbor averaging, classic rules like Conway's Game of Life, or arbitrarily complex custom rules.

6.2 The Registry System

The `RulesRegistry` maintains the correspondence between each trait and its rule. It is a structure that stores, for each trait index, the function that must be called to calculate its new value.

When creating the registry, one specifies which rule to apply to each trait. This flexibility allows simulations where different traits evolve according to different dynamics. For example, a "time" trait could increase with a very simple rule while a "population" trait would follow more complex growth rules.

6.3 Available Rules

The library includes several pre-defined rules:

Rule	**Behavior**
`Static`	No change: the cell keeps its current trait value
`Average`	The trait becomes the average of non-empty neighbors' traits
`WeightedAverage`	Weighted average of neighbors, with weight decreasing with distance
`Maximum`	The trait takes the maximum value among the cell and its neighbors
`Minimum`	The trait takes the minimum value among the cell and its neighbors
`Diffusion`	Diffusion toward the neighbor average with progressive trait decay
`Conway`	Application of classic Conway's Game of Life rules
`ConwayOptimized`	Optimized branchless implementation of Conway's Game of Life

Table 6.1: Available pre-defined rules

Other rules can be easily added according to simulation needs.

6.4 Adding a New Rule: The Power of Macros

One of Trait-AC's strengths is its ease of extension. The system uses a Rust macro called `define_rules!` that automatically generates all the code necessary to integrate a new rule.

To add a new rule, the process involves two simple steps:

First step Write the function that implements the rule logic. This function must follow the standard signature (receive trait index, position, neighborhood, and grid) and return the new trait value.

Second step Register the rule in the `define_rules!` macro. Simply add a single line with three elements: the enumeration variant name, the textual name (used in configuration files), and the reference to the implementation function.

The macro automatically generates the corresponding enumeration variant, conversion from textual name (which allows specifying rules in TOML files), and dispatch to the implementation function.

This approach offers the best of both worlds: the ease of addition of an extensible architecture, and the performance of direct calls since the compiler can optimize the generated code.

Chapter 7

Movement Module: The Movement System

7.1 Concept

The movement module is the most sophisticated part of Trait-AC. It allows agents to move across the grid, transforming the static cellular automaton into a dynamic multi-agent system.

Each agent independently decides where it wants to go based on its environment (cells in its movement neighborhood). But this freedom of movement creates coordination problems: what happens when multiple agents want to go to the same place? Or when agent A wants to go where B is, and B wants to go where A is?

Trait-AC implements a conflict resolution algorithm that guarantees three essential properties:

- **Consistency** ensures that each cell contains at most one agent after movement.
- **Fairness** guarantees that conflicts are resolved randomly, without favoring certain grid positions.
- **Priority to immobility** means that an agent choosing to stay in place always has priority over an agent wanting to take its place.

7.2 The Four-Phase Algorithm

Movement resolution proceeds in four distinct phases, each with a specific role:

7.2.1 Phase 1 — Intention Declaration (executed in parallel)

In this first phase, each agent examines its neighborhood and calculates its desired destination by calling its movement function. This function analyzes the state of surrounding cells and returns a displacement as a pair $(\Delta_{\text{row}}, \Delta_{\text{column}})$. For example, $(0, 1)$ means "go one cell to the right", $(-1, 0)$ means "go one cell up".

Each agent wishing to move (whose destination differs from its current position) makes a "bid" for its target cell. The bid is simply a random number generated for this agent and this time step, which will serve as a tiebreaker criterion in case of conflict. These bids are stored in a thread-safe manner using atomic operations, allowing all agents to be processed in parallel without locks.

7.2.2 Phase 2 — Pruning (executed in parallel)

For each cell that received multiple bids (multiple agents want to go there), only the agent with the highest bid is retained as a candidate. Other "losing" agents have their intention reset: they will have to stay in place.

This phase eliminates direct conflicts where multiple agents target the same empty cell. After this phase, each empty cell has at most one candidate.

7.2.3 Phase 3 — Chain Resolution (executed sequentially)

This phase handles more complex cases of dependencies between movements. Consider the following situation: agent A wants to go to position B, but there is already an agent B at that position. Agent B, for its part, wants to go to position C which is empty.

The question is: can A move? The answer depends on whether B moves or not. If B successfully goes to C, then position B becomes free and A can go there. But if B cannot move (because C is occupied by someone staying for example), then A must also stay in place.

The algorithm uses a depth-first search (DFS) to resolve these dependency chains. Starting from an agent wishing to move, it follows the chain of intentions until it encounters either an empty cell (in which case all agents in the chain can move), or a cell whose occupant stays in place (in which case the chain is blocked and no agent moves).

It is important to note that at the end of phase 2, each cell can have at most one external agent (i.e., not from that cell) wishing to move there. This implies that the intention chain will never lead to a cell newly occupied by an agent that has already moved during the same time step update.

An important special case deserves attention: if an agent decides to stay in place from the start, it always has priority. Even if another agent wants its position and won the phase 1 bid, the immobile agent wins the conflict. This rule ensures that no one can be forced out of their position.

7.2.4 Phase 4 — Grid Construction (executed in parallel)

Once all final destinations are determined, the new grid is constructed. For each position in the resulting grid, we now know where the data comes from: either from an agent that moved to this position, or the position remains empty.

Trait data is copied from source positions to destination positions. The `is_empty` vector is updated to reflect new empty and occupied positions.

7.3 Available Movements

The library includes several pre-defined movement behaviors:

Movement	**Behavior**
`Static`	No movement — optimized path that short-circuits the entire algorithm
`Random`	Random direction at each time step
`Gradient`	Moves toward areas with higher values of a given trait
`AvoidCrowding`	Flees densely populated areas

Table 7.1: Available pre-defined movements

The `Static` movement deserves special mention: when selected, the library detects that no movement is necessary and short-circuits the four resolution phases entirely. Only a pointer swap between memory buffers is performed, which is nearly instantaneous.

7.4 Adding a New Movement

As with rules, adding a movement is trivial thanks to the macro system. The `define_movements!` macro works exactly like `define_rules!`: simply write the movement function that returns the desired displacement, then add a line in the macro to register it.

The movement function receives the agent's current position, the movement neighborhood, and the grid state. It returns a pair $(\Delta_{\text{row}}, \Delta_{\text{column}})$ indicating the desired displacement. The library then handles all the conflict resolution work.

Chapter 8

Case Study: The Energy-Charge-Phase Model

8.1 Model Presentation

To concretely illustrate the possibilities offered by Trait-AC, let's examine a particularly rich model using three interdependent traits: **Energy**, **Charge**, and **Phase**. This model demonstrates one of the library's major strengths: the ability to create systems where traits mutually influence each other during their updates, producing complex emergent behaviors from relatively simple rules.

8.2 The Three Traits and Their Meaning

8.2.1 Energy (Heat)

Energy can be understood as temperature. Hot zones are active, cold zones are dormant. As in real physics, heat tends to diffuse and eventually dissipates over time.

8.2.2 Charge (Polarity)

Charge represents a kind of belonging or orientation, comparable to a team color or magnetic pole. Each individual is located somewhere on a spectrum from one polarity to another. In this model, similar charges attract while opposite charges repel.

8.2.3 Phase (Internal Clock)

Each individual has an internal rhythm that constantly advances, like a heartbeat. When two individuals are "in phase," their clocks are synchronized. When they are "out of phase," their rhythms clash.

8.3 Update Rules

What makes this model fascinating is that each trait depends on the others for its update, creating a cycle of dependencies where the three traits work together.

8.3.1 Energy Rule: Friction Creates Heat

When neighbors have different phases (desynchronized), they create friction. This friction generates energy. Energy also diffuses to neighbors (hot zones warm cold zones) and slowly decays over time, simulating entropy.

Consequence: Synchronized groups become cold. Chaotic groups stay hot. Heat propagates in waves.

8.3.2 Charge Rule: Flow Toward Energy

Charge evolves based on energy gradients. Individuals adopt the polarity of their high-energy neighbors and reject that of their low-energy neighbors. It's comparable to social influence: one wants to resemble successful individuals (high energy) and differentiate from those who struggle (low energy).

Consequence: Charge patterns follow energy waves. As heat moves, polarity changes in its wake.

8.3.3 Phase Rule: Synchronization

Here lies the key element of the model: synchronization or desynchronization with a neighbor depends on charge similarity.

- **Similar charges** → phases converge (synchronization)
- **Opposite charges** → phases diverge (anti-synchronization)

Consequence: Clusters of similar polarity synchronize their rhythms. Neighbors with opposite phases create friction.

8.4 Movement: Pulsed and Intentional

The movement system associated with this model has two constraints that make it interesting:

Phase-cadenced movement An agent can only move during a specific window of its internal clock, as if it could only walk on the "beat." This creates waves of movement rather than constant, disordered mixing.

Charge attraction When an agent moves, it is attracted to similar-charge individuals with high energy, and repelled by opposite-charge individuals.

8.5 The Interdependence Cycle

These three traits form an **inseparable trio**:

1. **Energy** is generated by **Phase** differences (friction)
2. **Charge** evolves based on **Energy** gradients (social influence)
3. **Phase** synchronizes or desynchronizes based on **Charge** similarity (tribalism)

This cycle creates a feedback loop where each trait influences the next, which influences the next, which returns to influence the first. None of the three can be considered in isolation.

8.6 Observed Emergent Phenomena

When the simulation is launched with this configuration, visually very different phenomena appear for each trait:

Energy presents explosive patterns, with high-intensity zones that propagate in waves and interact.

Charge develops polarized territories, with boundaries that evolve with energy waves.

Phase shows synchronized clusters within same-polarity groups, with turbulence zones at interfaces.

What is remarkable is that these three visually distinct phenomena are actually facets of the same unified system.

8.7 Phase's Role as Internal Clock

In this model, Phase plays a crucial role. To understand this, imagine stopping Phase updates (by setting its rule to "static").

Energy also stops, and Charge too. The entire system freezes. This is because without Phase evolution, there is no friction between neighbors, therefore no energy generation, therefore no gradient to make charge evolve.

If Phase is reactivated, the entire system instantly comes back to life.

8.8 Differentiated Behaviors Depending on Which Trait is Stopped

To understand these behaviors well, we must first explain the phenomenon of **stabilization**. At simulation startup, Phase is chaotic, agents' internal clocks are desynchronized. This desynchronization generates enormous friction between neighbors, and therefore an energy explosion. Progressively, phases synchronize within same-charge groups, friction decreases, and energy diffuses and drastically decays. The system then reaches a stable state: a diffuse background of energy that maintains itself at a low but constant level.

If Phase is frozen after stabilization Everything freezes. At this stage, Phase is relatively homogeneous from one agent to another (clocks are synchronized), so there is little friction and constant, fixed energy creation. Energy therefore remains stable, which keeps Charge stable. The system is in equilibrium.

If Phase is frozen before stabilization The result is radically different. Since Phase is still chaotic, there is enormous friction between neighbors. This friction produces so much energy that the entire grid quickly saturates at the maximum possible value (here 1.0). Without the possibility of synchronization, the system remains stuck in this state of permanent maximum heat.

If Charge is frozen The system continues but with very different dynamics. Without Charge evolution, Phase loses its notion of "tribe", it no longer knows who to synchronize with and who to desynchronize from. The friction dynamics change radically, and energy becomes abundant because synchronization can no longer establish itself to calm the system.

If Energy is frozen Charge no longer has a gradient to follow for evolution (no energy displacement). It therefore freezes at its current values.

8.9 From Trio to Duo: Experimentation

One of Trait-AC's major advantages is the ease of experimentation. What happens if Phase is assigned the same rule as Charge?

In this configuration, Phase and Charge converge to the same values. The three-beat cycle becomes a **duo**: Energy depends on Charge, and Charge depends on Energy.

This simplification produces unexpected results: patterns resembling fluid simulations, with **gas bubbles that form and disappear**, behavior difficult to predict from the individual rules.

8.10 Case Study Conclusion

This Energy-Charge-Phase model perfectly illustrates Trait-AC's philosophy:

Interdependent traits Traits form coupled systems where each influences the others according to rules inspired by real phenomena (thermodynamics, social influence, synchronization).

Easy experimentation Changing a rule or disabling a trait requires only a minor modification, allowing exploration of the possibility space and understanding of each component's role.

Emergent phenomena Relatively simple rules, combined creatively, produce complex and visually rich behaviors: energy waves, polarized territories, synchronized clusters, fluid simulations.

Intuitive understanding By anchoring traits in understandable analogies (heat, group belonging, heartbeat), the model remains accessible while producing a wealth of behaviors.

Trait-AC's goal is precisely to provide a framework where it is trivial to add new rules and movements to observe what emerges from their interactions, with multiple models working in concert to produce a final, unique, and often surprising phenomenon.

Chapter 9

Simulation Library Configuration

9.1 The TOML Configuration File

Trait-AC uses a **TOML** file to configure simulations. This format is both readable and easy to edit, and allows modifying all simulation parameters **without recompiling the code**. This makes it particularly suitable for experimentation, as well as for use by people who may not have programming skills.

It is important to note that the **number of traits (`num_traits`) is not dynamic**: it directly impacts the size of data structures used during simulation. However, other parameters such as `active_mask`, `initialisation_ranges`, and `rules` can contain **more values than necessary**. Excess values will simply not be used and have **no impact on performance**.

The configuration file controls several aspects of the simulation:

- The **grid dimensions**, which define the size of the simulated world (width and height in number of cells), as well as the **initial fill density**, i.e., the proportion of occupied cells at startup.
- The **simulation duration**, expressed in number of time steps.
- The **trait configuration**, including the number of traits per cell, active traits (which will actually be updated), and value ranges used for their initialization.
- The **behaviors**, which determine the rule applied to each trait as well as the type of movement used by all agents.
- The **neighborhoods**, defined as distinct masks for rules and for movement.

9.2 Configuration Example

Here is an example configuration file, accompanied by explanatory comments:

Listing 9.1: Example config.toml file

```toml
# === GRID DIMENSIONS ===
# A 3000x3000 cell grid, i.e., 9 million cells total
grid_width = 3000
grid_height = 3000

# A density of 1.0 means 100% of cells are occupied at start
grid_density = 1.0

# === SIMULATION DURATION ===
# The simulation will run for 100 time steps
timesteps = 100

# === TRAIT CONFIGURATION ===
# Number of traits used in this simulation
num_traits = 1

# Activation mask: 1 = active trait, 0 = ignored trait
active_mask = [
    1, 0, 0,
    0, 0, 0,
    0, 0, 0,
]

# Initialization ranges [min, max] for each trait
initialisation_ranges = [
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
]

# === RULES AND MOVEMENT ===
rules = [
    "conway optimized", "conway optimized", "conway optimized",
    "conway optimized", "conway optimized", "conway optimized",
    "conway optimized", "conway optimized", "conway optimized",
]
movement = "static"

# === NEIGHBORHOODS ===
neighborhood_traits_mask = [
    [1, 1, 1],
    [1, 1, 1],
    [1, 1, 1],
]

neighborhood_mvt_mask = [
    [1, 1, 1],
    [1, 1, 1],
    [1, 1, 1],
]
```

Chapter 10

Usage Guide

10.1 Workflow Overview

Using the library follows a standard multi-step pattern. First, create the grid with the desired dimensions and traits. Then, define the neighborhoods to be used for rules and movement. Next, configure the rule and movement registries, associating the desired behaviors. Finally, prepare a second buffer for double buffering and launch the simulation loop.

10.2 The Simulation Loop

Once initialization is complete, the simulation consists of repeating both phases (rules then movement) for each time step. At each iteration, rules are first applied to each active trait, calculating new values in parallel and storing results in the temporary grid. Then, movements are applied via the movement registry, which handles all conflict resolution and grid updating.

Double buffering is managed transparently: after the `apply_movement` call, the main grid contains the new state and is ready for the next time step.

Chapter 11

Simulation Library Performance

11.1 Typical Metrics

On a modern machine with multiple cores, Trait-AC achieves remarkable performance. For a 3000×3000 cell grid with Game of Life rules and no movement, the library can process over 100 time steps per second, i.e., over 900 million cell updates per second.

This performance is achieved through the combination of optimizations described earlier: parallelization across all available cores, cache-optimized memory layout, and fast paths for common cases like static movement.

11.2 Optimization Tips

To get the best performance with Trait-AC, several best practices can be followed:

Use static movement when possible If your simulation doesn't need movement, use `Movement::Static`. This completely short-circuits the conflict resolution algorithm, saving considerable time.

Limit active traits Only traits marked in `active_mask` are processed at each simulation step. Disable traits that are not necessary for your simulation.

Additional tip: Reducing the total number of traits (`num_traits`) decreases the size of data structures allocated for each cell, which reduces memory usage and speeds up calculation loops.

Optimize rules and movement Rule and movement functions represent the most expensive parts of the simulation. Choose simple rules when sufficient and limit the use of complex movements.

Choose appropriate neighborhoods A larger neighborhood means more cells to examine for each update. Use the smallest neighborhood that produces the desired behavior.

Adjust density Sparse grids (with a low density of occupied cells) are processed faster, because empty cells benefit from an optimized calculation path.

Chapter 12

Simulation Library Dependencies

The library relies on several Rust crates from the ecosystem:

Crate	**Usage**
`rand`	Random number generation for initialization and conflict resolution
`rayon`	Automatic parallelization of loops across all available cores
`serde`	Serialization and deserialization of configuration structures
`toml`	Parsing configuration files in TOML format

Table 12.1: Simulation library dependencies

Part II

Graphical User Interface (UI)

Chapter 13

Introduction to the Graphical Interface

13.1 UI Library Objective

The **Trait-AC UI** library provides an interactive graphical interface to visualize and control cellular automata simulations created with the Trait-AC library. It allows real-time observation of trait evolution on the grid, on-the-fly parameter modification, and easy experimentation with different configurations.

The interface is designed to be both performant (capable of smoothly displaying grids of several million cells) and intuitive (allowing interactive exploration without requiring recompilation).

Important note: The user interface requires trait values to be between 0.0 and 1.0 for proper display. Update rules must therefore ensure values stay within this range, or normalize them before display.

13.2 Technologies Used

The interface is built with **eframe/egui**, a Rust framework for creating native and web graphical interfaces. Grid rendering uses **OpenGL** (via glow) for optimal performance, with custom shaders to apply different color palettes to simulation data.

Chapter 14

Interface Architecture

14.1 Overview

The application is organized into three main areas:

The control panel (left) Contains all simulation parameters, play/pause, speed, grid dimensions, movement type, trait activation, and rules associated with each trait.

The visualization area (center) Displays the simulation grid with the selected trait, colored according to the chosen palette. Supports zoom and scrolling to explore large grids.

The statistics panel (right) Displays visualization options (displayed trait, color palette, cell size) and real-time statistics on each active trait.

14.2 Rendering Pipeline

Grid rendering follows an optimized pipeline to efficiently handle millions of cells:

1. **Data extraction**: Values of the selected trait are extracted from the simulation grid.
2. **Grayscale conversion**: Floating-point values are converted to bytes (0-255) in a parallelized manner, accounting for empty cells.
3. **Upload to GPU**: The grayscale buffer is sent to an OpenGL texture.
4. **Palette application**: A fragment shader applies the chosen color palette (Viridis, Plasma, etc.) to the texture in real time.
5. **Display**: The colored texture is rendered in the visualization area with zoom and scrolling management.

This approach transfers only one byte per cell to the GPU, rather than 4 bytes (RGBA), which significantly reduces memory bandwidth for large grids.

Chapter 15

GPU Renderer Module: Accelerated Rendering

15.1 Concept

The `GPURenderer` is the heart of the visualization system. It manages communication with OpenGL to efficiently display the simulation grid. Its role is to take raw simulation data (floating-point values per cell) and transform them into a colored image on screen.

15.2 Shaders and Color Palettes

Rendering uses GLSL shaders (OpenGL Shading Language) to apply color palettes directly on the GPU. This approach has two major advantages:

Performance The value → color conversion is performed by the GPU in parallel for all visible pixels, rather than by the CPU for all cells.

Flexibility Changing color palettes only requires changing the active shader, without recalculating anything on the CPU side.

The vertex shader transforms egui interface coordinates to normalized OpenGL coordinates, automatically handling zoom and scrolling:

Listing 15.1: Vertex shader excerpt

```
// Converting egui coordinates to OpenGL
vec2 local = a_pos - u_rect_min;
vec2 pos = local / u_rect_size;
pos = pos * 2.0 - 1.0;
pos.y = -pos.y;
```

Each color palette is implemented as a separate fragment shader. For example, the Viridis shader implements interpolation between 11 control points of the scientific Viridis palette:

Listing 15.2: Viridis shader excerpt

```
vec3 viridis(float t) {
    // 11 control points of the Viridis palette
    const vec3 c0 = vec3(0.267004, 0.004874, 0.329415);
    // ... other points ...
    const vec3 c10 = vec3(0.993248, 0.906157, 0.143936);

    // Linear interpolation between points
    float x = clamp(t, 0.0, 1.0) * 10.0;
    int i = int(floor(x));
    float f = fract(x);

    // Returns interpolated color
    if (i == 0) return mix(c0, c1, f);
    // ...
}
```

15.3 Texture Management

The GPU texture is updated in an optimized manner:

Conditional resizing The texture is only reallocated when grid dimensions change. Normal updates use `tex_sub_image_2d` which is much faster.

Minimal format The texture uses the `R8` format (single channel, one byte per pixel) rather than RGBA, reducing transfer size by a factor of 4.

Parallelized upload Conversion from floating-point values to bytes is performed in parallel on the CPU via Rayon before upload.

15.4 Adding a New Color Palette

The palette system uses the same macro architecture as the simulation library's rules and movements. To add a new palette:

1. Create a new shader file in `shaders/` (e.g., `magma.glsl`)
2. Implement the color mapping function
3. Register the palette in the `define_color_schemes!` macro

This ease of extension allows for easy experimentation with different visualizations.

Chapter 16

Simulation Controls

16.1 Play and Pause

The interface offers several ways to control simulation execution:

- **Play/Pause button**: Toggles between continuous execution and stop.
- **Space bar**: Keyboard shortcut for Play/Pause.
- **Step button**: Advances the simulation by a single time step, useful for observing transitions in detail.
- **Reset button**: Reinitializes the grid with new random values according to the current configuration.
- **Delete key**: Keyboard shortcut for the Reset button.
- **Randomize button**: Generates a new random configuration without modifying parameters.

16.2 Speed Control

The "Steps/sec" slider controls the number of simulation steps per second. The configurable range goes from 1 to 10,000 steps per second.

The interface uses an intelligent temporal accumulation system that:

- Maintains a stable pace even if rendering varies
- Avoids lag accumulation if the simulation is too slow
- Dynamically adapts the number of steps per frame to maintain fluidity

16.3 Grid Configuration

Grid dimensions can be modified on the fly:

- **Width**: From 3 to 7,500 cells
- **Height**: From 3 to 7,500 cells
- **Density**: From 1% to 100% of occupied cells

Modifying these parameters automatically reinitializes the simulation with the new dimensions.

Chapter 17

Trait and Rule Configuration in the Interface

17.1 Trait Activation

The interface displays a grid of checkboxes to activate or deactivate each trait. Only active traits are:

- Updated at each simulation step
- Displayable in the visualization area
- Included in statistics

This feature allows isolating certain traits to observe their behavior, or temporarily disabling traits to understand their influence on the global system (as demonstrated in the Energy-Charge-Phase case study).

17.2 Rule Selection

For each active trait, a dropdown menu allows selecting the update rule. All rules registered in the simulation library are automatically available in the interface.

Changing a rule for a trait takes effect immediately, without requiring a restart. This allows real-time experimentation: observe behavior with one rule, change it, and see how the system evolves differently.

17.3 Movement Selection

A global dropdown menu allows choosing the movement type for all agents. As with rules, all movements registered in the library are available.

Chapter 18

Visualization

18.1 Displayed Trait Selection

The "Trait" menu allows choosing which trait is visualized on the grid. Only active traits appear in this menu.

Changing the displayed trait is instantaneous, the data is already in memory, only the GPU texture is updated with the new values.

18.2 Color Palettes

Several scientific palettes are available:

Palette	**Description**
Viridis	Palette from dark purple to light yellow
Plasma	Warm palette, from dark purple to bright orange, with saturated transitions
Grayscale	Classic gradient from black to white
RedBlue	Two-color gradient: blue for low values, red for high values

Table 18.1: Available color palettes

These palettes, like Viridis and Plasma, are designed to be perceptually uniform: equal differences in values produce approximately equal differences in perceived color. This avoids visual artifacts where certain ranges would seem artificially more contrasted.

Conversely, simple palettes like RedBlue are not perceptually uniform, and certain intermediate values may appear darker or more saturated than others.

18.3 Base Color

The "Base color for non-empty cells" slider controls the minimum value displayed for non-empty cells. With a value of 0, a cell with a trait at 0 will appear black. With a value of 0.2, it will appear in the color corresponding to 20% of the palette.

This option is useful for visually distinguishing empty cells (always black) from occupied cells with a very low value.

18.4 Zoom and Navigation

The visualization area supports smooth navigation:

Zoom Ctrl + mouse wheel (or trackpad). Zoom is centered on the cursor position, allowing direct zoom into an area of interest.

Scrolling Mouse wheel (works in all four directions), or click and drag (trackpad or mouse).

Cell size The "Cell Size" slider allows precise control of zoom level, from 0.1 to 100 pixels per cell.

18.5 Value Display

When cells are large enough (20+ pixels by default), the "Show Values" option displays the numerical value of each cell directly on the grid. Text color automatically adapts (white on dark background, black on light background) to remain readable.

Chapter 19

Statistics

19.1 Statistics Panel

When the "Show Statistics" option is enabled, the right panel displays detailed statistics for each active trait:

- **Density**: Proportion of non-empty cells having a non-zero value for this trait.
- **Minimum and Maximum**: Extreme values among non-empty cells.
- **Average**: Average over all cells (including zeros).
- **Average (non-zero)**: Average only over cells with a positive value, often more representative.

These statistics are recalculated at each frame and allow tracking global system evolution without having to visually inspect each cell.

19.2 Benchmark Mode

The interface can operate in "timed simulation" mode to measure performance. In this mode:

- The simulation runs until a defined number of time steps
- At the end, performance statistics are displayed (steps/second, cells/second)
- The application closes automatically

This mode is useful for comparing the performance of different configurations or optimizations.

Chapter 20

Interface Configuration

20.1 Configuration File

The application uses a `config.toml` file that centralizes all parameters. This file is read at startup and defines the initial state of the simulation and interface.

20.2 Grid Parameters

```toml
# Initial dimensions
grid_width = 500
grid_height = 500
grid_density = 1.0
num_traits = 3

# Slider limits
grid_width_min = 3
grid_width_max = 7500
grid_height_min = 3
grid_height_max = 7500
```

20.3 Simulation Parameters

```toml
# Simulation speed
steps_per_second = 25.0
steps_per_second_min = 1.0
steps_per_second_max = 10000.0

# Benchmark mode (optional)
timed_simulation = false
timestep_max = 100
```

20.4 Visualization Parameters

```
# Cell size
cell_size = 0.5
cell_size_min = 0.1
cell_size_max = 100.0

# Value display
show_values = false
show_values_minimum_cell_size = 20.0

# Statistics
show_stats = false

# Colors
color_scheme = "viridis"
base_color_not_empty = 0.0
```

20.5 Trait Configuration

```
# Active traits (1 = active, 0 = inactive)
active_mask = [1, 1, 1, 0, 0, 0, 0, 0, 0]
initial_selected_trait = 0

# Initialization ranges
initialisation_ranges = [
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
    [0.0, 1.0], [0.0, 1.0], [0.0, 1.0],
]

# Rules per trait
rules = [
    "energy", "charge", "phase",
    "conway optimized", "conway optimized", "conway optimized",
    "conway optimized", "conway optimized", "conway optimized",
]

# Global movement
movement = "energy charge phase"
```

20.6 Neighborhoods

```
# Neighborhood for rules (5x5 in this example)
neighborhood_traits_mask = [
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
]

# Neighborhood for movement
neighborhood_mvt_mask = [
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
]
```

Chapter 21

Interface Performance

21.1 Implemented Optimizations

The interface is designed to remain fluid even with very large grids:

GPU rendering Cell coloring is performed entirely on the GPU, freeing the CPU for simulation.

Conditional updates The texture is only updated when data changes (new time step or change of displayed trait).

CPU parallelization Conversion of values to grayscale uses Rayon to exploit all available cores.

Partial rendering Only the visible portion of the grid is rendered, not the entirety.

Conditional value display Numerical values are only calculated and displayed for cells visible on screen.

21.2 Recommendations

For best performance:

- Disable "Show Values" for large grids or fast simulations
- Disable "Show Statistics" if statistics are not needed
- Use appropriate zoom, a very wide zoom on a large grid can slow down rendering

Chapter 22

Interface Dependencies

Crate	Usage
`eframe`	Application framework with egui support
`egui`	Immediate mode user interface library
`glow`	OpenGL bindings for Rust
`egui_glow`	egui integration with OpenGL
`rayon`	Parallelization of CPU calculations
`trait_ac`	Simulation library (engine)

Table 22.1: Graphical interface dependencies

Chapter 23

Performance Benchmarks

This chapter presents a comprehensive analysis of Trait-AC's performance evolution across multiple development versions. The benchmarks demonstrate the cumulative impact of the optimization strategies described throughout this documentation.

23.1 Benchmark Methodology

23.1.1 Test Configuration

All benchmarks were executed with a standardized configuration to ensure fair comparison across versions:

- **Grid Size:** 3000×3000 cells (9 million cells total)
- **Timesteps:** 100 iterations per benchmark run
- **Rule:** Conway's Game of Life (standard and optimized variants)
- **Movement:** Static (no agent movement)
- **Traits:** Single trait active
- **Density:** 100% cell occupancy

This configuration was chosen because:

- The 3000×3000 grid size is large enough to stress-test performance while remaining practical for repeated testing.
- Conway's Game of Life provides a well-understood baseline rule that exercises neighbor counting logic and makes it easier to compare performance with other projects.
- Static movement isolates library and rendering performance from conflict resolution overhead (to ensure a fair comparison with other projects).
- 100 timesteps provides sufficient iterations to obtain results before reaching a static state.

23.1.2 Benchmark Process

The benchmarking process followed a rigorous methodology:

Version Isolation Each version was benchmarked from its corresponding Git commit. To ensure reproducibility, minor modifications were applied to standardize the test harness across versions. These modifications are documented in the `benchmark_diffs/` folder, with a summary in `benchmark_diffs/SUMMARY.md`.

Dual-Target Testing Each version was tested twice:

1. `trait_ac` (library only): Measures pure simulation performance without any rendering overhead.
2. `trait_ac_ui` (with GUI): Measures simulation performance including graphical rendering overhead. The GUI runs in benchmark mode, which automatically closes after the specified number of timesteps.

Metrics Collected For each run, the following metrics were recorded:

- **Execution time**: Total wall-clock time for 100 timesteps. The timer starts *after* all variables and data structures are initialized, immediately before the first timestep calculation. This isolates simulation performance from initialization overhead.
- **Timesteps/second**: Simulation throughput
- **Cells/second**: Raw computational throughput (timesteps/sec × 9,000,000 cells)

Environment All benchmarks were executed on the same hardware configuration to ensure comparability. The `-release` flag was used for all Rust compilations to enable full optimizations.

23.2 Library Performance Results

The core simulation library (`trait_ac`) showed dramatic performance improvements across versions:

Version	Date	Time (s)	Steps/s	MCells/s	vs Base	vs Prev
v0	2025-12-19	171.700	0.58	5.24	—	—
v1	2025-12-22	26.465	3.78	34.01	6.49×	6.49×
v2	2025-12-23	26.466	3.78	34.01	6.49×	1.00×
v3	2025-12-24	12.287	8.14	73.25	13.97×	2.15×
v4	2025-12-25	1.855	53.90	485.08	92.54×	6.62×
v5	2025-12-26	1.639	61.03	549.25	104.79×	1.13×
v6	2026-01-07	0.901	111.04	999.32	190.65×	1.82×
v7	2026-01-27	0.674	148.36	1335.21	254.73×	1.34×

Table 23.1: Library performance across versions

Summary: The final version achieves a **254.7× speedup** compared to the baseline, reducing execution time from 171.7 seconds to 0.674 seconds.

23.2.1 Analysis of Key Improvements

The library optimization evolution can be grouped into three distinct phases:

Phase 1: Parallelization (v0–v1) The initial major improvement came from introducing parallel processing with Rayon. The naive sequential implementation was replaced with parallel iteration over grid rows, immediately exploiting all available CPU cores. This single change provided a 6.49× speedup with minimal code modifications.

Phase 2: Memory and allocation optimization (v2–v3) Version 2 focused exclusively on UI optimizations, leaving the library unchanged. Version 3 introduced two key algorithmic changes:

- **Double buffering**: Instead of creating a new grid for each timestep, two grids are maintained and swapped between iterations, eliminating per-timestep allocation overhead.
- **Fixed neighborhood indexing**: The `Neighborhood` structure was redesigned from storing per-cell references to neighbors (expensive allocation for each cell update) to a shared index-based structure where neighbor coordinates are computed on-demand via simple arithmetic.

Phase 3: Data layout and algorithmic optimization (v4–v7) This phase brought the most dramatic improvements through fundamental redesigns:

Version 4 (6.62× speedup) introduced the most impactful single optimization:

- **Structure of Arrays (SoA) layout**: The grid changed from `Vec<Vec<Cell»` (where each Cell contains traits) to separate contiguous arrays for traits and empty flags, dramatically improving cache locality.
- **Fast path for static movement**: When movement is disabled, the grid swap is performed without computing cell movements.
- **Optimized Conway implementation**: The rule was rewritten to use pre-calculated neighbor coordinates with branchless wrapping (avoiding expensive modulo operations) and direct memory access via `unsafe` pointer arithmetic.

Version 5 (1.13× speedup) saw only minor library improvements, as development focus shifted to the GPU rendering pipeline.

Version 6 (1.82× speedup) brought several refinements:

- **Improved data layout**: The trait storage changed from a fixed-size array of vectors (`[Vec<f32>; 9]`) to a single contiguous allocation, further improving memory access patterns.
- **Configuration files and macros**: Added support for external configuration and macro-based rule/movement registration.
- **Further Conway optimizations**: Additional refinements to the branchless implementation.

Version 7 (1.34× speedup) included the final optimizations:

- **Removal of `bitvec`**: The `BitVec<u64, Lsb0>` structure for empty cell tracking was replaced with a simpler `Vec<bool>`. Despite being "less sophisticated," this proved faster due to reduced bit manipulation overhead.

- **Lookup table for Conway rules**: The final Conway implementation counts alive neighbors using pointer arithmetic, then indexes into a precomputed 18-element result table (9 entries for dead cells, 9 for alive), eliminating all conditional branches from the hot path.

23.3 Graphical Interface Performance Results

The GUI version (`trait_ac_ui`) includes both simulation and rendering costs:

Version	Date	Time (s)	Steps/s	MCells/s	vs Base	vs Prev
v0	2025-12-19	540.806	0.18	1.66	—	—
v1	2025-12-22	320.368	0.31	2.81	1.69×	1.69×
v2	2025-12-23	32.625	3.07	27.59	16.58×	9.82×
v3	2025-12-24	21.195	4.72	42.46	25.52×	1.54×
v4	2025-12-25	7.371	13.57	122.10	73.37×	2.88×
v5	2025-12-26	4.277	23.38	210.41	126.44×	1.72×
v6	2026-01-07	2.262	44.22	397.94	239.12×	1.89×
v7	2026-01-27	0.913	109.59	986.28	592.65×	2.48×

Table 23.2: GUI performance across versions

Summary: The final version achieves a **592.7× speedup** compared to the baseline, reducing execution time from 540.8 seconds to 0.913 seconds.

The GUI improvements are even more dramatic than the library improvements because the rendering pipeline underwent a complete architectural overhaul and benefits from the improvements of the library, as detailed in the next section.

23.4 UI vs Library Time Breakdown

To understand where time is spent, we can decompose the GUI execution time into library computation and pure UI overhead:

$$\text{Pure UI Time} = \text{GUI Total Time} - \text{Library Time}$$

Version	Date	UI Total	Lib	Pure UI	Lib %	UI %	Pure UI Speedup
v0	2025-12-19	540.806	171.700	369.106	31.7%	68.3%	—
v1	2025-12-22	320.368	26.465	293.903	8.3%	91.7%	1.26×
v2	2025-12-23	32.625	26.466	6.159	81.1%	18.9%	59.93×
v3	2025-12-24	21.195	12.287	8.908	58.0%	42.0%	41.43×
v4	2025-12-25	7.371	1.855	5.516	25.2%	74.8%	66.92×
v5	2025-12-26	4.277	1.639	2.639	38.3%	61.7%	139.88×
v6	2026-01-07	2.262	0.901	1.361	39.8%	60.2%	271.19×
v7	2026-01-27	0.913	0.674	0.238	73.9%	26.1%	1547.82×

Table 23.3: UI overhead breakdown by version

Pure UI Overhead Improvement: 1547.8× speedup (from 369.1 seconds to 0.238 seconds)

23.4.1 Analysis of UI Optimization Phases

The UI overhead evolution reveals three distinct phases:

Phase 1: CPU-bound rendering (v0–v1) In the initial versions, rendering was performed entirely on the CPU using egui's immediate mode drawing. Each frame required:

- Iterating over all 9 million cells individually
- For each cell: computing the color, creating a rectangle, and issuing a `rect_filled` draw call
- Additionally drawing a stroke border around each cell
- Optionally rendering text for cell values

The UI consumed 68–92% of total execution time. Version 1 saw a modest 1.26× UI speedup from the parallelized library, but the rendering remained the dominant bottleneck.

Phase 2: Texture-based rendering (v2) Version 2 introduced a fundamental architectural change—switching from individual draw calls to texture-based rendering. Instead of 9 million draw calls, the grid is now rendered as a single texture: pixel colors are computed in parallel using Rayon, stored in a `ColorImage`, and uploaded to the GPU via `load_texture`. The entire grid is then displayed with a single `egui::Image` widget. This change produced an immediate **47.7× speedup** in pure UI overhead.

Phase 3: GPU shader rendering (v5) Version 5 moved color palette application from CPU to GPU using custom OpenGL shaders. The CPU now uploads only grayscale values (1 byte per cell = 9 MB) instead of full RGBA colors (4 bytes per cell = 36 MB), and a GPU shader applies the color palette in parallel for all pixels. The renderer handles viewport clipping via texture coordinates, only drawing the visible portion of the grid.

Phase 4: Final optimizations (v6–v7) Version 6 refactored the UI codebase into separate modules (`color_scheme.rs`, `gpu_renderer.rs`, `config.rs`) and added configuration file support.

Version 7 achieved a significant pure UI speedup (5.72×) through a simple but effective change: **making statistics display a toggle** and disabling it during benchmarks. The statistics computation and rendering had been adding significant overhead that was unnecessary for performance measurement.

23.4.2 Bottleneck Shift

An important observation from Table 23.3 is the **bottleneck shift**:

- **v0–v1**: UI was the bottleneck (68–92% of time)

- **v2–v3**: Library became the bottleneck (58–81% of time)
- **v4–v6**: UI became the bottleneck again (60–75% of time) after the major changes in the library
- **v7**: Finally, the library is dominant (73.9%), but both are now very fast

This pattern is typical of optimization work: fixing one bottleneck reveals the next.

23.5 Version History

For reference, here are the Git commits corresponding to each benchmarked version:

Version	Commit	Date	Description
v0	`d3c5067`	2025-12-19	update readme.md
v1	`d76218f`	2025-12-22	performance boost
v2	`6002d09`	2025-12-23	performance boost + refactor
v3	`4e7ee8c`	2025-12-24	perf boost (less allocation + new approach)
v4	`c1d129c`	2025-12-25	refactor + better perf
v5	`de2a57f`	2025-12-26	gpu rendering
v6	`d5b73d0`	2026-01-07	refactor, cleanup, fixed small bugs, added conf...
v7	`95c8875`	2026-01-27	removed bitvec, updated conway_optimized, upda...

Table 23.4: Git commit history for benchmarked versions

23.6 Overall Summary

Component	Baseline	Final	Speedup
Library (`trait_ac`)	171.70s	0.674s	254.7×
GUI (`trait_ac_ui`)	540.81s	0.913s	592.7×
Pure UI Overhead	369.11s	0.238s	1547.8×

Table 23.5: Overall performance improvement summary

23.6.1 Peak Throughput

The final version (v7) achieves:

- **Library**: 1,335.21 million cells/second (**1.34 billion cells/second**)
- **With GUI**: 986.28 million cells/second (∼**1 billion cells/second**)

To put this in perspective, assuming 1 billion cells/second: a cell is processed every 0.000000001s, i.e., every 1 nanosecond.

To be fair, there is parallelization using Rayon and in my testing 20 cores are used, so the actual processing would be: a cell is processed every 0.00000002s, i.e., every 20 nanoseconds, for each core of the processor.

23.6.2 Optimization Lessons Learned

The benchmark journey illustrates several important principles:

Parallelization first The easiest and most impactful optimization was adding Rayon parallelization (v1). With minimal code changes, this immediately leveraged all available CPU cores.

Data structure design matters The transition from Array of Structures to Structure of Arrays (v4) and the subsequent refinement to a single contiguous allocation (v6) provided substantial speedups by improving cache locality and memory access patterns.

Algorithm matters more than micro-optimization The branchless Conway implementation with lookup table (v4, refined in v7) provided important speedups. Understanding CPU architecture (branch prediction penalties) led to algorithmic improvements that dwarfed low-level tweaks.

Move work to the GPU for rendering The CPU-to-texture rendering change (v2) was transformative for UI performance, providing a 47.7× speedup in pure UI overhead. Moving color palette computation to GPU shaders (v5) further improved performance.

Profile before optimizing The UI/Library breakdown analysis revealed when the bottleneck shifted, guiding where to focus optimization efforts. Flamegraph profiling was used extensively throughout development to identify hot paths and pinpoint specific functions consuming the most CPU time. This data-driven approach ensured that optimization efforts targeted actual bottlenecks rather than assumptions. Without this analysis, effort might have been wasted optimizing components that were not bottlenecks.

Sometimes simpler is faster Replacing `BitVec` with `Vec<bool>` (v7) improved performance despite being a "less sophisticated" data structure. The overhead of bit manipulation outweighed the memory savings for this use case.

Don't forget the "obvious" optimizations Making statistics display optional (v7) provided a 5.72× pure UI speedup. Features that seem minor can have significant performance impact when executed millions of times per second.

General Conclusion

The Trait-AC project proposes a generalization of classical cellular automata, enabling the modeling of complex multi-agent systems through a multi-trait representation. This project consists of two complementary libraries that together offer a complete environment for exploration and experimentation.

The Simulation Library

It provides a performant and extensible engine. Its design balances three often contradictory objectives:

- **Performance** is ensured by automatic parallelization, cache-optimized data structures, and fast paths for common cases. The library can process hundreds of millions of cells per second on modern hardware.
- **Flexibility** comes from multiple traits per cell, configurable neighborhoods, and clear separation between rules and movements. This architecture allows simulating both classic automata and multi-agent systems.
- **Extensibility** is guaranteed by the macro system that allows easily adding new rules and movements in a few lines of code, without sacrificing performance.

The UI Library

It complements the simulation engine by providing an interactive interface to explore cellular automata. It emphasizes:

- **Experimentation**: Modify parameters, change rules, activate/deactivate traits—everything is possible on the fly without recompilation.
- **Visualization**: Scientific color palettes and GPU rendering allow clear observation of emergent phenomena.
- **Performance**: The architecture is optimized to handle grids of several million cells while maintaining a responsive interface.
- **Extensibility**: Like the simulation library, adding new color palettes follows the same simple macro pattern.

Together, these two libraries constitute the ideal tool for exploring the possibilities offered by multi-trait cellular automata, whether for research, education, or simply the pleasure of discovering the emergent behaviors of complex systems.