UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 1728

SIMULATION AND VISUALIZATION OF CELLULAR AUTOMATA IN DISCRETE SPACE

Vito Vrbić

Zagreb, June 2025

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 1728

SIMULATION AND VISUALIZATION OF CELLULAR AUTOMATA IN DISCRETE SPACE

Vito Vrbić

Zagreb, June 2025

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 03 March 2025

BACHELOR THESIS ASSIGNMENT No. 1728

Student:	**Vito Vrbić (0036542047)**
Study:	Electrical Engineering and Information Technology and Computing
Module:	Computing
Mentor:	prof. Željka Mihajlović, PhD

Title: **Simulation and visualization of cellular automata in discrete space**

Description:

In computer graphics, an interesting problem is the procedural generation and simulation of cellular automata, which are used to model various phenomena, from textures to complex dynamic systems. For this, it is necessary to research cellular automata, their components, types, and applications. Analyze different methods of cellular automata simulation. Design, develop, and optimize a library for their simulation. Extend the library with an add-on for 3D visualization of cellular automata. Conduct testing on a set of examples and analyze and evaluate the achieved results. Consider the usability of the obtained results and possible directions for further development. Develop an appropriate software product using the C# programming language and OpenGL for the graphical interface. Make the results available online. Attach algorithms, source code, and results with necessary explanations and documentation. Cite the used literature and acknowledge received assistance.

Submission date: 23 June 2025

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 3. ožujka 2025.

ZAVRŠNI ZADATAK br. 1728

Pristupnik:	**Vito Vrbić (0036542047)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Simulacija i vizualizacija staničnih automata u diskretnom prostoru**

Opis zadatka:

U računalnoj grafici jedan od zanimljivih problema je proceduralno generiranje i simulacija staničnih automata, koji se koriste za modeliranje različitih fenomena, od tekstura do složenih dinamičkih sustava. Potrebno je istražiti stanične automate, njihove komponente, vrste i primjene. Analizirati različite metode simulacije staničnih automata. Osmisliti, razviti i optimizirati biblioteku za njihovu simulaciju. Proširiti biblioteku dodatkom za vizualizaciju 3D staničnih automata. Provesti testiranje na nizu primjera te analizirati i ocijeniti ostvarene rezultate. Razmotriti upotrebljivost dobivenih rezultata i moguće smjerove proširenja. Izraditi odgovarajući programski proizvod koristeći programski jezik C# i OpenGL za grafičko sučelje. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 23. lipnja 2025.

Contents

1 Introduction

1.1 Background and Motivation

Cellular automata, in the simplest of terms, are discrete n-dimensional systems made up of a grid of cells, each of which can, at a single time, be in one and only one of a few possible predefined states. The behaviour of each cell at a given time is determined by its current state, and the state of its neighbouring cells. While the rules of these systems are straightforward (e.g., if this cell is black, and two cells above this one are white, this one should turn white), the resulting behaviour can often be complex and highly unpredictable. In many ways, cellular automata can be thought of as a sort of game where the outcome isn't decided at the start but unfolds based on interactions among the cells as time progresses [1].

What makes cellular automata interesting is their ability to demonstrate a phenomenon known as 'emergence'. Defined in the Dictionary of Cambridge as "the fact of something becoming known or starting to exist", in this area, 'emergence' refers to the property of cellular automata to achieve the emergence of complex patterns and behaviours from very simple rules [2]. As an example, even though each individual cell follows a simple rule, the entire grid can display intricate, chaotic, and even life-like behaviour.

Due to this property, these mathematical systems have a wide range of applications in both scientific areas and creative ones. One of the most famous examples of cellular automata is Conway's Game of Life, which got its name due to the fact that it, ultimately, models the way life might evolve in a very abstract and simplified way. This automaton easily showcases emergence even at very simple initial grid layouts [3].

1.1.1 Cellular Automata in Computer Graphics

In the ever-growing field of computer graphics, cellular automata have proven themselves to be very useful tools in many different domains of the field. They've also proven themselves in connected fields—such as the video game industry [4].

Patterns and Textures

On their own, alongside neural networks, or with noise functions, cellular automata can be, and are, used for generating textures and patterns from limited rules, producing detail-rich results.

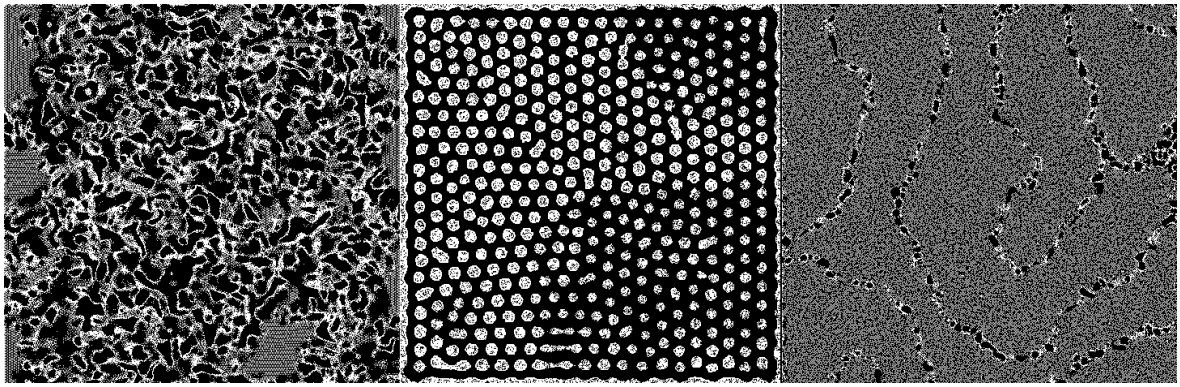

Figure 1.1: Textures generated using the Compound Cellular Automata Processing sketch by Tamas Karpati [5].

An example of textures generated with cellular automata without the addition of other tools/methods is shown in Figure 1.1. To still add complexity without making the rules too complex, adding noise in various ways (to the grid after each discrete time step or to the initial layout of the grid) can produce satisfactory results. This is shown in Figure 1.2.

A progressively more complex texture type—procedurally generated textures resembling real-life textures (e.g., surface of a leaf) can also be generated using neural cellular automata, as shown in Figure 1.3.

Procedural Generation

Cellular automata have usage in procedural generation as well. Not only are they useful in this area, but they have easily proven themselves to be a noteworthy functionality for

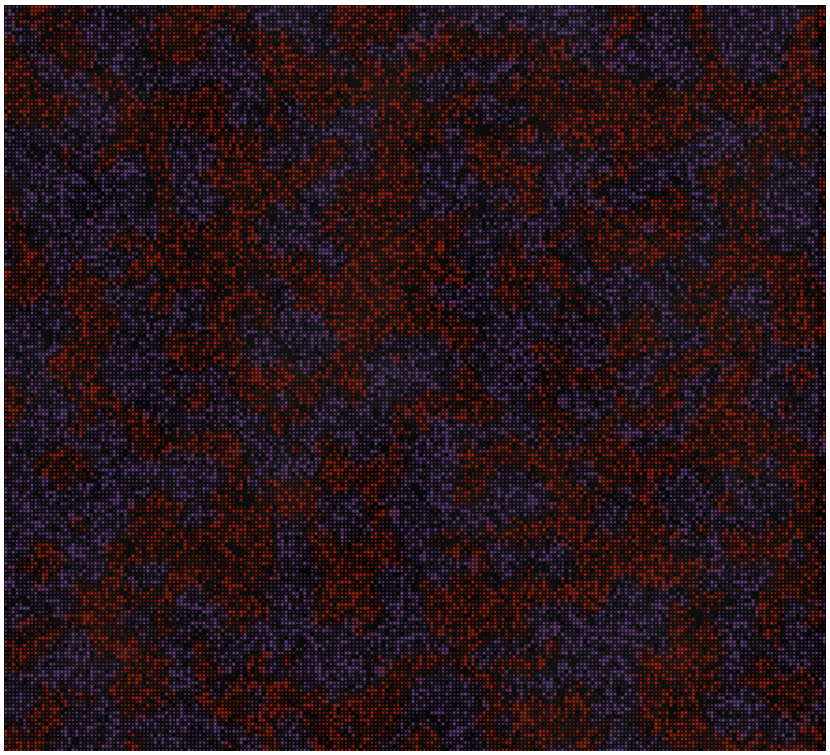

Figure 1.2: Texture generated in p5.js, using cellular automata, with Perlin noise attached.

both 2-dimensional and 3-dimensional procedural generation (e.g. generating landmass or cave systems, as showcased on Figure 1.4).

When modelling virtual environments—such as realistic terrain meshes, detail patterns on other meshes, or meshes depicting naturally occurring phenomena—cellular automata prove themselves as a valuable technique as well. An example of this can be seen in Figure 1.5.

Simulation of Physics

Due to their property of emergence, one of the most interesting ways cellular automata are used in the field of computer graphics (and adjacent) is for the simulation of physical phenomena (e.g., terrain erosion or fluid simulation). Cellular automata aren't only an efficient way to achieve this, but in case of proper rule definition, can also be a very accurate tool [9].

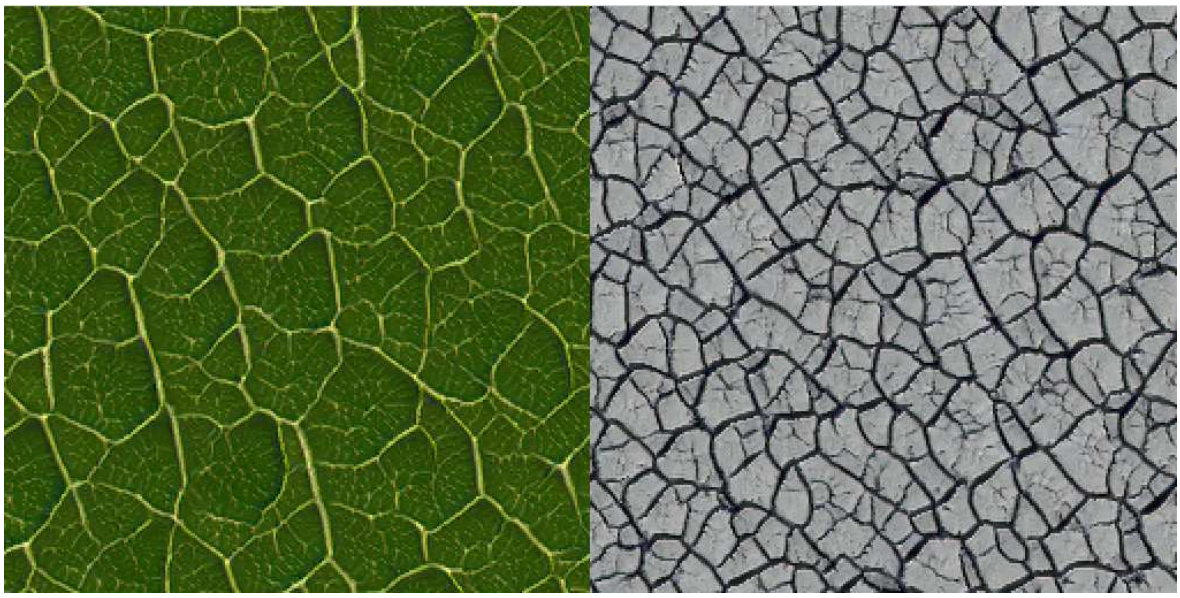

Figure 1.3: Texture generated using the cellular automaton model from Distill [6].

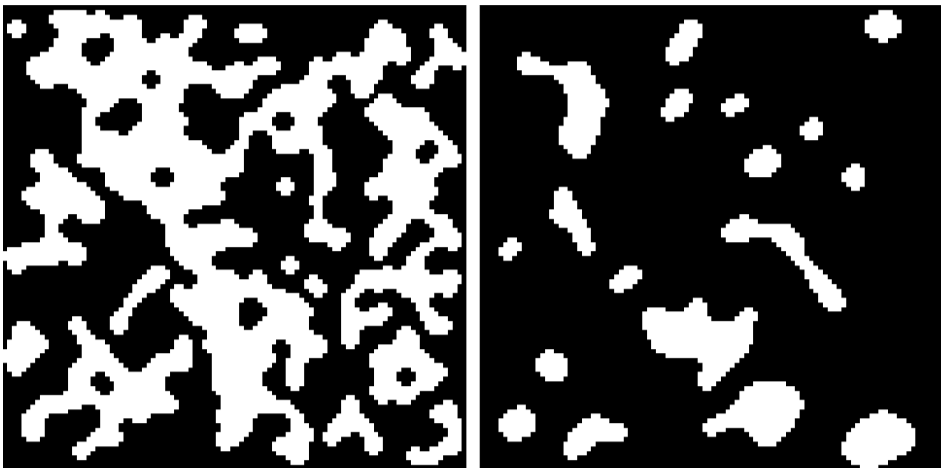

Figure 1.4: Images generated in Unity project by Zgeb [7].

1.2 Goals and Objectives

This thesis focuses on the design, development, evaluation, and optimization of a flexible and extensible C# library for the simulation of cellular automata on both small and large scales. The main goal is to create a tool that enables development, testing, and visualization of various cellular automata models in an accessible and efficient way.

Although many implementations of cellular automata already exist in academic or research contexts, this thesis seeks to emphasize clarity, modularity, and interactivity. The intention is for the library to serve not only as a demonstration tool but also as a platform for experimentation, learning, and further development.

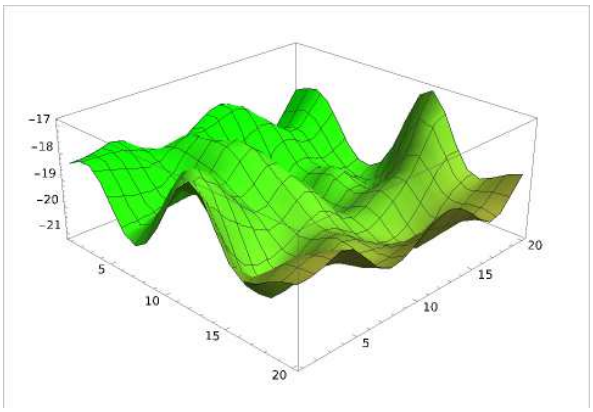

Figure 1.5: Terrain generated using the cellular automaton model from Cawley [8].

To achieve this, the library will be built with a focus on clean architecture, extensibility, and performance. Special attention will be given to enabling easy integration of custom rule sets and flexible grid configurations. Additionally, the system will include a visualization component to observe the evolution of automata in real time, providing valuable feedback for both development and exploration of cellular automata.

The project blends three key aspects:

- **Development:** Building a robust C# library that supports rule definition, state updating, and optimized grid management, with a clear and modular API.
- **Experimentation:** testing and comparison of different automaton rules, grid sizes, and boundary conditions to evaluate performance and behaviour.
- **Visualization:** Providing real-time graphical feedback of system evolution, with optional tools for pausing, stepping, and analysing rule behaviour over time.

The ultimate goal is to deliver a system that is both technically sound and useful in practice. Whether useful as a teaching tool, a research utility, or a base for artistic or scientific experimentation. The project focuses not just on accuracy and speed, but also on making the library easy to use and adaptable for a variety of future uses.

1.3 Structure of the Thesis

This thesis is organized into 6 chapters:

- **Chapter 1 – Introduction** provides background information, outlines the research

problem, and highlights the relevance of the topic in the fields of computer graphics and adjacent/related areas.

- **Chapter 2 – Theoretical Background** offers a theoretical overview of the concepts necessary for understanding the subsequent chapters in the thesis.
- **Chapter 3 – Simulation Library** describes the design of the C# library with UML diagrams and detailed explanations of all the core library components.
- **Chapter 4 – Visualization Library** describes the design of the C# library add-on with UML diagrams, detailed explanations of all the library components, and with an in-depth explanation of the rendering process behind the visualization.
- **Chapter 5 – Results** Showcases two cellular automata simulations and visualizations.
- **Chapter 6 – Conclusion and Future Work** summarizes the findings and achievements of the thesis, discusses areas where the implementation was faced with limitations, and suggests potential directions for future development and research.

2 Theoretical Background

2.1 Cellular Automata

For proper simulation of cellular automata (referred to as CA from now on), it is important to establish a definition that captures their structural and dynamic properties. While there is no single universally agreed-upon definition of a CA, most definitions share a common set of mathematical components which will be described.

2.1.1 Formal Definition of a Cellular Automaton

A cellular automaton of dimension d is a tuple:

$$(L, S, N, f)$$

L: Lattice of Cells

A discrete lattice/grid of cells, represented by integer d-tuples, each of which indexes a cell in space.

This lattice can be infinite, in which case we can say $L = \mathbb{Z}^d$; or it can be finite, in which case we can define the lattice as $L = (\mathbb{Z}/n\mathbb{Z})^d$ or $L = \{0, ..., n-1\}^d$. In this thesis, we will be working with cellular automata that have finite lattices because these are easier to simulate and visualize.

S: Finite Set of States

Each cell in the lattice can take a state from a finite set of states S. All finite sets of states can be represented using integers: $S = \{0, 1, ..., k-1\}$ where k is the number of unique states.

N: Set of Relative Positions (Neighbourhood)

A finite set of relative position vectors which determine which cells influence the update of a given cell is called a neighbourhood and is denoted as $N \subset \mathbb{Z}^d$.

f: Local Transition Rule Function

A (usually) deterministic function that maps the states of a cell's neighbourhood to the next state of the cell: $f : S^{|N|} \to S$. Some cellular automata define different local transition rule functions; one for each unique state in S. In this case f is defined as a function which calls other functions depending on the state of the cell for which the rule is called $f : S^{1+|N|} \to S$.

2.1.2 Derived Concepts

From the formal mathematical definition of a cellular automaton, two concepts can be derived which further explain how cellular automata behave through time.

c_t: Configuration at time t

A configuration at time t is a function $c_t : L \to S$ where $c_t(x)$ gives us the state of a cell $x \in L$ at time t.

F: Global Update Function

The global update function $F : S^L \to S^L$ takes the configuration of the entire system (at time t), and returns the next configuration of the system (at time $t+1$). This global update function defines the discrete time characteristic of every cellular automaton and is better defined in equation 2.1. If one uses multiple local transition rule functions, the global function is defined in equation 2.2.

$$F(c)(x) = f(c(x+n_1), ..., c(x+n_k)) \text{ for all } x \in L, \text{ where } k = |N|. \tag{2.1}$$

$$F(c)(x) = f(x, c(x+n_1), ..., c(x+n_k)) \text{ for all } x \in L, \text{ where } k = |N|. \tag{2.2}$$

2.1.3 Dimensionality

The dimensionality of a cellular automaton defines the structure of its spatial domain. A cellular automaton of dimension d operates on a d-dimensional lattice, where each cell is indexed by a d-tuple of integers. Common dimensionalities in theoretical works include:

- **1D**: $L = \mathbb{Z}$ or $L = \{0, ..., n-1\}$, used for linear automata.
- **2D**: $L = \mathbb{Z}^2$ or $L = \{0, ..., n-1\}^2$, used in grid-based models (e.g., Conway's Game of Life).
- **3D and Higher**: $L = \mathbb{Z}^d$, $d > 2$, useful for simulations in physical and biological systems such as crystal growth or lattice gases.

As the dimensionality increases:

- The size of the neighbourhood typically increases exponentially (but doesn't have to).
- The number of possible transition rules grows with the neighbourhood size.
- The complexity of dynamics and emergent behaviours tends to be higher—yet allows us to explore more interesting behaviours.

Despite this, all dimensionalities can be treated uniformly under the general framework of a cellular automaton defined on a lattice $L = \mathbb{Z}^d$.

2.1.4 Lattice Structures and Coordinate Systems

While most cellular automata use regular Cartesian grids, alternative lattice structures can be employed to model different interaction topologies. Common lattice types include:

- **Rectangular (Cartesian) Grids**: The standard choice, where each cell has fixed orthogonal neighbours (e.g., 4 or 8 in 2D).
- **Hexagonal Grids**: Common in simulations requiring rotational symmetry (e.g., some biological systems) [10].

- **Triangular Grids**: Used in some physics simulations or mesh-based environments [11].
- **Irregular or Sparse Lattices**: Modelled as graphs or masked arrays, where active cells are explicitly tracked.

In all cases, non-rectangular lattices can be represented on a rectangular grid $L = \mathbb{Z}^d$ by encoding geometry in the update rule f, using masking, or modifying the neighbourhood set N. This allows a general-purpose CA engine to simulate a wide range of lattice topologies.

2.1.5 Neighbourhood Variants

The neighbourhood N defines which cells influence the update of a given cell. It is specified as a finite set of relative position vectors in $\mathbb{Z}^d$ [1]. Different neighbourhood types are used depending on the dimension and the model's needs:

- **von Neumann Neighbourhood** (2D): Includes the four orthogonal neighbours (up, down, left, right).
- **Moore Neighbourhood** (2D): Includes all eight surrounding cells in a 3×3 grid.
- **Extended Moore**: Includes all cells within a certain radius r using either Manhattan or Chebyshev distance.
- **Radius-r Neighbourhoods** (1D): Includes r cells to the left and r to the right; total size is $2r + 1$.
- **Custom/Probabilistic Neighbourhoods**: Used in stochastic or irregular models; neighbours can be defined dynamically or probabilistically.

The choice of neighbourhood strongly affects the cellular automaton's dynamics. For example, using a Moore neighbourhood can lead to more complex behaviour than when using von Neumann neighbourhood..

2.1.6 Simulation Approaches

Simulating CA requires designing data structures and algorithms that can both efficiently update and efficiently store cell configurations across time steps. The choice of simulation approach often depends on the dimensionality, lattice structure, neighbourhood size, and sparsity of the grid.

Algorithmic Implementations

CA are typically implemented using:

- **Array-based representations** for dense grids, where the configuration is stored as a multidimensional array indexed by cell coordinates.
- **Hash tables or sparse arrays** for large grids with localized activity, reducing memory usage and improving performance [12].
- **Bitwise representations**, especially in 1D or 2D binary-state CAs, which allow for high-speed parallel updates via bit operations [13].
- **GPU-accelerated implementations**, which exploit data parallelism inherent in synchronous CA updates [14].

Computational Complexity and Performance

The computational complexity of simulating a cellular automaton is generally linear in the number of cells per time step, i.e., $\mathcal{O}(|L|)$ for one update if the neighbourhood size is constant. However:

- Complex local rules or larger neighbourhoods increase the per-cell computation cost.
- In high-dimensional or sparse automata, performance is heavily dependent on data structure optimization.
- Time complexity can grow in reversible or stochastic automata, or where global state tracking is required.

3 Simulation Library

3.1 Design Goals and Requirements

The design of the library's simulation component was guided by a mix of practical goals and established software engineering principles. It was built with both functionality and usability in mind—aiming to support a range of real-world use cases. It does this while staying flexible, efficient, and easy for developers to work with. The result is a system that makes it straightforward to define, run, and experiment with custom cellular automata.

3.1.1 Functional Requirements

The simulation component should meet the following functional requirements:

- Provide a clear, consistent, and concise API for developers to interact with when using the library.
- Support the definition of custom cellular automata and their essential elements.
- Allow for the simulation of any valid user-defined cellular automata.

3.1.2 Non-Functional Requirements

Several non-functional requirements have influenced architectural and design decisions:

- **Modularity**: The library should be composed of well-defined, independent components, reflecting the structure of CA (defined and explained in Chapter 2), to promote reusability.
- **Extensibility**: It should be straightforward to introduce new features or plug-ins without altering existing code. This should be achieved through the use of inter-

faces, abstract classes, factory methods, and adherence to the Liskov Substitution Principle.

- **Performance**: The system must maintain low memory usage and computational overhead, even when handling CA of a larger scale with complex behaviuor behind them.

- **Usability**: The public API should be intuitive and easy to use. Clear documentation via C# XML comments is provided to ensure developers can get started with minimal configuration, and get support from IntelliSense and similar tools which provide code completion and inline documentation.

3.2 High-Level Architecture

With these requirements in mind, the architecture was designed to prioritize crality, modularity, and extensibility. Because of this, the architecture of the simulation component of the library follows a plugin-based style which is centered around a core engine responsible for the management of the simulation lifecycle.

To use the library, users define their own rule classes, state enums, and state property classes, which integrate into the system at compile time. These custom components must conform to a set of predefined interfaces and abstract base classes, allowing them to plug into the simulation engine seamlessly and ensuring compatibility across different implementations.

Supporting this high-level structure, a number of lower-level design patterns and decisions were applied to reinforce flexibility and maintainability.

3.3 Design Decisions and Patterns

3.3.1 Use of Interfaces and Abstract Classes

Interfaces and abstract base classes serve as contracts for user-defined plug-ins, ensuring that all extensions follow a consistent API. This design decision supports polymorphism and interchangeability.

3.3.2 Factory Method Pattern

In order to create user-defined components at runtime/compile time with flexibility, factory methods are defined in interfaces and configuration classes, centralizing object creation and hiding complex instantiation logic.

3.3.3 Separation of Concerns

The simulation component of the library (and the library as a whole) is separated into layers and modules (the API layer, the core simulation engine, configuration classes) to isolate responsibilities and improve maintainability and testability.

3.3.4 Use of Configuration Classes for Object Creation

The simulation component of the library uses configuration classes responsible for holding the data vital for the creation of component objects (such as cellular automaton, grid, rule, state).

This design detaches the construction logic from the client code, allowing flexible customization of diverse properties without modifying core implementation. It promotes a cleaner and more readable user codebase.

3.4 Component Overview

The simulation side of the library is composed of several core components, each responsible for a distinct part of the simulation lifecycle. Together, these components form the foundation of the plug-in-based architecture. Where appropriate, class diagrams have been provided to illustrate the structure and extensibility points of each component.

3.4.1 State

In the library, the finite set of cellular automata states is represented via a user-defined enumeration available inside the C# programming environment (using the `enum` keyword). Most class components in the library (which depend directly on this finite set), contain a generic parameter `SE`, and unless stated otherwise, a generic parameter `SE` in

these classes will refer to a state enumeration. For a user to be able to create a cellular automaton, they need to define their own enumeration.

EData<SE>

A static class which holds automatically-calculated metadata about an enumeration type. On usage, it validates necessary qualifications for the enumeration type (state set), and provides public methods for quicker, easier, and meticulous interaction with the enumeration type. An UML class diagram for this class can be seen in Figure 3.1.

Public properties:

- `int Size` – returns the amount of values in the enumeration; equivalent to the amount of states in the finite set.
- `SE[] Values` – returns an array of the values.
- `int[] IntegerValues` – returns an array of state values mapped to integer values.

Public methods:

- `int Index(SE value)` – method which returns the index of the value. This is used mostly as syntactic sugar and doesn't affect performance.
- `SE GetRandom()` – method which returns a random state value from the finite state. Keeps randomization of an enumeration value connected to a single `Random` object used for randomization therefore reducing memory overhead slightly.

EArray<SE,T>

A wrapper class for the original `T[]` (array) class in C# which allows the user to access a value `T` in the same manner as they would from an array, but using a state value instead of an integer. This wrapper provides properties, constructors, and an indexer similar to an `Array` class and an UML class diagram for this class can be seen in Figure 3.2. This class provides syntactic sugar for the API.

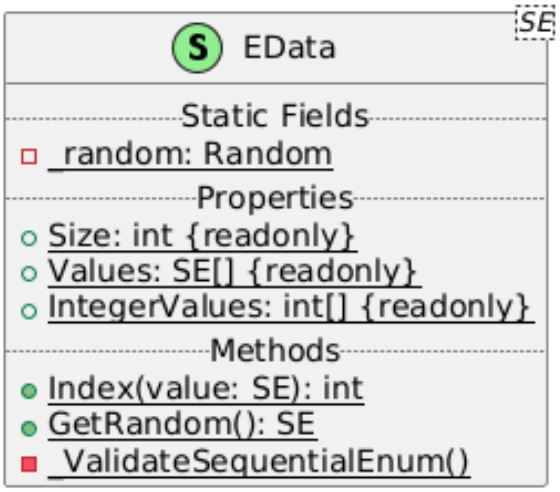

Figure 3.1: EData UML class diagram.

3.4.2 Grid

The cell state values are stored in specialized n-dimensional grid objects designed to allow for quick and efficient access to data.

IGrid<SE>

The `IGrid<SE>` interface defines the abstraction for the core data structure used to store and manipulate the states of a cellular automaton. It generalizes the concept of a multidimensional grid whose element type is a state value. The UML class diagram for this class can be seen in Figure 3.3.

Required public properties:

- `int Rank` – returns the dimensionality of the grid—and, in extent—of the cellular automaton.
- `IReadOnlyList<int> Extents` – returns a readonly list of sizes of each dimension of the grid.

Required public methods:

- `IGrid<SE> Create(GridConfig<SE> config)` – a factory method to create a grid object using a configuration object.

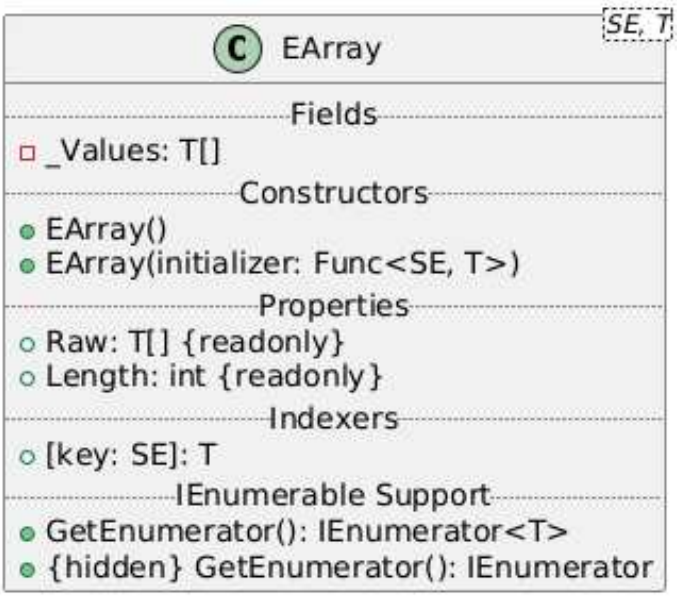

Figure 3.2: EArray UML class diagram.

- `IGrid<SE> Clone()` – method which creates and returns a clone of the grid object it is called on.
- `void Refactor()` – non-mandatory method which allows for the refactoring of the data placement inside the grid object.

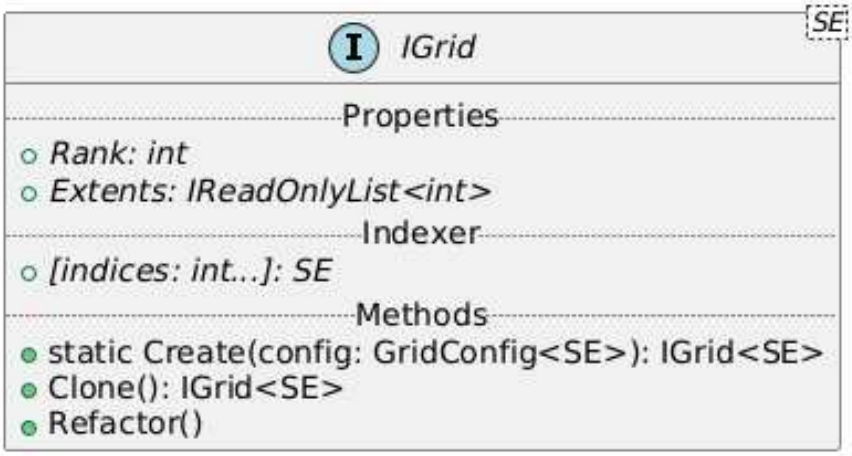

Figure 3.3: IGrid UML class diagram.

The main role of this interface is detaching simulation logic from specific data storage implementations (both custom and library-provided), enabling interchangeable grid back-end implementation.

GridConfig<SE>

The `GridConfig<SE>` class serves as a configuration container for creating grid instances through the `IGrid<SE>.Create()` method. It encapsulates all parameters needed to initialize a grid with specific dimensionality, border behaviour, and default values. The UML class diagram is shown in Figure 3.4.

Public properties:

- `GridBorder Border` – Determines border handling (Wrapped, Fixed, or Mirrored are available values).
- `SE FixedBorderValue` – Conditionally available border state (when `Border == Fixed`).
- `SE Default` – Initial state for all cells.
- `int Rank` – Gets/sets the grid's dimensionality (resizes extents)
- `bool Randomize` – Enables random initialization of the cells in the grid.

Public methods concerning the extents of the grid:

- `SetExtents(params int[] extents)` – Bulk-sets all dimension sizes.
- `GetExtents()` – Retrieves a copy of dimension sizes.
- `GetExtentAt(int index)` – Returns the dimension size of dimension at `index`.
- `SetExtentAt(int index)` – Sets the dimension size of dimension at `index`.

Builder pattern methods:

- `WithExtents(params int[] extents)` – Fluent setter for extents
- `WithRank(int rank)` – Fluent setter for dimensionality

These builder pattern methods serve as syntactic sugar, and can be used like this:

```
var config = new GridConfig<CustomCellState>().WithRank(3)
```

```
.WithExtents(10, 20, 5) { Border = GridBorder.Fixed };
```

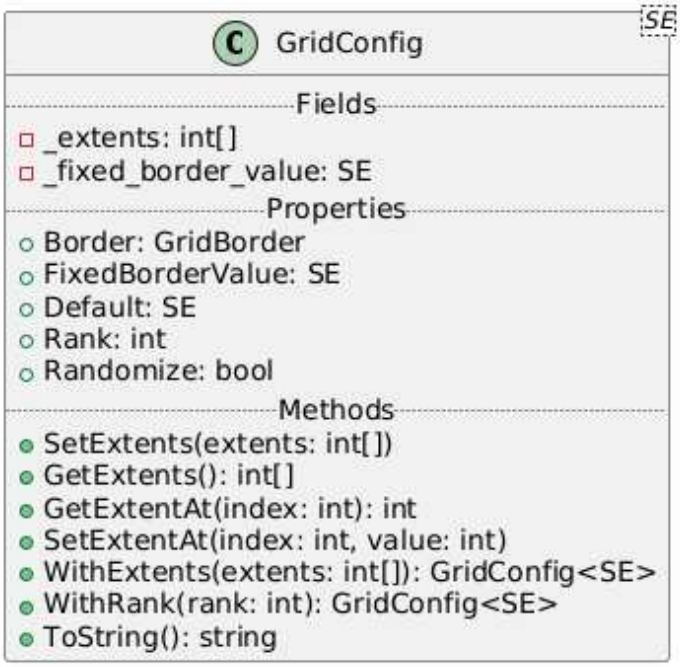

Figure 3.4: GridConfig UML class diagram.

This configuration object allows for type-safe grid initialization while handling edge cases such as:

- Automatic extent adjustment when changing rank;
- Validation of dimension sizes (≥ 1);
- Conditional availability of border values;
- Thread-safe extent array copying.

3.4.3 Grid Implementations

In this library, there are three classes which implement the `IGrid<SE>` interface. Each of these three focuses on a different sparsity level of the grid.

ArrayGrid<SE>

The `ArrayGrid<SE>` class provides a concrete implementation of the `IGrid<SE>` interface using a flat array to store multidimensional grid data. It is best for fully populated

grids due to its dense representation. Internally, it flattens multidimensional indices into a single-dimensional array.

Key characteristics:

- Uses a single contiguous `SE[]` array for storage.
- Converts multidimensional indices to a flat index via a multiplier array.
- Supports fixed, wrapped, and mirrored border behaviour.

Advantages:

- Excellent performance for dense data.
- Low overhead for index resolution due to precomputed multipliers.
- Compatible with any dimensionality and efficient cache locality.

Limitations:

- Memory cost grows exponentially with the number of dimensions.
- Very inefficient for sparse or large grids where many cells share the default value because it still uses space for these default values.

DictionaryGrid<SE>

The `DictionaryGrid<SE>` class provides a sparse implementation of the `IGrid<SE>` interface using a dictionary to store only non-default values. It is ideal for grids with large dimensionality where only a small subset of cells hold non-default data. Internally, it maps multidimensional indices to values via hashed composite keys.

Key characteristics:

- Uses a `Dictionary<IndexKey, SE>` to store explicitly set values.
- Unset cells implicitly return the configured default value.

- Supports fixed, wrapped, and mirrored border behaviour.

Advantages:

- Memory-efficient for sparse or large grids.
- Avoids storing default values, reducing unnecessary allocation.
- Flexible and scalable across arbitrary dimensionalities.

Limitations:

- Slower access time compared to array-based grids due to dictionary lookups. This is ignorable for a dictionary with a smaller amount of stored values.
- Higher overhead for index hashing and equality checks.
- No inherent memory locality, which can affect performance with dense grids.

ChunkGrid<SE>

The `ChunkGrid<SE>` class implements a hybrid sparse grid by partitioning the space into equally sized dense subregions (chunks). Each chunk is allocated only when needed, enabling efficient memory usage for large, sparsely populated multidimensional grids.

Key characteristics:

- Divides the grid into fixed-size chunks, each backed by a dense `SE[]` array.
- Uses dictionary-based chunk management with string keys derived from chunk coordinates.
- Unset cells return the configured default value without allocating chunks.

Advantages:

- Efficient balance between dense and sparse representations.
- Memory allocation is localized to active regions, reducing footprint.

- Fast access within allocated chunks due to dense array storage.

Limitations:

- Slightly higher access overhead compared to flat arrays due to chunk lookup.
- Chunk granularity must be chosen carefully to balance performance and memory use.

3.4.4 Neighborhood

The neighborhood class is used as a readonly view into an `IGrid<SE>` object that supports observing from a local center, and allows the user to create rules for how a cellular automaton should behave.

NeighborhoodConfig<SE>

The `NeighborhoodConfig<SE>` class defines reusable and configurable neighborhood patterns for cellular automata simulations. It encapsulates the neighborhood type, radius, and supports automatic offset generation based on grid dimensionality. The class also includes an internal caching mechanism to avoid redundant computations and ensure thread-safe performance. The UML diagram for this class is shown in Figure 3.5.

Public properties:

- `NeighborhoodType Type` – Specifies the neighborhood type (supported types are Moore and Von Neumann).
- `uint Radius` – Defines the maximum distance from the central cell to include in the neighborhood.

Public methods:

- `IReadOnlyList<int[]> GetPrecomputedOffsets(int rank)` – Returns a list of relative coordinate offsets corresponding to the selected neighborhood type and radius in a grid of given dimensionality. Results are cached using a thread-safe dictionary to avoid recomputation for identical configurations.

- `static NeighborhoodConfig<SE> Moore(uint radius = 1)` – Factory method for creating a configuration for a Moore neighborhood with a specified radius.
- `static NeighborhoodConfig<SE> VonNeumann(uint radius = 1)` – Factory method for creating a configuration for a Von Neumann neighborhood with a specified radius.

Design characteristics:

- Thread-safe offset caching for high-performance simulations.
- Avoids unnecessary recalculations of offset patterns.
- Flexible design supports arbitrary grid dimensionality (rank ≥ 1).
- Decouples neighborhood definition logic from simulation runtime.

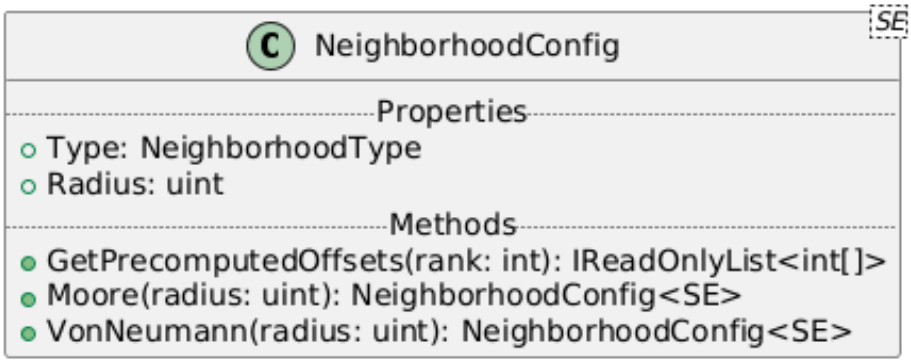

Figure 3.5: NeighborhoodConfig UML class diagram.

This class plays a critical role in parameterizing neighborhood structure for cellular automata, supporting pattern reuse and runtime optimization through offset caching and efficient recursive construction.

Neighborhood<SE>

The `Neighborhood<SE>` class represents a logical view of the cells surrounding a central cell in a grid, as defined by a specific `NeighborhoodConfig<SE>`. This structure enables relative neighborhood-based queries and local computations in multi-dimensional cellular automata. A UML diagram is provided in Figure 3.6.

Public constructor:

- `Neighborhood(int[] center, IGrid<SE> grid, NeighborhoodConfig<SE> config)` – Initializes a new neighbourhood instance centered at a given grid position. Validates that `grid` and `center` are not null, and that `center.Length` matches the grid's rank.

Public indexer:

- `SE this[params int[] offset]` – Provides access to the state of a cell at a specified relative position from the center. Throws exceptions if:
  - The offset dimension does not match grid rank.
  - The offset lies outside the defined neighbourhood bounds.

  The center cell (offset = 0) is included but handled separately.

Public methods:

- `int HasEqual(SE[] values)` – Returns the number of neighbour cells whose states are equal to any of the given values.
- `int HasDifferent(SE[] values)` – Returns the number of neighbour cells whose states differ from all given values.
- `IEnumerable<(int[] Offset, SE State)> EnumerateNeighbors()` – Returns all neighbouring cells' relative offsets and corresponding states.

Design characteristics

- Generic over cell state enum type `SE`.
- Compatible with grids of arbitrary dimensionality.
- Efficient recursive enumeration with filtering.
- Integrates cleanly with `NeighborhoodConfig<SE>` and `IGrid<SE>` interface.

- Includes utility queries for cell comparison (`HasEqual`, `HasDifferent`).

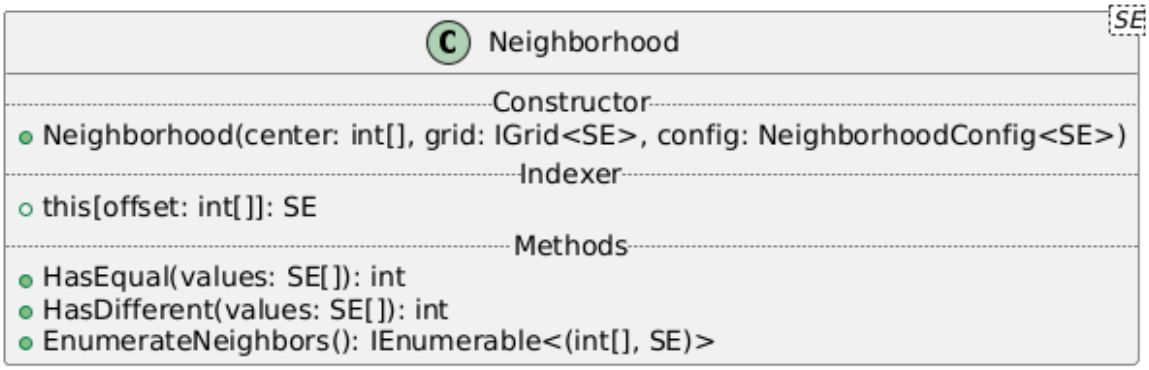

Figure 3.6: Neighborhood UML class diagram.

This class encapsulates all logic for neighborhood-based inspection in cellular automata and is a critical building block for rule evaluation and pattern recognition in local update schemes.

3.4.5 Rule

In order to create a rule which the cellular automaton will follow, the user needs to implement a new class which will extend `IRule<SE,SD>`. A created object of this (user-defined) class should ideally be stateless and reusable.

IRule<SE, SD>

The `IRule<SE, SD>` interface defines the contract for any cellular automaton rule that computes the next state of a cell based on its local `Neighborhood<SE>`. This interface makes it possible to create pluggable and reusable rule logic with different automaton configurations. The UML diagram is shown in Figure 3.7.

The generic parameter of this class `SD` represents a class with a public parameterless constructor which holds and represents per-state data (e.g., if the automaton simulates liquids, and all state values are liquids, SD class can contain a property for temperature and viscosity).

Required static factory method:

- `static abstract IRule<SE, SD> Create()` – Defines a static method that must

be implemented to construct a new instance of the rule. This allows instantiating rules generically without knowing the concrete type.

Required primary method:

- `SE Apply(Neighborhood<SE> nbh, EArray<SE, SD> data, int step)` – Applies the rule logic to the specified neighbourhood. It takes into account the current local cell environment, any additional data, and the automaton step number. Returns the new state to assign to the central cell.

Design characteristics:

- Fully generic and extensible rule interface.
- Supports n-dimensional neighbourhoods via `Neighborhood<SE>`.
- Encourages separation of state-transition logic from grid infrastructure.
- Auxiliary data (`SD`) allows for rule-specific extensions.

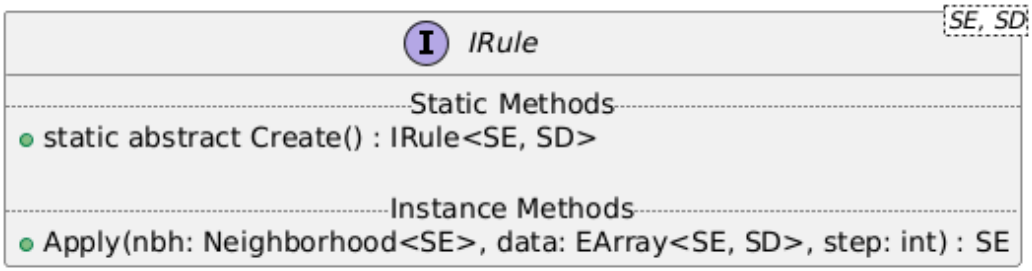

Figure 3.7: IRule interface UML diagram.

This interface serves as the formal abstraction for defining rule logic within the framework. It guarantees that all rule implementations align with the underlying neighbourhood model and can be managed consistently by the simulation engine.

3.4.6 Automaton

The `Automaton` class integrates all key components to provide a flexible cellular automaton engine. It enables users to create automata with minimal complexity.

AutomatonConfig<SE, SD, R, G>

The `AutomatonConfig<SE, SD, R, G>` class encapsulates all necessary configuration parameters to instantiate a cellular automaton. It is a generic class with the following type parameters and constraints:

- `SD` — Data type associated with each cell.
- `R` — A rule type implementing `IRule<SE, SD>`, which defines the cell update logic.
- `G` — A grid type implementing `IGrid<SE>`, responsible for spatial storage and management of cell states.

The constructor optionally accepts a `rank` parameter, defaulting to 1, which sets the dimensionality of the grid.

Primary properties:

- `State`: An instance of `EArray<SE, SD>` holding state data for every state.
- `Neighborhood`: A `NeighborhoodConfig<SE>` instance specifying the neighborhood type and radius, defaulting to a Moore neighborhood of radius 2.
- `Grid`: A `GridConfig<SE>` instance configuring the grid's extents and border behavior.

Automaton<SE, SD, R, G> and Static Factory

The `Automaton<SE, SD, R, G>` class implements the core execution engine of the cellular automaton. Its parameters are the same as `AutomatonConfig<SE, SD, R, G>`.

A static `Automaton` class in the same namespace offers a factory method:

- `Create<SE, SD, R, G>(AutomatonConfig<SE, SD, R, G> config)` — Constructs a new `Automaton` instance from the provided configuration.

This class and method are used as syntactic sugar to avoid constant redefinition of parametrized values.

Internal fields and properties:

- Reference to the underlying grid instance.
- Neighborhood configuration.
- Auxiliary state array.
- A thread-safe queue for managing pending state changes.
- Publicly exposed grid property.
- Current generation counter.

The Step method: The `Step(int amount = 1)` method advances the automaton by the specified number of iterations. Each iteration executes the following steps within a private method `_StepOver`:

- Computes the total number of cells from the grid extents.
- Creates a new instance of the rule `R`.
- Executes a parallel loop over all cell indices, converting linear indices to multi-dimensional coordinates.
- Applies the rule to each cell's neighborhood, producing the next state.
- Enqueues changes where the next state differs from the current.
- Applies all queued state changes atomically to the grid.
- Increments the generation counter.

4 Visualization Library

4.1 Design Goals and Requirements

The visualization add-on for the cellular automata simulation library is designed to support real-time, interactive 3D rendering with a strong focus on user engagement. Its primary goal is to enhance the simulation experience by offering clear, intuitive, and high-performance visual representations of cellular automata states and their evolution over time.

4.1.1 Functional Requirements

The visualization component should meet the following functional requirements:

- Render cellular automata states in a 3D environment with voxel-based representation, supporting color-coded states.
- Provide real-time interactive control of the camera, including rotation, panning, and zooming.
- Enable user interaction to advance simulation steps and observe changes dynamically.
- Support rendering modes based on state alpha values, switching between solid face rendering and wireframe visualization.
- Integrate well with the simulation library as an add-on, interpreting simulation data and exposing an extensible API.

4.1.2 Non-Functional Requirements

The visualization component should meet the following non-functional requirements:

- **Performance**: Leverage GPU-accelerated mesh generation through compute shaders to efficiently render large-scale cellular automata, minimizing CPU overhead and ensuring smooth real-time interaction.
- **Extensibility**: Built as a modular layer on top of the simulation engine, it needs to allow users to tailor interaction models and visualization settings without altering the core system.
- **Usability**: Offer intuitive default controls for navigation and interaction with minimal setup, while also supporting advanced customization options for experienced users.
- **Maintainability**: Adhere to modular design principles and use clearly defined interfaces to separate visualization logic, making the system easier to test, update, and extend over time.
- **Forward Compatibility**: Designed with future enhancements in mind—such as improved transparency and lighting techniques—ensuring the architecture remains adaptable without requiring major refactoring.

4.2 High-Level Architecture

The visualization add-on is implemented as a modular extension layered on top of the core simulation library. It acts both as a visual interpreter of simulation states and as an interactive interface for exploring 3D-rendered cellular automata.

The system is built using OpenTK, which provides OpenGL bindings for graphics rendering and user input handling. To ensure high performance, the visualization uses compute shaders for mesh generation directly on the GPU—minimizing CPU-GPU synchronization overhead and enabling relatively-smooth real-time updates.

Interaction features are designed for flexibility, with a default controller that includes standard camera navigation and simulation stepping via keyboard or UI input. Thanks

to its modular architecture, users can easily override or extend these behaviours to fit their own use cases.

By adhering to well-defined interface contracts, the visualization component maintains a loosely coupled relationship with the simulation engine, supporting independent development and reuse. Rendering options—such as wireframe mode or transparency control—are exposed through configuration objects, allowing for easy customization of visual styles.

4.3 Component Overview

The visualization side of the library is composed of several key components. Each of these is responsible for a distinct aspect of rendering and interaction. Together, these components form the foundation of the plug-in-based architecture that supports real-time 3D visualization. Where applicable, class diagrams are included to illustrate the structure and extensibility points of each component.

4.3.1 State

VisualData

For the simulation to work properly, the user must define a class which defines properties of a state. For the visualization to work properly, this class must extend a class defined in this library called `VisualData`.

This class is intentionally small, containing only two fields:

- `float Light` - a light attribute for a cell state type, implemented as a preparation for future implementations of lighting.
- `Color4 Color` - a color attribute to define a cell state type's color and transparency, implemented in the actual version of the add-on library.

4.3.2 Internal

To simplify the internal structure of the visualization, certain functionalities of the visualization have been encapsulated into classes for ease of use.

Camera

The `Camera` class is a highly extensive class which allows the user to control (move, rotate, animate, etc.) the camera smoothly and with ease. The class itself has many properties, and matrices, each clearly named to reflect its purpose. The class diagram of this class is shown in Figure 4.1.

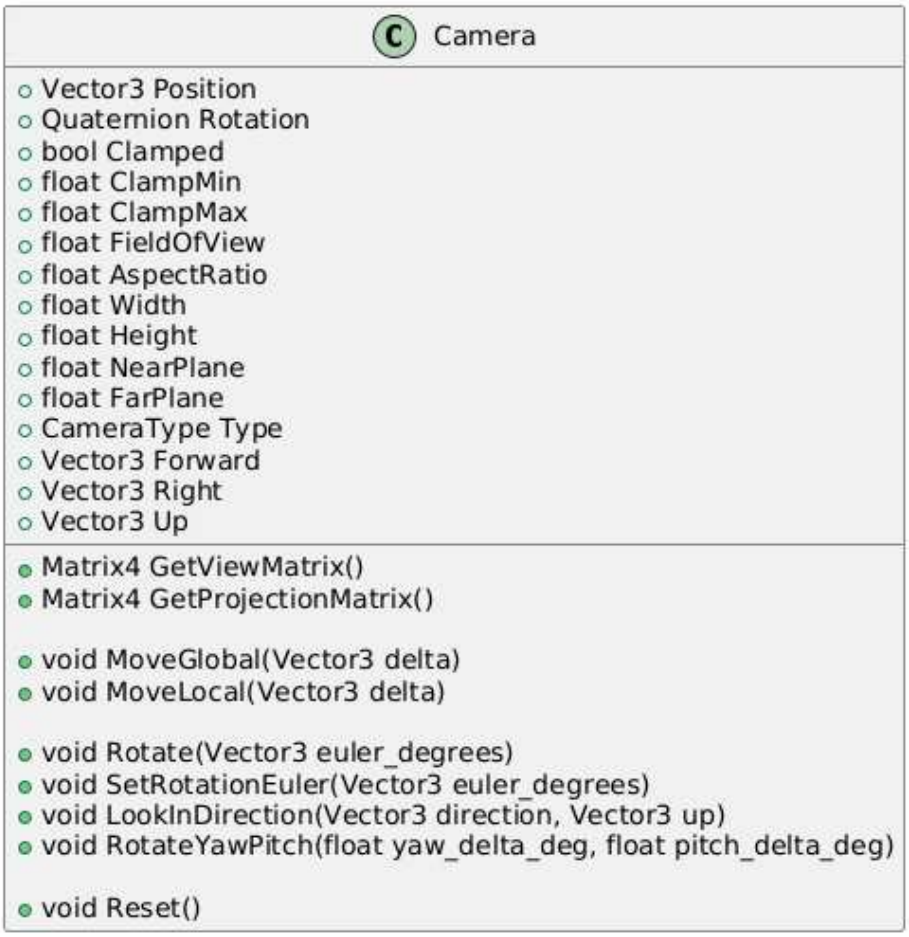

Figure 4.1: Camera UML class diagram.

Window

The `Window` class encapsulates the creation and management of a native OpenTK window used for rendering the cellular automaton. It provides access to key properties such as window size, title, VSync, and fullscreen mode, and it exposes essential lifecycle events including load, render, update, and resize. By abstracting the underlying GameWindow, the class simplifies window initialization and control. A class diagram illustrating its structure is shown in Figure 4.2.

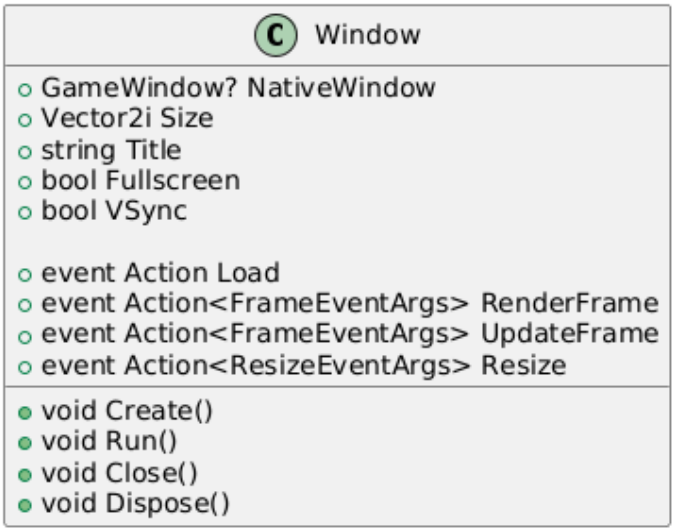

Figure 4.2: Window UML class diagram.

4.3.3 Chunk

Chunk<SE, SP>

The `Chunk<SE, SP>` class is a generic representation of a voxel-based chunk of size: `Size = 32`. This class is used in rendering cellular automata in a 3D environment. It is designed to work with a grid reference which manages voxel state, and a handler reference that manages visualization data.

Public Members

- `static int Size` – Size of the chunk in each dimension (fixed to 32).
- `Vector3i Position` – The world-space position (in chunk units) of this chunk.

Constructor

- `Chunk(Vector3i position, IGrid<SE> grid, ChunkHandler<SE, SP> handler)` – Constructs a new chunk with a position and references to the voxel grid and visual data handler.

Methods

- `void Reset(Vector3i, IGrid<SE>?, ChunkHandler<SE, SP>?)` – Resets the chunk to a new position, with optional new grid and handler.

- `void BuildMesh()` – Generates or regenerates the mesh data using compute shaders.
- `void RenderOpaque()` – Renders the chunk's opaque geometry.
- `void RenderTransparent()` – Renders the chunk's transparent geometry.
- `void MarkDirty()` – Marks the chunk as needing a rebuild.
- `bool UpdateChunk()` – Rebuilds the mesh if the chunk is dirty.
- `void Dispose()` – Frees OpenGL resources allocated by the chunk.

Rendering Pipeline

- The mesh generation uses a compute shader to process voxel data into vertex/index buffers.
- These buffers are stored in Shader Storage Buffer Objects (SSBOs) and later used for indirect rendering of both opaque and transparent geometry.

ChunkHandler<SE, SP>

The `ChunkHandler<SE, SP>` class manages chunk initialization, visibility updates, and rendering in a 3D voxel-based grid. It also works with OpenGL to store and update GPU-compatible visual data.

Public Properties

- `EArray<SE, SP> StateVisualData` – The shared visual state data array for all chunks.
- `int Radius` – The chunk load radius around the camera (in chunk units).
- `int ChunkUpdatesPerFrame` – Maximum number of chunks to update per frame.
- `int StateDataSSBO` – OpenGL Shader Storage Buffer Object storing GPU-formatted visual data.

Constructor

- `ChunkHandler(Camera camera, IGrid<SE> grid, EArray<SE, SP> stateData, int radius = 5, int chunkUpdatesPerFrame = 1)` – Initializes the handler with the camera, voxel grid, and visual state array. Builds chunks within a camera-centered radius and uploads visual data to the GPU.

Public Methods

- `void UpdateChunksIncremental()` – Updates dirty chunks incrementally, capped by `ChunkUpdatesPerFrame`.
- `void MarkAllChunksDirty()` – Marks all active chunks as dirty and queues them for update.
- `void Render()` – Renders all chunks (opaque and transparent passes) relative to the camera view.
- `void InitializeStateDataSSBO()` – Initializes or rebuilds the SSBO with current visual data for use in compute/fragment shaders.
- `void Dispose()` – Frees all GPU and chunk-related resources.

Private/Internal Methods

- `Vector3i GetCameraChunkPosition()` – Returns the current chunk-space position of the camera.
- `void InitializeChunks()` – Loads all chunks within the defined radius of the camera at startup.
- `void RefreshChunkSet(Vector3i newCenter)` – Replaces and reuses chunks based on the camera's movement.
- `bool IsChunkWithinGrid(Vector3i pos)` – Determines if a chunk lies within the voxel grid's bounds.

- `void SetShaderMatrices()` – Uploads the camera's view and projection matrices to the chunk shader.

Rendering Pipeline

- The rendering is split into opaque and transparent passes. Transparent geometry uses blending and wireframe rendering.
- Visual metadata (e.g., color, lighting) is uploaded to the GPU via a Shader Storage Buffer Object (`StateDataSSBO`).
- Chunks use indirect drawing and compute-generated mesh data, referenced by the handler during rendering.

VisualDataGPU Layout

- Struct used to represent each voxel's visual data on the GPU.
- Fields: `Vector4 Color`, `float Light`, `Vector3 Padding`.
- Structured for alignment in SSBOs using `[StructLayout(LayoutKind.Sequential)]`.

4.3.4 Input

IInput<SE, SP, R, G>'

The `IInput<SE, SP, R, G>` interface defines an abstraction for handling user input and frame lifecycle events in a cellular automaton rendering context. Its parameters are the same as `AutomatonConfig<SE, SD, R, G>`.

Public Methods

- `void RenderFrame(FrameEventArgs e, Renderer<SE, SP, R, G> renderer)` – Invoked during each render frame to handle visual updates.
- `void UpdateFrame(FrameEventArgs e, Renderer<SE, SP, R, G> renderer)` – Invoked during each update frame to handle user input, camera control, and simulation steps.

DefaultInput<SE, SP, R, G>

The `DefaultInput<SE, SP, R, G>` class implements `IInput` and handles user interaction, camera movement, and automaton updates based on keyboard and mouse input.

Public Properties

- `float MoveCameraSpeed` – Speed factor for camera movement (default: 5).
- `float RotateCameraSensitivity` – Sensitivity for camera rotation (default: 0.1).
- `int Steps` – Number of simulation steps triggered per key press (default: 1).

Methods

- `void RenderFrame(FrameEventArgs e, Renderer<SE, SP, R, G> renderer)` – Handles camera visibility and input initialization during rendering.
- `void UpdateFrame(FrameEventArgs e, Renderer<SE, SP, R, G> renderer)` – Orchestrates camera movement, rotation, and simulation step logic.

Input Behaviors

- **Camera Movement:** Uses `W, A, S, D, Q, E` keys for directional movement. Holding Shift doubles movement speed.
- **Camera Rotation:** Hold right mouse button and move the mouse to rotate the view.
- **Automaton Step:** Press `F` to trigger a simulation step, processed asynchronously to avoid blocking rendering.

Mouse Interaction

- Pressing the right mouse button grabs the cursor for rotation.
- Releasing it resets the cursor to screen center and restores normal mode.

4.3.5 Renderer<SE, SP, R, G>

The `Renderer<SE, SP, R, G>` class manages the visualization of a 3D cellular automaton. It integrates camera control, chunk management, user input handling, and the rendering loop. Its parameters are the same as `AutomatonConfig<SE, SD, R, G>`.

Fields

- `DefaultInput<SE, SP, R, G>? _interaction` – Optional input handler for user interaction (default: null).
- `int _radius` – The radius (in chunks) around the camera to render (default: 5).
- `ChunkHandler<SE, SP>? _chunk_handler` – Handles chunk management and rendering.

Properties

- `Automaton<SE, SP, R, G> Automaton` – The cellular automaton instance being visualized.
- `Camera Camera` – The camera used to view the simulation.
- `Window Window` – The main rendering window.
- `IInput<SE, SP, R, G> Interaction` – Input handler for user interactions (defaults to `DefaultInput`).

Constructor

- `Renderer(Automaton<SE, SP, R, G> automaton, int radius = 5)` – Initializes the renderer with the automaton and render radius. Throws an exception if the automaton grid is not 3D or if radius is not positive.

Public Methods

- `void ChunksDirty()` – Marks all chunks as dirty to force a rebuild on next update.

- `void Start()` – Creates the rendering window and starts the main render loop, handling OpenGL initialization, event hookup, and continuous rendering and updating.

Rendering Pipeline and Initialization

- `Start()` hooks window lifecycle events such as `Load`, `RenderFrame`, `UpdateFrame`, and `Resize` to manage rendering and input.
- The private method `_InitializeOpenGL()` configures OpenGL state: enables depth testing, back-face culling, blending, sets clear color, line width, and enables debug output.
- `ChunkHandler` is instantiated on load and is responsible for updating and rendering voxel chunks.

Static Factory

- The static `Renderer` class provides a `Create` method for convenient instantiation, enforcing the 3D constraint on the automaton grid.

4.4 Rendering Flow

The rendering flow is decoupled from the components behaviour and can be explained on its own, only referencing the components in which the behaviour takes place.

4.4.1 Renderer

The `Renderer` class handles the entire rendering process of the 3D cellular automaton simulation. It is responsible for initializing OpenGL, managing the rendering window and its events, handling user input, and coordinating the chunk management and drawing pipeline.

- **Initialization:** Upon starting, it creates a window and sets up OpenGL state (depth testing, back-face culling, blending, and debug output).

- **Chunk Management:** The renderer instantiates a `ChunkHandler` which maintains and updates voxel chunks within a configurable radius around the camera.
- **Rendering Loop:** The rendering loop is driven by window events:
  - `Load` event initializes OpenGL and chunk handler.
  - `RenderFrame` clears the screen, updates visible chunks incrementally, renders them, invokes the input handler's rendering method, and swaps buffers.
  - `UpdateFrame` delegates to the input handler to process camera movement, rotation, and automaton stepping.
  - `Resize` updates the camera's aspect ratio.
- **Input Handling:** User input is abstracted via an `IInput` interface, allowing camera control and automaton interaction.

4.4.2 ChunkHandler

The `ChunkHandler` manages a collection of `Chunk` instances representing localized 3D volumes of the automaton grid.

- **Chunk Grid:** Organizes chunks in a spatial data structure for efficient update and rendering based on camera position.
- **Incremental Updates:** Chunks are updated incrementally per frame to spread out the workload and maintain performance.
- **Dirty Flags:** Chunks marked as dirty are rebuilt, which regenerates their mesh data to reflect state changes in the automaton.
- **Rendering Coordination:** The handler issues draw calls for all visible chunks, separating opaque and transparent geometry rendering.

4.4.3 Chunk

The `Chunk` class represents a fixed-size 3D block (typically 323323 voxels) of the automaton's grid.

- **State and Position:** Each chunk tracks its world-space position in chunk units and holds references to the voxel grid and rendering handler.
- **Mesh Generation:** Utilizes a compute shader to process voxel data and generate vertex and index buffers for rendering.
- **Rendering Data:** Stores generated mesh data in GPU buffers (SSBOs) used for indirect rendering.
- **Lifecycle:** Supports resetting to new positions and disposes of GPU resources when no longer needed.
- **Dirty Flag:** Tracks when the chunk needs to rebuild its mesh to represent updated voxel states.

4.4.4 Rendering

The rendering phase consists of multiple stages to draw the 3D automaton:

- **Clearing Buffers:** At each frame, the color and depth buffers are cleared.
- **Chunk Updates:** The `ChunkHandler` updates a subset of chunks incrementally, rebuilding meshes as needed.
- **Drawing Geometry:**
  - Opaque geometry of each chunk is rendered first.
  - Transparent geometry is rendered afterwards to ensure correct blending.
- **Input-Driven Updates:** Camera movements and automaton stepping can trigger chunk updates or visual changes.
- **Double Buffering:** The final rendered frame is presented by swapping buffers.

4.4.5 Compute Shader

The compute shader is responsible for generating the mesh data for each chunk of the automaton in a massively parallel manner on the GPU.

- **Input Buffers:**
  - `VoxelBuffer` holds the voxel types for the chunk, including a padding layer for boundary checks.
  - `TypeBuffer` stores color and lighting properties for each voxel type.
- **Output Buffers:**
  - Separate vertex and index buffers for opaque and transparent geometry are generated to optimize rendering order and blending.
  - Atomic counters track the current count of emitted vertices and indices.
  - Indirect draw command buffers are populated for use with `glDrawElementsIndirect`, enabling GPU-driven rendering.
- **Execution:** The shader dispatches one thread per voxel within a chunk. For each voxel:
  - It reads the voxel type and skips empty or fully transparent voxels.
  - For each of the six faces of the voxel, it checks neighboring voxels to decide whether to emit a face. Faces are emitted only if the neighbor is empty, of a different type, or differs in transparency.
  - Vertices for quads representing faces are generated and indexed accordingly.
- **Mesh Output:** This approach generates a complete mesh representing only visible faces, optimizing performance by culling internal faces.

4.4.6 Vertex and Fragment Shaders

Vertex Shader The vertex shader transforms each vertex position using the model, view, and projection matrices. It also retrieves the color and lighting information for the voxel type from a shader storage buffer and passes these to the fragment shader.

Fragment Shader The fragment shader receives interpolated color and light values from the vertex shader and outputs the final fragment color. It currently applies the

color directly without additional lighting effects, but the lighting value is available for future enhancements.

5 Results

As proof of both concept and implementation, two example projects using both the simulation library and the visualization add-on library have been created.

5.1 Plant Growing Cellular Automaton

5.1.1 Overview

This example demonstrates a 3D cellular automaton simulating plant growth in a voxel grid. The automaton evolves states such as air, seed, branches, and leaves to model the development of a plant-like structure.

5.1.2 Plant States

- **Air**: Empty space, no plant voxel.
- **Seed**: Initial growth point.
- **Branch**: Growing plant structure.
- **StrongBranch**: Mature and reinforced branch.
- **Leaf**: Photosynthetic elements.
- **PseudoLeaf**: Visual leaf state with **Air** behavior.

5.1.3 Automaton Setup

- Grid size: 25 (X) $\times$ 100 (Y) $\times$ 25 (Z)
- Border fixed to **Air** state.

- Neighborhood: Von Neumann, radius 2.
- Initial seed placed at grid position (12, 0, 12).
- Each state assigned a color for rendering (e.g., seed is brownish, leaves are greenish).

5.1.4 Growth Rules

- **Seed** transforms into **Branch** on the first step.
- **Branch** becomes **StrongBranch** if supported by branches above or sufficient adjacent branches (checked periodically).
- **StrongBranch** remains stable.
- **Leaf** disappears if adjacent to a **StrongBranch**, otherwise may revert to **Branch** based on adjacency.
- **PseudoLeaf** and **Air** can convert to **Leaf** if near branches under certain conditions.
- Various time-based and neighbor-count rules control growth patterns.

5.1.5 Visualization Examples

Visualization examples have been provided on Figure 5.1, Figure 5.2, and Figure 5.3.

5.2 Cave Formation Cellular Automaton

5.2.1 Overview

This example simulates cave-like structures using a 3D cellular automaton. The system evolves states representing solid rock and air, modeling natural erosion and filling processes to form caves.

5.2.2 States

- **Air**: Empty space inside the cave.

Figure 5.1: Initial seed voxel at the center of the grid.

- **Solid**: Solid rock or ground.

5.2.3 Automaton Setup

- Grid size: $100 \times 100 \times 100$ (1,000,000 cells total).
- Default state: Air.
- Border state: Fixed to Solid.
- Initial grid randomized with a mixture of solid and air.
- Neighborhood: Moore neighborhood with radius 1.

5.2.4 Erosion and Filling Rules

- For a **Solid** voxel:
  - If fewer than 12 neighboring voxels are solid, it erodes into **Air**.
  - There is a very small random chance (0.1%) it erodes regardless.
  - Otherwise, it remains **Solid**.
- For an **Air** voxel:

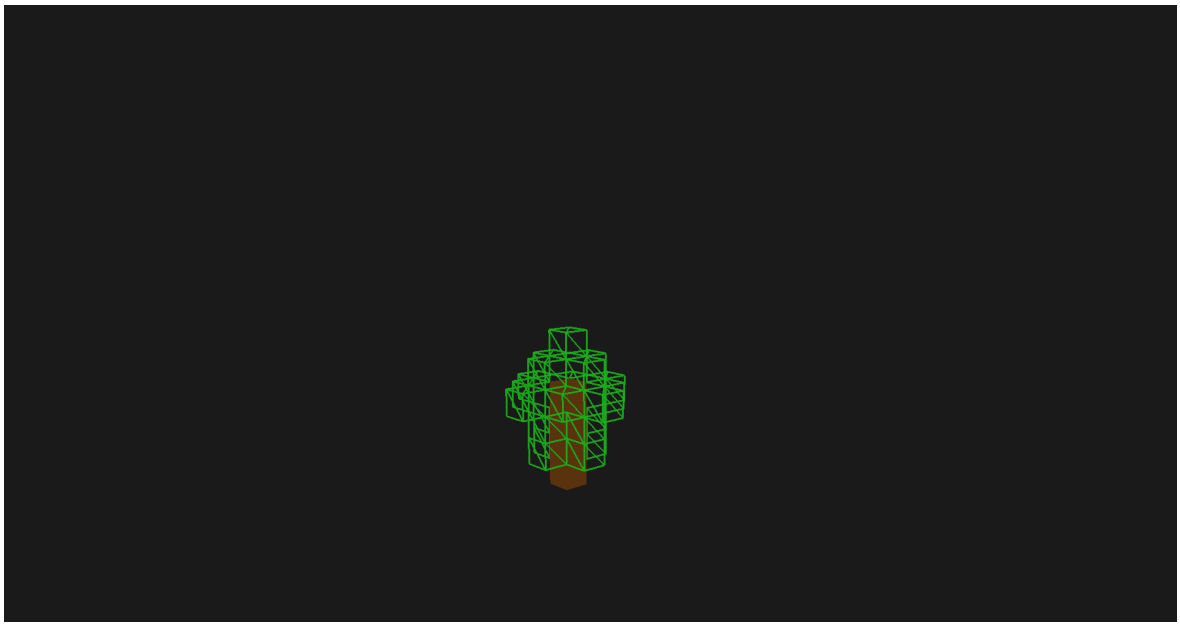

Figure 5.2: Intermediate growth with branches and leaves forming.

- If more than 13 neighbors are **Solid**, it becomes **Solid** (filling in).
- Otherwise, it remains **Air**.

5.2.5 Performance Metrics

This simulation involves a relatively complex 3D cellular automaton with $100 \times 100 \times 100$ cells (1,000,000 in total), processed using neighborhood-based rule evaluation for each voxel in a dense grid.

To evaluate performance, execution times were recorded during the simulation setup and execution steps:

- Automaton object creation: 321 ms.
- Single automaton step (without visualization): 5100 ms.
- Renderer object creation: 2 ms.
- Automaton step with renderer active: 5134 ms.

It is important to note that automaton stepping with the renderer being active takes place in a separate thread, and does not slow down rendering.

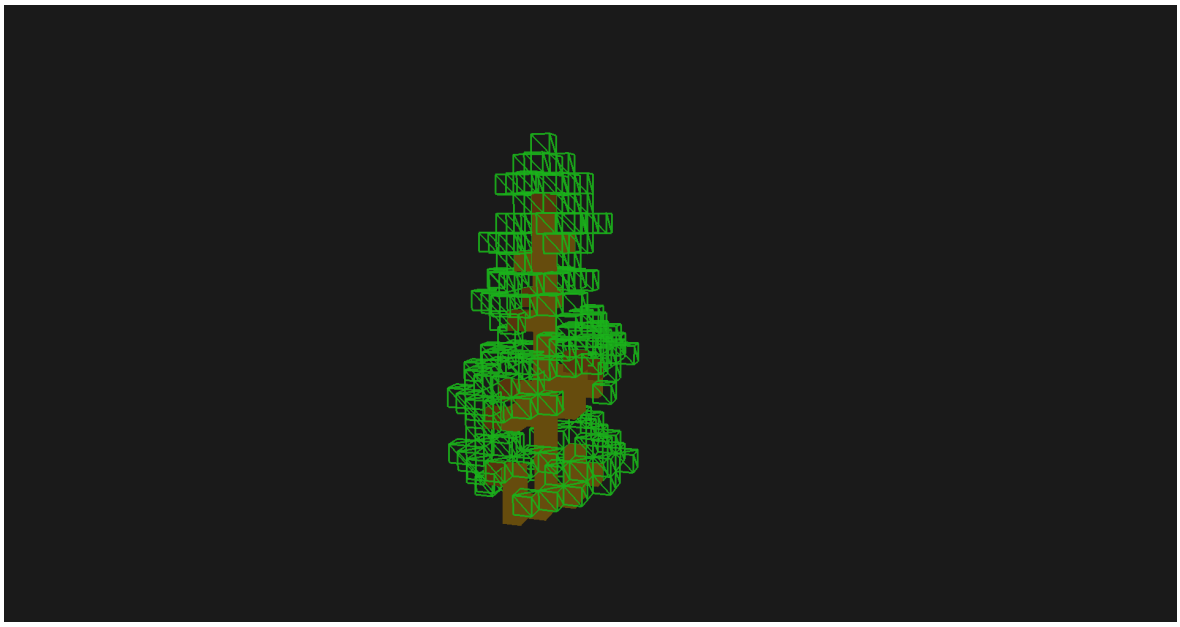

Figure 5.3: Fully grown plant with strong branches and dense foliage.

Performance measurements were collected on a machine with the following configuration:

- CPU: AMD Ryzen 5 5600U with Radeon Graphics
- RAM: 16 GB
- GPU: NVIDIA GeForce RTX 3050
- OS: Windows 11 64-bit

5.2.6 Visualization Examples

Visualization examples of a smaller ($32 \times 32 \times 32$) cellular automaton of the same behaviour have been provided in Figure 5.4, Figure 5.5

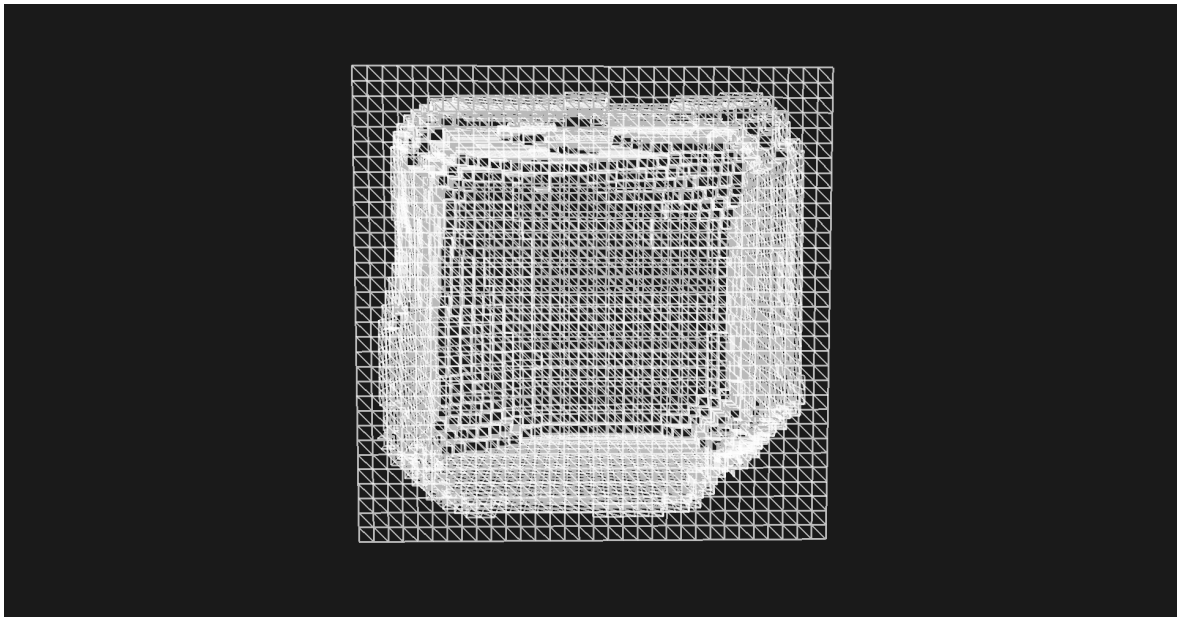

Figure 5.4: Cave formation with low initial solid density and minimal erosion, resulting in a large hollow space.

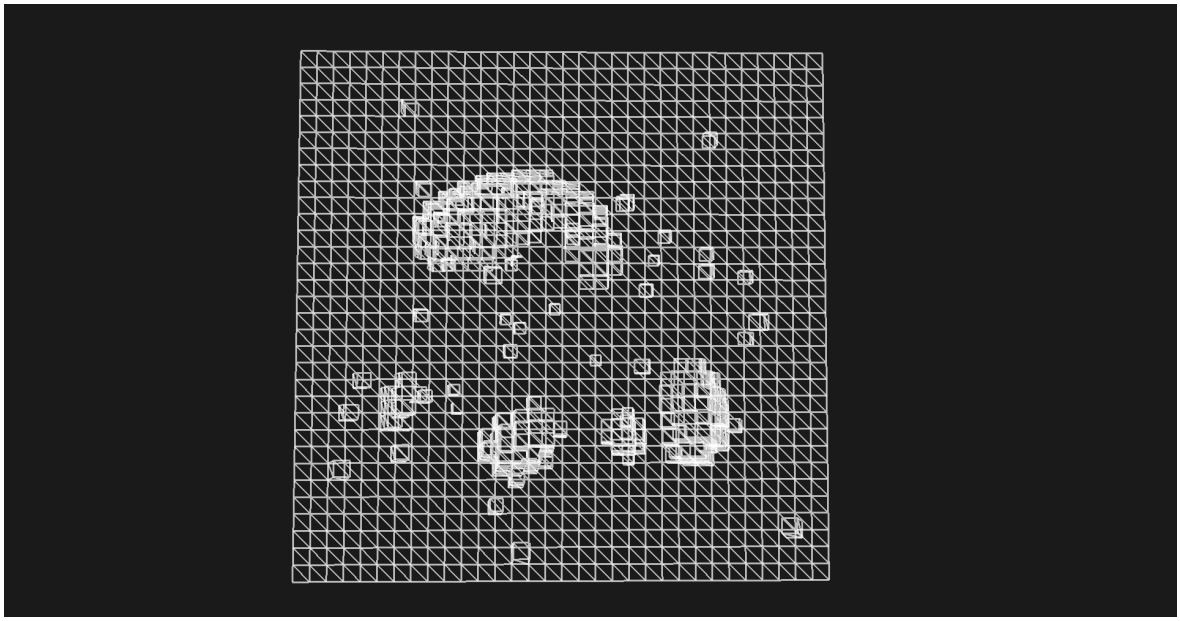

Figure 5.5: Cave structure with very high solid fill and minimal erosion, creating smaller cave openings and more compact formations.

6 Conclusion and Future Work

This thesis documents the creation of a C# cellular automata framework that evolved through iterations. The final architecture emerged from solving problems in simulating 3D automata while maintaining interactive visualization—challenges particularly apparent in the plant growth simulation's branching behaviour.

6.1 Core Achievements

1. **Modular Design** – The library's architecture solves a key extensibility problem: how to support arbitrary rules without compromising simulation performance. The solution combines:

   - Interface-driven state management,
   - Abstract base classes for grid implementations,
   - A factory pattern for automaton composition.

   This approach proved crucial when adapting the system between plant growth and cave formation models.

2. **Performance Tradeoffs** – The three grid implementations (`ArrayGrid`, `DictionaryGrid`, `ChunkGrid`) each excel in different scenarios:

   - `ArrayGrid` for small, dense automata,
   - `DictionaryGrid` for sparse populations,
   - `ChunkGrid` as a balanced hybrid best for a middle ground.

Because of this, parallel processing maintained responsiveness during multi-state simulations.

3. **Visualization System** – The OpenGL renderer's two-phase approach (compute shaders for mesh generation, traditional rendering for display) successfully helped bridge the gap between discrete automata states and smooth 3D visualization. The camera controls enabled detailed inspection of emergent patterns.

6.2 Limitations Observed

- **Rendering**: Current capabilities don't fully capture translucent materials like foliage or liquids due to lack of depth sorting implementation.
- **Rules**: The system implements deterministic rules only, but allows for user to create their own rules which don't stick to deterministic approaches and use stochastic ones; although this is slower.
- **Scale**: Memory constraints affect very large grids, and interactiveness with rebuilding meshes shows visible artifacts when updating chunks on automaton step over.
- **Interaction**: Real-time interaction required mesh recomputation without the ability for caching.

6.3 Future Work

The current implementation provides a solid foundation for several meaningful enhancements:

- **Advanced Lighting** – Implementing physically-based rendering with shadow mapping would address current limitations in material representation. This includes:
  - Dynamic shadows for organic structures (e.g., plant canopies);
  - Normal mapping for surface detail;
  - Ambient occlusion for depth perception;

  - Light propagation through transparent and non-existent voxels.

- **Transparency Handling** – Proper order-independent transparency could be achieved through:
  - Depth sorting implementation inside the compute shader;
  - Depth peeling for accurate foliage rendering;
  - Screen-space refraction for liquid effects;
  - Particle-based alternatives for performance-critical cases.
- **Hybrid Rendering** – Combining the current voxel approach with:
  - Signed distance fields for smooth surfaces;
  - Tessellation for dynamic LOD;
  - Ray marching for advanced effects.

These improvements would specifically address the current rendering limitations while maintaining the system's real-time performance characteristics. The architecture's modular design makes these extensions feasible without fundamental restructuring.

6.4 Final Remarks

This framework demonstrates that cellular automata remain valuable for both research and education. Its architecture intentionally supports several logical extensions while already providing a complete foundation for 3D automata experimentation. All project and test files are available for community use and extension at `https://github.com/vito-vrbic/CADNDS/releases/tag/v1.0`.

References

[1] S. Wolfram, “Cellular automata,” https://plato.stanford.edu/entries/cellular-automata/, 2021, [Online; accessed May 2025].

[2] Cambridge University Press, “Emergence,” https://dictionary.cambridge.org/dictionary/english/emergence, 2025, [Online; accessed May 2025].

[3] M. Gardner, “The game of life,” https://web.stanford.edu/class/sts145/Library/life.pdf, 1970, [Online; accessed June 2025].

[4] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002, extensive discussion of cellular automata applications, including computer graphics and simulation.

[5] T. Karpati, “Compound cellular automata,” https://tatasz.github.io/compound_ca/, 2021, [Online; accessed April 2025].

[6] R. Fong, K. Aberman, K. S. McDonough, and A. M. Saxe, “Self-organizing patterns: How cellular automata can generate complex textures,” https://distill.pub/selforg/2021/textures/, 2021, [Online; accessed April 2025].

[7] B. Zgeb, “Procedural generation with cellular automata,” https://bronsonzgeb.com/index.php/2022/01/30/procedural-generation-with-cellular-automata/, 2022, [Online; accessed May 2025].

[8] J. Cawley, “Algorithmic terrain with cellular automata,” https://demonstrations.wolfram.com/AlgorithmicTerrainWithCellularAutomata/, 2011, [Online; accessed April 25, 2025].

[9] H. W. Franke, “Cellular automata: Models of the physical world,” in *High Performance Computing in Science and Engineering ’12*, W. E. Nagel, D. H. Kröner,

and M. M. Resch, Eds. Springer, 2013, pp. 3–13, [Online; accessed May 2025]. https://doi.org/10.1007/978-3-642-35482-3_1

[10] M. Fridénfalk, “Pattern generation with cellular automata in hexagonal modular spaces,” in *Proceedings of SIGRAD 2016*, 2016, discusses hexagonal CA for procedural content generation in games and design.

[11] M. Saadat and B. Nagy, “Cellular automata approach to mathematical morphology in the triangular grid,” *Acta Polytechnica Hungarica*, vol. 15, no. 6, pp. 46–61, 2018, covers CA in triangular grids used for image processing and dilation/erosion.

[12] M. Fridénfalk and Cr-eatures, “Bucket-hashing implementation of sparse cellular automata in cellang,” Wolfram Group, Technical Report, 2008, describes using hash tables to store cell coordinates in sparse CA.

[13] S. Szkoda, Z. Koza, and M. Tykierko, “Accelerating cellular automata simulations using avx and cuda,” *arXiv preprint arXiv:1208.2428*, 2012, uses AVX/SSE and CUDA to optimize bitwise CA (FHP model).

[14] C. A. Navarro, F. A. Quezada, E. Meneses, H. Ferrada, and N. Hitschfeld, “Cat: Cellular automata on tensor cores,” *arXiv preprint arXiv:2406.17284*, 2024, introduces GPU-based tensor-core CA with high performance on complex neighborhoods.

Abstract

Simulation and visualization of cellular automata in discrete space

Vito Vrbić

This paper explores methods for simulating and visualizing cellular automata in discrete space, with an emphasis on their application in computer graphics. The theoretical foundations of cellular automata are analysed, including the formal definition, lattice structure, and neighbourhood variants. An implementation of a modular library in the C# programming language for automata simulation is presented, along with an extension for 3D visualization using OpenGL. The results are demonstrated through two examples: plant growth and cave formation. The paper concludes with a discussion of the advantages, limitations, and possible directions for further development, providing useful insights into applications in the areas of procedural generation and dynamical systems modelling.

Keywords: Cellular Automata, OpenGL, C#, Library, Visualization, Simulation, Optimization

Sažetak

Simulacija i vizualizacija staničnih automata u diskretnom prostoru

Vito Vrbić

U ovom radu istražuju se metode simulacije i vizualizacije staničnih automata u diskretnom prostoru, s naglaskom na njihovu primjenu u računalnoj grafici. Analiziraju se teorijski temelji staničnih automata, uključujući formalnu definiciju, strukturu rešetke i varijante susjedstva. Prikazana je implementacija modularne biblioteke u programskom jeziku C# za simulaciju automata, zajedno s proširenjem za 3D vizualizaciju korištenjem OpenGL-a. Demonstriraju se rezultati kroz dva primjera: rast biljaka i formiranje špilja. Rad zaključuje raspravom o prednostima, ograničenjima i mogućim smjerovima daljnjeg razvoja, pružajući korisne uvide za primjenu u područjima proceduralne generacije i modeliranja dinamičkih sustava.

Ključne riječi: Stanični Automati, OpenGL, C#, Biblioteka, Vizualizacija, Simulacija, Optimizacija