# A novel data structure for voxel rendering

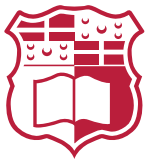
**Keith Farrugia**

Supervisor: Dr Keith Bugeja

Co-Supervisor: Dr Sandro Spina

June 2025

*Submitted in partial fulfilment of the requirements for the degree of Computing Science.*



L-Università ta' Malta
Faculty of Information &
Communication Technology

# Abstract

This project presents a novel data structure implementation designed specifically for voxel-based rendering. The primary aim is to develop a dynamic system capable of managing voxel data efficiently while supporting real-time updates and rendering. By leveraging a tiered hierarchy of sectors, chunks, and compact voxel arrays, the system allows for fast access, minimal memory overhead, and smooth integration with a mesh generation pipeline. Key features include spatial indexing, chunk-level organisation, and dynamic streaming, all geared towards maintaining responsiveness in large or procedurally generated worlds.

To evaluate the effectiveness of the design, a range of configurations and workloads were tested, including variations in voxel storage models, mesh generation strategies, and level-of-detail (LOD) approaches. Benchmarks were gathered to assess memory usage, update times, and mesh complexity across different scenarios. While the implementation does not aim to outperform all alternatives, it prioritises simplicity, extensibility, and practical performance for real-time applications, offering a solid foundation for further development and experimentation.

# Acknowledgements

I would like to sincerely thank my supervisor, Dr Keith Bugeja, and co-supervisor, Dr Sandro Spina, for their continuous guidance, patience, and encouragement throughout the duration of this project. Their insight and feedback were invaluable in shaping both the direction and depth of this work.

I am also deeply grateful to my family for their unwavering support and understanding during the more demanding phases of this journey. Their belief in me has been a constant source of motivation.

Finally, I wish to thank my peers and friends for providing a collaborative and open environment where I could freely ask questions, exchange ideas, and learn from their experiences. Their input helped shape my understanding and approach, and their camaraderie made the process all the more meaningful.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

CCD  Compact Colour Data.

CLD  Compact Location Data.

CND  Compact Normal Data.

LOD  Level of Details.

SVO  Sparse Voxel Oct-Trees.

# 1 Introduction

Voxels offer a powerful solution to a specific yet intricate category of problems. They are widely used in areas such as game development, 3D model rendering, physics simulations, and various forms of spatial data representation. One of the most recognisable examples of voxel usage is in procedural and malleable environments, where the world can be generated or modified in real-time, usually built upon a voxel-based system. Voxels are also commonly adopted as a stylistic choice, giving certain games or scenes a distinct visual identity reminiscent of pixel art in three dimensions.

Despite their strengths, voxels come with a host of technical challenges. A key issue lies in performance: voxel systems tend to scale poorly, consume large amounts of memory, and can be expensive when it comes to mesh generation and real-time updates. These drawbacks make naïve implementations impractical for large or highly dynamic worlds. Various voxel engine implementations attempt to mitigate these issues through a range of optimisation strategies. Each solution tackles a different aspect of the problem, often making trade-offs between speed, memory use, visual fidelity, and implementation complexity.

This area remains an appealing space for experimentation and optimisation. The voxel paradigm presents a rich design space, offering many paths to explore depending on the specific goals of a project, whether it's real-time interaction, high-resolution detail, or efficient storage. Understanding and addressing the core problems of voxel-based systems is essential for creating performant and scalable applications that harness their full potential.

## 1.1 Research Question, Goals & Objectives

*"Can a novel and lightweight data structure be used to create an effective and efficient voxel engine?"*

The primary goal of this project is to determine whether a novel and lightweight data structure can be used to power a voxel engine that performs effectively and efficiently in real-time environments. For the purpose of this project, **lightweight** refers to a structure that reduces memory usage, minimises CPU overhead, and limits GPU load, particularly during mesh generation and rendering. The project focuses on exploring a practical, clear, and adaptable design that can handle the core challenges of voxel rendering. This includes achieving reasonable performance in terms of mesh generation, memory usage, and dynamic updates. The engine should be capable of representing and managing voxel data in a way that supports smooth streaming, face

culling, and optional level of detail (LOD) mechanisms. To achieve this goal, the project will focus on the following key objectives:

- To design a novel voxel data structure that supports fast lookups, updates, and chunk-level organisation.
- To implement a mesh generation system that minimises redundant geometry while maintaining visual consistency.
- To evaluate the performance of the system under various workloads and configurations.
- To identify the trade-offs involved in using a compact structure over more complex alternatives.

## 1.2 Structure of the Report

The remainder of this report is structured as follows:

- **Chapter 1 - Introduction**
  Introduces the motivation behind the project, outlines the problem space, presents the research question, and defines the primary goals and objectives.
- **Chapter 2 - Technical Background**
  Covers the core concepts required to understand the rest of the report, including how voxels work, rendering pipelines, mesh structures, and GPU considerations.
- **Chapter 3 - Literature Review and Related Work**
  Surveys previous research and practical implementations related to voxel rendering, identifying key strengths, limitations, and gaps that motivate the current work.
- **Chapter 4 - System Architecture and Implementation**
  Details the proposed data structure, design choices, and implementation strategy, including mesh generation, chunk organisation, and streaming logic.
- **Chapter 5 - Performance Evaluation and Analysis**
  Presents benchmark results and discusses how the system performs under various configurations, highlighting trade-offs and bottlenecks.
- **Chapter 6 - Conclusion and Future Work**
  Summarises the project's outcomes, reflects on the initial research question, and outlines possible directions for future improvement or extension.

# 2 Background

This chapter presents fundamental concepts and terminology that are essential for understanding the material discussed in the rest of the report. It aims to provide readers, particularly those who may not be familiar with voxel-based systems or 3D rendering, with the necessary context to engage with the literature review, methodology, and technical implementations that follow.

## 2.1 Definition and Role of Voxels

The word voxel has its etymological roots in the two words, 'vox' and 'el', which respectively mean 'volume' and 'element' [1]. In other words, a voxel is a volumetric element. This is similar to how the word pixel originates from the two words 'pix' and 'el', which mean 'picture element'. In fact, the best way to visualise an example of a voxel would be by comparing it to a pixel. If a pixel holds information on a 2D grid, then a voxel can be thought of as its three-dimensional counterpart, a discrete unit of volume that holds data within a 3D space.

## 2.2 Meshes and the Rendering Process

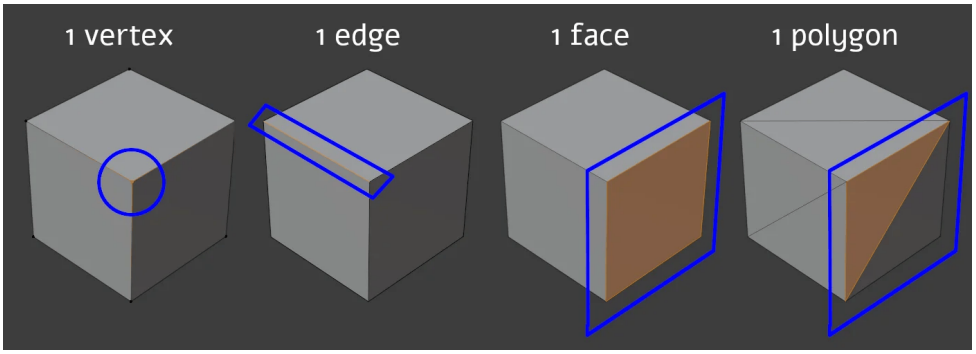



Figure 2.1 Basic mesh elements [2]

Meshes are geometric structures used to define the shapes of 3D objects. At a fundamental level, a mesh consists of a set of vertices, each typically describing a position in three-dimensional space (x, y, z), along with optional attributes such as colour (r, g, b), normals, and texture coordinates (u, v) [3][4]. These vertices are connected to form primitives such as triangles, which together describe the surface of the object.

Rendering a mesh involves processing its vertex data on the GPU through programmable units called shaders. These shaders transform the mesh into fragments

(potential pixels) and apply operations such as lighting and depth testing to ensure the correct appearance of the scene [5] [6]. An overview of the rendering pipeline will be provided in the next section of this report to give readers a conceptual grounding. Interested readers are directed to the official Khronos Group specification [7] and the LearnOpenGL tutorial by de Vries [6] for a more in-depth exploration of graphics APIs and shader programming.

### 2.2.1 The Modern Rendering Pipeline

The graphics rendering pipeline is divided into several stages (Figure 2.2), beginning with mesh creation and ending in the final image being rasterised and displayed on screen. The initial stage runs on the CPU, where the mesh discussed earlier is generated, configured, and preprocessed for the GPU. Preprocessing, a standard preparatory step, includes modifying the scene before rendering, such as determining object visibility and setting the camera's position and perspective parameters [8][6].

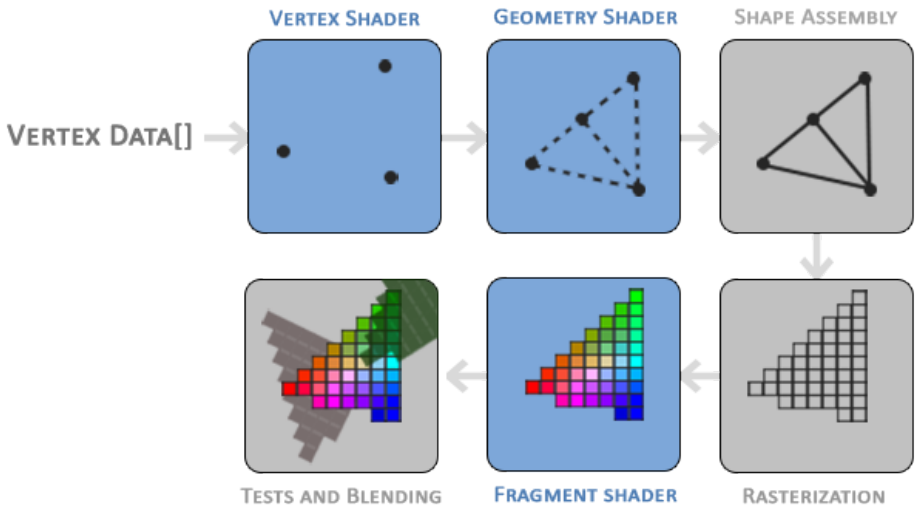



Figure 2.2 Render pipeline [6]

**Geometry Stage** The geometry stage transforms these primitives using matrices. In practice, model, view, and projection matrices transform each vertex from world space into camera and clip space so they appear correctly from the viewer's perspective [8][9]. These per-vertex operations are performed by the vertex shader, the first GPU program in the pipeline, which executes them in parallel [6]. This stage also includes normal transformation, screen-space mapping, and clipping against the view frustum [10]. Two optional shader stages can follow: tessellation control and evaluation shaders subdivide coarse meshes based on level-of-detail criteria [11], and the geometry shader can discard or generate primitives [8].

**Rasterisation Stage** Next, rasterisation takes the transformed and clipped primitives and determines which pixels (fragments) they cover. Hardware interpolation blends vertex attributes across a primitive's surface, producing one fragment per covered pixel [9][6].

**Fragment Shader** The fragment stage, controlled by the fragment (or pixel) shader, computes each fragment's final colour by applying per-pixel operations such as lighting models, texture sampling, and bump mapping [6][8]. Depth and stencil tests are then used to discard occluded fragments: if a fragment lies behind previously rendered geometry, it is discarded; otherwise, it updates the frame buffer [10].

**Output Merger** Finally, the output-merger stage blends the remaining fragments into the frame buffer, handling transparency (alpha blending), multi-sample anti-aliasing, and other compositing operations before presenting the final image on screen [8][9].

### 2.2.2 Draw Calls and GPU Communication

Draw calls describe individual commands or data transfers from the CPU to the GPU. Each call may, for instance, send vertex data or issue a command to execute a shader program [5]. It is commonly held that reducing the number of draw calls per frame can improve frame rate, since each call incurs CPU overhead and may involve changing the GPU's render state [12][13]. Constant data shared across all vertices, known as uniforms, must also be updated via draw calls when their values change, necessitating a balance between update frequency and performance [5]. Moreover, minimising the volume of data transferred or stored on the GPU can further enhance performance, as smaller models are quicker to upload and manage [14]. Techniques such as instancing and mesh batching are often employed to combine multiple objects into fewer draw calls without sacrificing visual fidelity [15].

## 2.3 Summary

The stages of the modern graphics pipeline, from geometric transformations to fragment shading and output, form the foundation of real-time rendering. Understanding these stages is key to grasping the rendering techniques, optimisations, and data management discussed later. This context is essential for evaluating implementations aimed at improving visual fidelity, performance, and adaptability in voxel rendering. The following chapter reviews relevant literature, highlighting key methods, challenges, and advancements.

# 3 Literature Review

This section reviews existing research and practical work related to voxel systems. It provides an overview of the range of approaches and applications explored by various researchers, before narrowing the focus to methods and implementations relevant to the aims of this project.

## 3.1 Polygons vs. Voxels

Most modern rendering pipelines use a polygon-based system, typically composed of triangles or quads, to represent 3D objects [16]. This is evident in the way most modelling software, such as Blender [17], Houdini [18], and SketchUp [19], produces polygonal meshes as the output of model creation. Rendering refers to the process of sampling this 3D scene to generate a 2D image. However, this process can scale poorly if left unoptimised.


(a) A fully modelled 3D tree.

(b) A billboard tree used at a distance.

(c) Detail of the billboard rendering technique.

Figure 3.1 Examples of billboard rendering in *The Legend of Zelda: Breath of the Wild.* [20]

As meshes recede into the distance, densely packed faces can often project to the same screen pixel, leading to aliasing artefacts. Various solutions have been developed to determine how best to resolve the colour of a given pixel under such conditions. One classic method from offline rendering is *adaptive supersampling*, which extends traditional supersampling by dynamically adjusting the sampling rate based on scene complexity. It calculates pixel values using weighted contributions from sampled elements [21, 22].

However, in real-time applications, adaptive supersampling is rarely used due to its computational cost. Instead, modern real-time renderers favour techniques such as temporal anti-aliasing (TAA) [23] or deep learning-based reconstruction methods like DLSS [24], which are more efficient at runtime.

To improve rendering performance at the geometry level, a widely adopted approach is to reduce the Level of Details (LOD) in the mesh as the distance from the camera increases.This form of LOD reduces vertex processing load, eases vertex cache pressure, and simplifies triangle setup in the rasterisation stage, making it especially useful for large scenes. While LOD does not directly reduce pixel shading costs, it can indirectly improve performance when combined with mipmapping to prevent texture aliasing.

This method is effective in both polygon-based and voxel-based systems; however, our current focus remains on the former. Numerous mesh simplification techniques exist, as explored in [25], each offering trade-offs between visual fidelity and computational efficiency. One popular example of geometric LOD is the use of billboard trees, flat images rendered in place of complex geometry when viewed from a distance. A real-world application of this technique can be seen in Fig. 3.1.

Real-time modifications to polygonal models present significant challenges. While algorithms for adaptive mesh simplification, such as those discussed by [26][27], exist, controlling which details are removed during the simplification process is often difficult. Consequently, in many cases, objects of varying complexity are created manually to ensure the desired LOD and that the overall shape is maintained.

Another limitation of polygonal models is their handling of complex terrain. Although it is feasible to generate intricate features like tunnels, caves, and overhangs using polygons, the process can become exceedingly complex [28]. Furthermore, real-time manipulation and editing of such terrain, as well as models and objects in general, are challenging due to the computational demands and structural constraints inherent in polygonal meshes [28].

### 3.1.1 Advantages of Voxel-Based Rendering

With the above explanation in mind, we will now begin discussing why voxels are used, the challenges encountered when using them, and possible solutions that have proven effective.

Voxels serve as a method for data storage, representing models or terrain as volumetric datasets rather than traditional polygonal geometry. In this approach, the visible mesh is generated at runtime based on the underlying voxel data, allowing for dynamic determination of the mesh's shape and LOD. This methodology facilitates real-time modifications to models, as the system can regenerate meshes on the fly to reflect changes. Additionally, voxels are very well suited to terrain generation. This is because their grid-like structure allows for the complex creation of geological features such as tunnelling, overhangs, caverns, arches, and many others [29] [28].

### 3.1.2 Challenges in Voxel Rendering

That said, it is important to note that voxels come with many challenges, both in efficient data structure usage and rendering techniques. Given the segmented nature of voxels, it is computationally expensive if unoptimised to generate cohesive meshes, especially when scaled up to large terrains. The remainder of this literature review will discuss some of the main challenges and the proven methods to minimise complexity.

## 3.2 Survey of Voxel Data Structures

A number of data structures have been developed for voxels, each optimised for different use cases and computational constraints. While some have become more prominent over time, a variety of alternative approaches have also been explored [29]. A selection of commonly referenced voxel data structures is presented below [29].

- **Regular Grid or Look-up Table**: A straightforward 3D-array implementation, allowing for direct access using the voxel's position [30]. This has the downside of being memory-intensive for sparse datasets.
- **Lattice Occupied Voxel Lists (LOVLs)**: Proposed as a memory-efficient structure for storing occupancy data in 3D environments. Unlike traditional Cartesian grids, this method uses a Face-Centred Cubic (FCC) lattice, which offers denser packing of voxels and better spherical symmetry [31].
- **Signed Distance Transform**: A voxel representation where each element stores the signed distance to the nearest surface, positive outside, negative inside. This is particularly useful in shape analysis, collision detection, and physics simulations. Sigg et al. [32] present an efficient GPU-based method for computing these transforms in real time.
- **Sparse Octree**: A hierarchical tree structure that recursively subdivides space into eight octants. Sparse octrees efficiently represent regions with varying levels of detail by only subdividing occupied or detailed areas, thus conserving memory [16], [33], [34].
- **Sparse Block Grid**: This structure divides space into blocks or chunks, storing only non-empty blocks. It is efficient for managing large, sparse voxel spaces by focusing resources on occupied regions [35].
- **Dynamic Tubular Grid**: Specific details on this structure are limited, but it likely refers to a grid that dynamically adjusts around tubular or elongated features within the data, optimising storage and processing for such shapes [36].

- **Volumetric Dynamic Grid**: A grid that can dynamically adapt its resolution or structure based on the complexity or features of the volumetric data, allowing for efficient representation and processing [37].
- **Sparse Paged Grid**: This structure manages voxel data in pages or chunks, loading them into memory as needed. It is particularly useful for handling large datasets that do not fit entirely into memory, facilitating efficient data access and manipulation [38].

Naturally, this is not an exhaustive list; other voxel data structures have been developed for specific domains and requirements. However, those outlined above represent some of the more well-known and practical approaches. The remainder of this thesis will focus particularly on sparse octrees, given their widespread adoption, as well as grid-based approaches, which are most relevant to the implementation described herein.

| Data Structure | Density | Hierarchy | Representation |
|---|---|---|---|
| Regular Grid | Dense | Flat | Explicit |
| Lattice Occupied Voxel Lists (LOVLs) | Dense | Flat | Explicit |
| Signed Distance Transform | Dense | Flat | Implicit |
| Sparse Octree | Sparse | Hierarchical | Explicit |
| Sparse Block Grid | Sparse | Flat | Explicit |
| Dynamic Tubular Grid | Sparse | Flat | Explicit |
| Volumetric Dynamic Grid | Sparse | Hierarchical | Explicit |
| Sparse Paged Grid | Sparse | Flat | Explicit |

Table 3.1 Classification of Common Voxel Data Structures

For clarity, the classifications used in Table 3.1 are as follows: *Dense* structures store voxel data for most or all of the spatial volume, whereas *Sparse* structures store data only where necessary, saving memory in regions of empty space. *Hierarchical* data structures organise voxels in multiple levels of detail or subdivisions, facilitating efficient querying and rendering, while *Flat* structures use a single-level organisation without subdivision. The *Representation* category differentiates between *Explicit* data, where voxel occupancy or values are directly stored, and *Implicit* data, where values are derived from functions or distance fields, such as in signed distance transforms.

### 3.2.1 Sparse Voxel Octrees (SVOs)

An octree is a hierarchical data structure that recursively subdivides three-dimensional space into eight octants, facilitating efficient spatial partitioning [39]. A SVO optimises

this concept by storing only the non-empty regions, thereby reducing memory usage and computational overhead [40].

In the context of ray tracing, SVOs are particularly advantageous due to their ability to represent complex scenes with varying levels of detail. Each node in an SVO can have up to eight child nodes, corresponding to the eight subdivisions of its spatial volume. To manage these child nodes, two 8-bit masks are employed[40]:

- **valid_mask**: Indicates which child slots are occupied by voxels.
- **leaf_mask**: Specifies which of the occupied slots are leaf nodes, representing terminal voxels with no further subdivisions.

A child slot with neither bit set denotes an empty space. If only the valid_mask bit is set, the slot contains a non-leaf voxel with further subdivisions. When both bits are set, the slot represents a leaf voxel [40].

#### Traversal

In an SVO, each voxel is subdivided into eight equal parts, corresponding to the eight octants of 3D space. To determine which child node to traverse based on a given position, the algorithm compares the position's coordinates to the centre of the current voxel along each axis. For each axis (x, y, z), if the position's coordinate is greater than or equal to the centre, a bit value of 1 is assigned; otherwise, it's 0. Combining these three bits yields a 3-bit index ranging from 0 to 7, uniquely identifying one of the eight child nodes. This index is then used to access the corresponding child slot. The valid_mask and leaf_mask bitmasks are subsequently consulted to determine whether the child exists and whether it's a leaf node or requires further traversal[40].

#### Limitations of Sparse Octrees

While SVOs are highly efficient for static scenes, they present challenges when applied to dynamic environments. Updating an SVO to reflect changes in the scene, such as moving objects or alterations in geometry, can be computationally expensive. This is because modifications may necessitate restructuring the tree or rebuilding significant portions of it, which is not conducive to real-time performance [41].

To mitigate these challenges, hybrid approaches have been proposed. For instance, separating static and dynamic elements into different data structures can help manage updates more efficiently. However, this introduces additional complexity in rendering and data management [42].

#### Voxel Data Collapsing

One of the key advantages of SVOs is their ability to compress homogeneous regions. For example, an $8 \times 8 \times 8$ block of identical voxels can be represented as a single node, significantly reducing memory usage [40] (Fig. 3.2). However, in applications where voxel diversity is high, such as in scenes with varied materials or colours, this compression benefit is diminished.

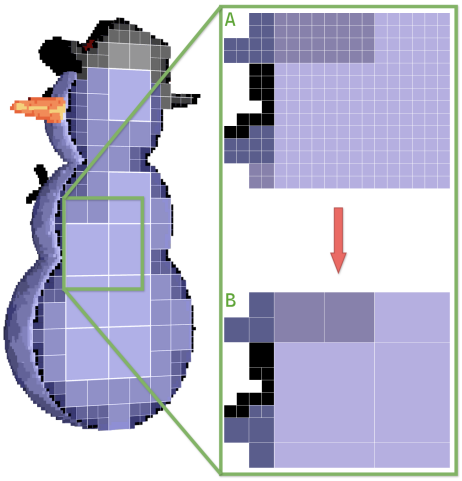



Figure 3.2 SVO collapsing voxel data [43]

Fig. 3.2 illustrates the process of collapsing voxel data using a SVO. ”A” shows a more naïve storage implementation, while ”B” uses an octree to compact the memory. This technique involves recursively subdividing 3D space into octants, allowing for an efficient representation of volumetric data. By collapsing homogeneous regions, SVOs reduce memory usage and enhance rendering performance, making them particularly useful in applications such as real-time 3D graphics and simulations.

### 3.2.2 Spatial Partitioning Techniques for Voxel Data

Given the assumption of a dynamic scene incorporating voxels, partitioning the data into smaller, independently manageable regions is preferable. This approach, widely adopted in real-time applications, allows efficient memory and update management. A well-known example is Minecraft [44], which organises its world into fixed-size chunks that can be loaded, updated, or saved independently.

Regular grids are often the initial choice for many implementations. These have been utilised in certain projects, although more commonly in earlier representations, such as the one described by Wang and Kaufman [30]. Storing all the voxels in a large 3D array offers the fastest possible access speeds (as voxel positions are directly indexable $O(1)$), but it is highly inefficient in terms of memory usage, especially for sparse scenes [45].

Static grids are typically employed in fixed environments or simulations where the scene does not change. These prioritise memory efficiency and are often implemented using sparse data structures like SVOs, as discussed by Niu et al. [45]. Such structures reduce memory usage by storing only non-empty nodes, making them ideal for static or mostly static content, though not particularly suited to rapidly changing environments, as discussed in Section 3.2.1.

In contrast, dynamic grids are more appropriate for interactive or evolving scenes. These often divide voxel data into structured chunks or regions that can be processed independently. For example, Minecraft [44] uses $16 \times 16 \times 256$ block chunks for efficient world management, enabling scalable and interactive terrain rendering.

GigaVoxels, developed by Crassin [16], also follows a similar partitioning strategy but with added complexity. It uses a SVO where each internal node can reference a small block of voxel data, typically $8 \times 8 \times 8$ or $16 \times 16 \times 16$ "bricks". These bricks are GPU-resident and streamed in on demand using a brick pool and indirection buffers, enabling real-time traversal and efficient memory reuse. The system maintains a hierarchical mipmapping scheme and leverages a brick cache to manage active voxel data dynamically during rendering. However, GigaVoxels assumes largely static scenes and can incur substantial memory usage or runtime overhead in highly dynamic environments. Its reliance on CUDA-based architecture also complicates integration with standard rendering pipelines and reduces portability [16], [42], [46].

Another example is the Dynamic Tubular Grid by Nielsen and Museth [36], which similarly breaks voxel data into cells for simulation purposes. This method focuses on high-resolution level set calculations, making it ideal for simulations involving fluid or surface dynamics. Although not designed for visual rendering, its partitioning logic is comparable.

Although these methods vary in complexity and use case, the core concept of breaking the scene down into smaller chunks remains consistent. This concept will be adopted for the actual implementation discussed later on.

## 3.3 Meshing Algorithms

With the organisation of data covered, the next step is to decide how to render the voxels and the shapes they represent. There are two primary methods that will be examined: the Marching Cubes algorithm and the cubic (blocky) representation.

### 3.3.1 Marching Cubes

Voxel-based environments are well-suited for malleable terrain and real-time modifications. However, representing these environments with smooth surfaces, rather than sharp-edged cubes, presents a challenge. Games such as [47] demonstrate how voxel systems can be combined with smoother rendering styles.

One popular approach for achieving such smoothness is the *Marching Cubes* algorithm. Introduced by Lorensen and Cline in 1987 [48], Marching Cubes is a high-resolution 3D surface construction algorithm that generates triangle meshes from volumetric data. It works by evaluating scalar values at the corners of each voxel and

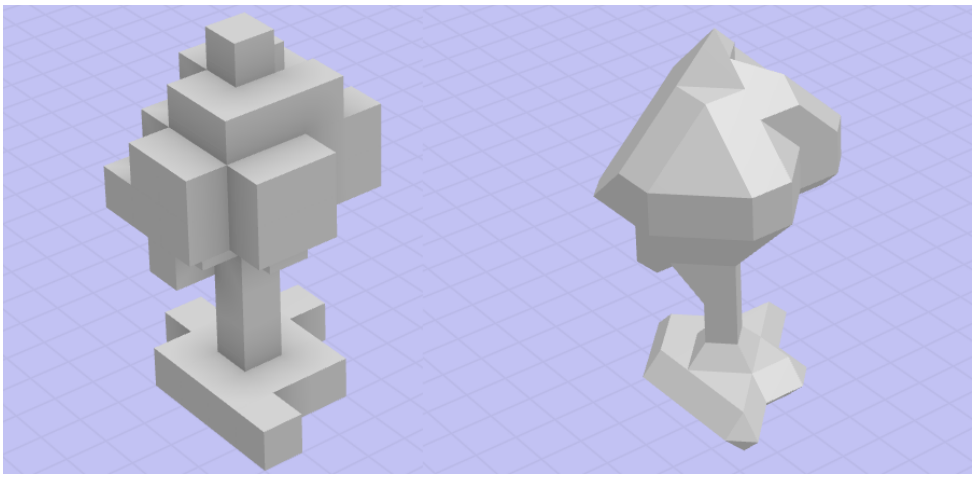


Figure 3.3 Cube Rendering Vs Marching Cubes [52]

determining how the surface intersects the voxel cube, based on a lookup table of 256 configurations.

Despite its simplicity, the algorithm can produce detailed and smooth surfaces that are ideal for rendering terrain or organic shapes. Various optimisations have been proposed over the years to address ambiguities and improve performance. For example, Dyken et al. introduced a GPU-accelerated version using HistoPyramids [49], significantly increasing rendering speed. Other approaches focus on real-time implementations using vertex shaders [50], and improvements in topology resolution, such as edge-growth variants [51].

### 3.3.2 Cubic Voxel Rendering Techniques

Cubic rendering is the most iconic and recognisable approach to displaying voxel environments, popularised by games such as [44] and [53], both of which embrace the distinctive cube-based aesthetic.

One straightforward technique for rendering voxels is *instanced rendering*. Since each cube shares the same geometry, a single cube mesh can be drawn multiple times, with positions and attributes provided via instance buffers. This enables substantial performance gains through batching, allowing vast numbers of voxels to be rendered in a single draw call [54]. However, instanced rendering has notable drawbacks: each cube is rendered in its entirety, so faces hidden between adjacent cubes are still drawn, leading to significant overdraw and increased fragment-shader workload [55].

A more common solution is to cull internal faces [56]. Although this approach demands additional preprocessing on the CPU, it is generally considered the standard method. Each cube's six faces are tested against neighbouring voxels, and any face adjacent to a solid voxel is omitted during mesh generation [57]. By excluding these internal faces, the pipeline processes far fewer polygons, reducing overdraw and

improving performance [56] [55]. Various culling strategies exist, such as column-based culling [58], and studies comparing rasterisation to ray casting demonstrate that pruning internal faces can lower frame times by over 30% in complex scenes [59].

A particularly effective optimisation of face culling is *greedy meshing* [56]. Instead of rendering each face individually, this algorithm merges adjacent, coplanar faces into larger quads wherever possible [60]. This dramatically cuts the vertex count and reduces draw-call overhead, especially in flat or uniform regions such as terrain. Greedy meshing underpins many high-performance voxel engines and is widely adopted in both academic and commercial projects, though it falls outside the scope of this work.

## 3.4 Design Rationale and Methodological Considerations

The literature presents various approaches to 3D scene representation and rendering, each with trade-offs in performance, memory efficiency, and complexity. Traditional mesh pipelines excel at smooth surfaces but struggle with dynamic or destructible environments, where voxel systems offer better modularity and spatial coherence [3], [6]. Uniform voxel grids are simple but require large memory, leading to hierarchical structures like octrees or sparse voxel DAGs [7]. Such structures have traversal costs; for example, octree traversal runs in $\mathcal{O}(\log n)$ time but incurs overhead from pointer chasing and node indirection.

Shader-based pipelines provide flexibility, yet real-time applications need spatially aware, GPU-friendly data structures. Many systems favour visual fidelity over responsiveness, leaving room for lightweight, specialised solutions.

This project adopts a hybrid structure that balances performance and simplicity. Spatial partitioning is implemented using either an AVL tree, which offers $\mathcal{O}(\log n)$ operations, or an unordered map, which provides average-case $\mathcal{O}(1)$ complexity, enabling fast lookups and dynamic insertion with support for world streaming. Theoretically, the AVL tree performs better with smaller datasets due to its balanced nature, while the unordered map tends to be more stable and efficient with larger datasets. Additionally, both octrees and binary space partitioning trees typically exhibit search complexities on the order of $\mathcal{O}(\log n)$, so the system should theoretically perform similarly to other spatial partitioning methods. Within chunks, voxel data is stored in a contiguous flat array, optimising cache locality and iteration without padding. This reduces fragmentation and maintains real-time responsiveness.

The next chapter expands on these foundations and explains key architectural decisions.

# 4 Methodology

This chapter discusses the process behind developing a novel data structure for voxel rendering. It outlines the choices made during the design and implementation phases, explaining why certain approaches were favoured over others. Additionally, it covers the rationale behind specific calculations used throughout the system.[1]

## 4.1 Internal Data Structure

Voxel storage presents specific challenges in balancing dynamic updates, memory efficiency, and access speed. There are three main aims in the context of this project. The data structure must be dynamic so that sets of voxels can be added and removed in real time. It should be efficient to some degree, avoiding the storage of too many unnecessary voxels. And, it must allow for quick access to the voxels themselves.

The most naïve implementation would be to simply have a dynamic 3D array (using `std::vector`, for example) that represents the world of voxels. The array would match the size of the player's view, the visible world, and always be centred around the player, constantly storing the maximum amount. This approach would have the fastest access time of any structure (as it is $\mathcal{O}(1)$ with minimal calculation overhead), but is not efficient at all, as there is no control for large empty spaces.

The resulting concept is to split the data into groups. This is a similar implementation to the Sparse Block Grid and Sparse Paged Grid, as well as in Section 3.2.2. We store a tiered hierarchy of groups, where the atomic unit is the voxel itself. A group of voxels is called a *chunk*; chunks store a set of $16 \times 64 \times 16$ voxels. A group of chunks is then referred to as a *sector*. A single sector can theoretically address up to $32 \times 64 \times 32$ chunks. The world itself is composed of these sectors and supports an addressable space of $32766 \times 32766$ sectors, though in practice only a small subset is allocated dynamically.

Chunks are the most important, as they control and constrain certain features which will be discussed later. The chosen chunk size strikes a balance between containing enough cubes to avoid frequent transitions between chunks, and not containing so many that the program is forced to load and unload large sections at once, thereby losing fine-grained control over memory and voxel loading.

[1] Code repository: `https://github.com/KeithFarrugia/Coperium-Voxels.git`

### 4.1.1 Location Indexing

Voxels, chunks, and sectors are all accessed through compacted location data. This means that a location, typically represented as $(x, y, z)$, is encoded into a single 32- or 16-bit number using bit manipulation. This approach offers two main advantages: it reduces memory usage and simplifies indexing, as the location can be treated as a standard integer once the bits are set. Locations are also defined relative to their containing group. For example, a chunk at position $(0, 0, 0)$ refers to its position within its sector, rather than in world space. The table below outlines the bit ranges used and the value ranges for each structure:

| Struct | X Range | Y Range | Z Range |
|---|---|---|---|
| `voxel_loc_t` | $0$ to $15$ | $0$ to $63$ | $0$ to $15$ |
| `chunk_loc_t` | $0$ to $31$ | $-31$ to $+31$ | $0$ to $31$ |
| `sector_loc_t` | $-32767$ to $+32767$ | – | $-32767$ to $+32767$ |

Table 4.1 Value ranges for coordinate components in compact location structs

Each location encoding is structured as follows:

- `voxel_loc_t` **(16 bits)**:

$$\underbrace{0\ldots0}_{\text{unused (2 bits)}}\ \underbrace{0000}_{\substack{\text{X}\\(4\,\text{bits})}}\ \underbrace{000000}_{\substack{\text{Y}\\(6\,\text{bits})}}\ \underbrace{0000}_{\substack{\text{Z}\\(4\,\text{bits})}}$$

- `chunk_loc_t` **(16 bits)**:

$$\underbrace{00000}_{\substack{\text{X}\\(5\,\text{bits})}}\ \underbrace{0}_{\substack{\text{Y sign}\\(1\,\text{bit})}}\ \underbrace{00000}_{\substack{\text{Y abs}\\(5\,\text{bits})}}\ \underbrace{00000}_{\substack{\text{Z}\\(5\,\text{bits})}}$$

- `sector_loc_t` **(32 bits)**:

$$\underbrace{0}_{\substack{\text{X sign}\\(1\,\text{bit})}}\ \underbrace{000000000000000}_{\substack{\text{X abs}\\(15\,\text{bits})}}\ \underbrace{0}_{\substack{\text{Z sign}\\(1\,\text{bit})}}\ \underbrace{000000000000000}_{\substack{\text{Z abs}\\(15\,\text{bits})}}$$

### 4.1.2 Hierarchical Organisation of Voxel Spaces

The next essential consideration in the design is how to organise the higher-level grouping of voxels, chunks, and sectors. This structure determines how efficiently voxel data can be accessed, updated, and rendered. For this project, two different data structures have been implemented and evaluated in order to determine which is better suited for use in real-time voxel rendering scenarios.

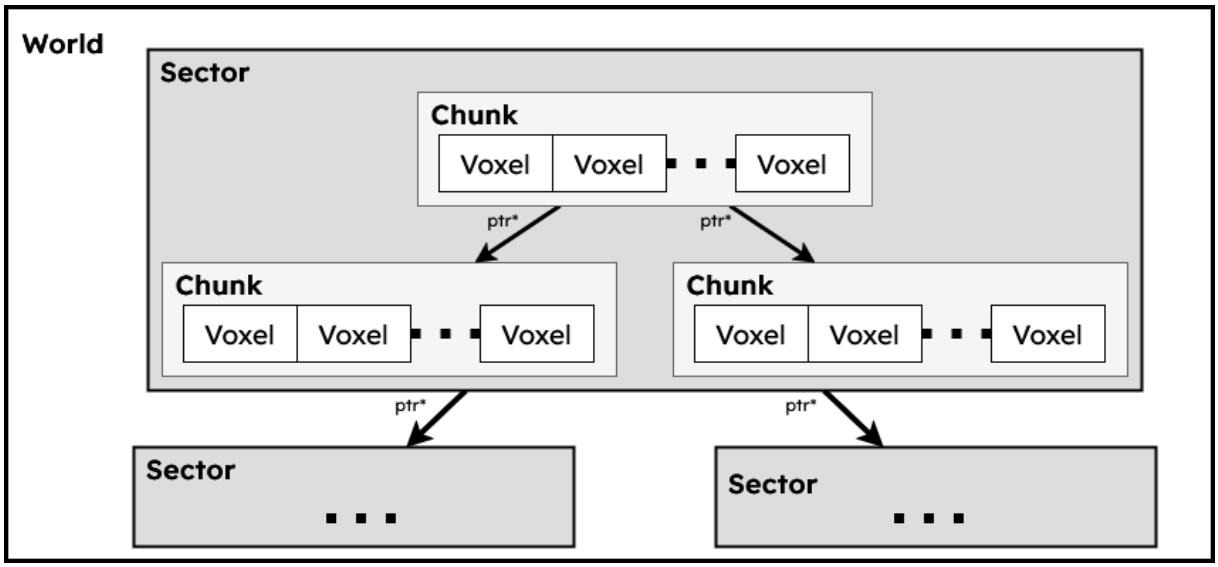



Figure 4.1 Overview of how voxels, chunks, and sectors are organised using an AVL tree

### Chunks & Sectors

The chunks themselves store voxels in a flat array (which will be discussed later), so the focus here is on the organisation of chunks and sectors. The first data structure implemented is a custom indexable AVL tree, which is similar to a standard AVL tree [61], but stores tuples in the form `<index, data>` and includes performance-oriented optimisations such as avoiding recursion. The AVL tree was chosen over alternatives such as Red-Black Trees [62] because it maintains a more tightly balanced binary structure. In this voxel-based context, indexing and traversal operations occur far more frequently than insertions or deletions, making AVL trees a more suitable choice. SVOs were not considered due to their added complexity when it comes to deletion and rebalancing, as discussed in Section 3.2.1.

The second structure used is `std::unordered_map`, which acts as a general-purpose baseline for comparison. While both implementations exhibited similar performance characteristics, the standard library's unordered map was observed to be significantly slower during debugging.

A comparison between the two data structures is provided in Section 5. There, a concise overview is given of which combination or set of data structures appeared to perform most optimally.

### Voxels

Initially, the voxels were also stored within the indexed AVL tree or the map-based data structures. While this approach reduced memory usage, it significantly increased the time required to access voxels. During mesh generation, each voxel must access six of

its neighbours, and the overhead introduced by these structures quickly accumulates. For this reason, a simple flat array is used instead. Although this incurs a larger memory footprint, it provides significantly faster access times, which is advantageous when traversing large voxel spaces.

The flat array is implemented using `std::vector`. This benefits from two factors: indexing can be performed directly using a single integer representing the voxel's position, and the memory is stored contiguously, which improves caching efficiency.

## 4.2 Mesh Generation

Mesh generation is where most of the optimisation techniques were developed and implemented. As previously mentioned in Section 3.3.2, rendering a cube mesh for every single voxel and issuing a draw call per voxel is an extremely inefficient approach. This introduces so much overhead that it becomes unusable in practice. Instead, meshes are grouped into larger sets, in this case, per chunk. Each chunk therefore contains a single mesh representing all the cubes within it. This is beneficial for two reasons: firstly, it allows individual chunks to update their meshes without affecting the rest of the world; and secondly, the mesh is compacted in such a way that it only represents its associated chunk, as will be seen in the later section (Section 4.2.6).

Rendering non-visible faces is also highly inefficient, as it results in an excessively large mesh that places unnecessary strain on the GPU. To address this, the mesh generator traverses the voxels within a chunk and determines, for each voxel, which of its six faces (if any) need to be included in the mesh. A flag register is used to indicate which faces should be added during this traversal. However, this introduces complications when a voxel is situated on a chunk boundary and requires access to voxels in neighbouring chunks or sectors.

### 4.2.1 Straightforward Implementation

The most straightforward approach one might initially consider is to query the world for each neighbouring voxel during mesh generation, as shown in Algorithm 1. While functional, this leads to significant overhead: for every neighbour check, the program must traverse the entire sector $\rightarrow$ chunk $\rightarrow$ voxel hierarchy to locate the target voxel.

Moreover, for each lookup, the code must verify whether the relevant sector, chunk, and voxel even exist. These conditional checks, necessary to avoid null access or segmentation faults, further compound the problem. On modern processors, conditional jumps introduce a performance hit due to instruction pipelining and branch misprediction. As this process runs six times for every single solid voxel (once per face),

and with millions of voxels processed during mesh generation, minimising unnecessary traversal and conditional logic is crucial for streamlining performance.

**Algorithm 1** Straightforward Chunk Mesh Generation

```
Input: World w, Chunk c, Sector s
for each voxel v in c do
    if v is not solid then
        continue
    end if
    Compute world-space position of v
    flags ← 0
    for each of the 6 face directions do
        neighbour_pos ← v's position + face offset
        neighbour ← World.get_voxel(neighbour_pos)
        if neighbour is not solid then
            Set corresponding face bit in flags
        end if
    end for
    Add cube to mesh using flags and colour
end for
Finalise and upload mesh in chunk
```

### 4.2.2 Optimised Face Culling and Neighbour Evaluation

Instead of querying the hierarchy for each neighbour, we can implement a more efficient method. At the start of the meshing process, we query all six neighbouring chunks once and store pointers to them. This means that any subsequent neighbour checks no longer require traversal through the world hierarchy; we can simply check whether a neighbour lies within the same chunk or a neighbouring one. If a neighbouring chunk does not exist, we substitute a generic placeholder chunk, containing only air blocks, for instance. This approach significantly reduces both the number of queries and the number of validation checks during mesh generation.

The idea behind this implementation is to classify voxels based on which neighbouring chunks they need to access. Several examples can be drawn from Fig. 4.2. Voxels that are not on the outer layer of the chunk (highlighted in red) only need to reference voxels within the same chunk; these are grouped into a single case. Voxels on the front face of the chunk (dark yellow) require access to one neighbour, the chunk directly in front, while all other accesses remain within the same chunk.Voxels in the light green area need to access both the front and right neighbours. The corner

case (pink) involves three neighbour-chunk accesses.
In total, there are 26 distinct cases that can be categorised based on the required neighbour-access pattern. Rather than checking which case each voxel belongs to during runtime, we simply handle each case in bulk, sequentially for each chunk. This version should result in a significant speed-up of the system overall.

That said, the code itself becomes harder to maintain and to rework in order to support additional features, such as LODs, because the direct voxel-access assumptions embedded in each bulk case must be revisited and recalculated for each level of detail.

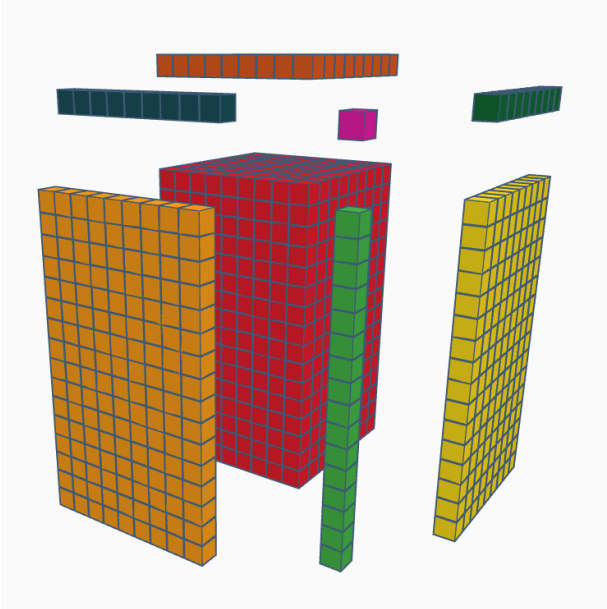


Figure 4.2 An exploded chunk example.

### 4.2.3 Compacted Solution

The final implementation, and the one used in this project, is a compacted version of the previous approach. As before, we begin by querying the six neighbouring chunks at the start of the meshing process. However, this time we store them in a different structure. We use a two-dimensional array (see Table 4.2) to store the neighbours in a way that allows for fast access via bitwise operations.

| | |
|---|---|
| LEFT_NEIGH | CENTER |
| RIGHT_NEIGH | CENTER |
| UP_NEIGH | CENTER |
| DOWN_NEIGH | CENTER |
| FRONT_NEIGH | CENTER |
| BACK_NEIGH | CENTER |

Table 4.2 Neighbouring chunk lookup layout

Since it is structured as a two-dimensional array, we can determine whether to access the current or a neighbouring chunk using a simple Boolean boundary check. For instance, if we want to access the right neighbour for a voxel at position $x = 15$, we evaluate the expression `x < 15`, cast the Boolean result to an integer (where `false` becomes 0), and use it as an index. This yields the following lookup:

```
neighbour_table[RIGHT][int(x < 15)]
```

In this way, we avoid conditional jumps entirely and instead rely on fast, predictable indexing. This technique ensures reliable and rapid access to neighbouring

cubes. Furthermore, it eliminates the need to separate the logic into 26 distinct cases as in the previous approach. While this method adds a few extra instructions, it significantly reduces code duplication and simplifies the traversal logic.

### 4.2.4 LOD Mesh Generation

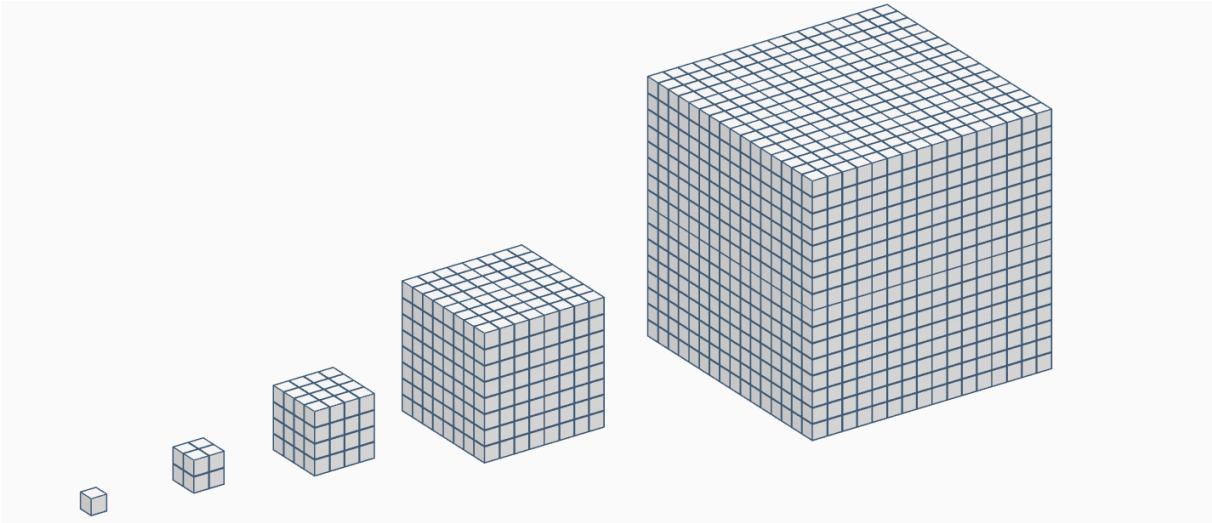


Figure 4.3 A single Voxel at different LOD levels

This implementation builds upon the compact mesh generation system by introducing LOD support. It is important to note that both versions are necessary, as the LOD-based approach does not perform efficiently when LODs are not utilised, whereas the compact version handles such scenarios more effectively. The process begins by caching references to all six neighbouring chunks at the start of meshing. This caching reduces the need for expensive hierarchy queries during mesh generation.

Rather than examining every individual voxel, the mesh generator iterates in increments of l_o_d, effectively grouping a cube of $l\_o\_d^3$ voxels into a single representative "super-voxel." For each group, the average colour of all non-air voxels is computed to maintain visual consistency, and the number of air voxels is counted. If every voxel in the group is air, the group is skipped to improve performance.

If at least one voxel in the group is solid, the group is treated as solid. This conservative assumption simplifies face-visibility checks, avoiding the need to determine group boundaries or to calculate a threshold of solid versus air. Once a valid super-voxel is identified, the generator determines whether it lies on a chunk boundary and whether its neighbours are within the same or adjacent chunks. Neighbouring chunks must have their LOD levels established in advance to ensure correct face generation when different LODs abut.

Face generation proceeds by checking voxels adjacent to each face of the super-voxel. If an adjacent chunk's LOD level is greater than or equal to the current

level, a face is rendered only if no neighbour voxel exists, since a larger or equal cube in that chunk would otherwise cover it. If the adjacent chunk uses a finer LOD (i.e. higher resolution), then the face is omitted only if all corresponding voxels in that chunk are present; any absence necessitates rendering to prevent artefacts.

There is one special case in which a face is always rendered even if it may be covered. If a neighbouring chunk has the same or a coarser LOD and lacks an immediate neighbour voxel,but contains a solid voxel elsewhere within the same LOD group, the face is still rendered. Handling this exception precisely would require detecting the extent of the neighbouring group and iterating through its voxels, offering minimal gain at considerable computational cost. Since LODs already reduce overall face counts, this conservative strategy yields a net performance improvement.

On a separate note, if LOD levels change dynamically based on the player's position, neighbouring chunks must be updated accordingly. When a chunk's mesh is regenerated at a new LOD, faces in direct neighbours may become exposed; hence, both the current chunk and its adjacent chunks require mesh updates.

### 4.2.5 Complexity Analysis

To analyse the computational cost of mesh generation under different LODs, we examine voxel accesses, defined here as each call to `Get_Voxel(...)->IsAir()`. This provides an estimate of how often the CPU must query voxel data, which directly impacts performance.

#### Full Resolution (No LOD)

At full resolution, every voxel in the chunk, denoted $V$, is processed individually. For each voxel, the algorithm performs:

- One "self" fetch, to determine whether the voxel itself is solid or air.
- Six neighbour fetches, to determine which of its faces, if any, should be rendered.

This results in a total of $1 + 6 = 7$ accesses per voxel. Therefore, the total voxel-access count across the chunk can be defined as:

$$\text{Total accesses} = 7 \times V$$

#### With Level of Detail (LOD)

With LOD enabled, the process changes. Instead of evaluating each voxel's visibility independently, we group voxels into cubic blocks of size $L \times L \times L$ and compute a "super-voxel" to represent each block during meshing. Importantly, every voxel in the

chunk is still visited at least once to check whether the group is solid and to calculate the average colour. The key difference lies in neighbour checks and face evaluations. Each LOD group involves:

- $L^3$ accesses (one per voxel) for computing the average colour and solidity of the group.
- $6 \times L^2$ accesses to query the neighbour data of surface voxels, used to determine which faces to emit.

Hence, the number of accesses per group becomes:

$$\text{Per-group accesses} = L^3 + 6L^2$$

An alternative expression, using the total number of voxels $V$ in a group where $V = L^3$, is:

$$\text{Total accesses} = V + 6V \cdot \frac{1}{L}$$

For example, for $L = 4$ (so that $V = 64$), the number of neighbour checks is the following:

$$64 + 6 \times \frac{64}{4} = 64 + 96 = 160$$

#### Case Study

To illustrate, Table 4.3 shows total voxel accesses for a range of block sizes, each based on an LOD leve:

| **Block Size** ($B$) | **LOD 1** | **LOD 2** | **LOD 4** | **LOD 8** | **LOD 16** |
|---|---|---|---|---|---|
| $1^3 = 1$ | 7 | - | - | - | - |
| $2^3 = 8$ | 56 | 32 | - | - | - |
| $4^3 = 64$ | 448 | 256 | 160 | - | - |
| $8^3 = 512$ | 3584 | 2048 | 1280 | 896 | - |
| $16^3 = 4096$ | 28672 | 16384 | 10240 | 7168 | 5632 |

Table 4.3 Total voxel accesses under different LOD levels

Cells marked with "-" indicate configurations where the LOD block size does not divide evenly into the chunk, resulting in partial samples along edges (not considered here).

#### Implications and Complexity Breakdown

The main performance benefit of LOD in mesh generation does not stem from skipping voxels, since each voxel is still visited at least once, but from reducing the number of

neighbour queries and face evaluations. By grouping voxels into super-voxels, the algorithm avoids per-voxel visibility checks, significantly lowering computation, particularly in distant or less detailed regions. Additionally, LODs greatly reduce the number of faces rendered, as each group emits at most six faces regardless of how many voxels it contains, compared to six faces per voxel at full resolution.

Table 4.4 summarises the key stages of the mesh generation pipeline and their computational costs. Complexity is expressed in terms of:

- $V$: total number of voxels in the chunk
- $f$: number of visible faces generated

Table 4.4 Mesh generation stages and computational complexity

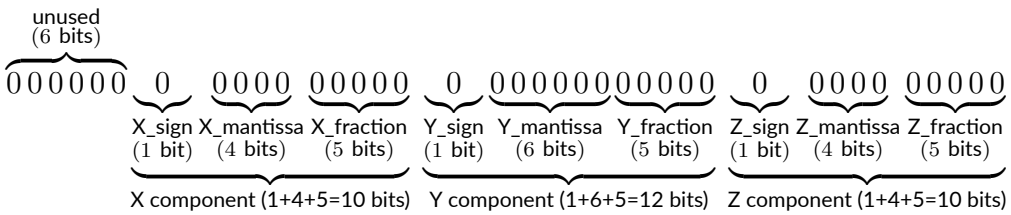
| **Stage** | **Workload** | **Complexity** |
|---|---|---|
| Voxel scan | $7V$ voxel-access checks | $O(V)$ |
| Face-flag computation | Constant work per voxel | $O(V)$ |
| Mesh-building loops | Iterate over voxels and faces | $O(V + f)$ |
| GPU buffer upload | Transfer mesh to GPU | $O(f)$ |
| Memory usage | Store mesh data | $O(f)$ |

Although the overall process is $O(V)$, performance varies significantly based on the number of faces generated ($f$) and implementation efficiency. This is especially relevant in real-time rendering, where reducing face count through LOD has a substantial impact even on relatively small chunks.

### 4.2.6 Mesh Compaction

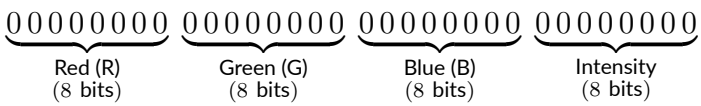
The collective chunk meshes we generate occupy a significant portion of GPU memory, especially on systems without a discrete GPU, and even more so when LODs are disabled or when distance thresholds are set extremely high. To use memory more efficiently, and to ease transfers between CPU and GPU (or to and from virtual memory), we apply a compaction scheme that packs each vertex's attributes into just three 32-bit floats:

- **Compact Location Data (CLD) (32 bits)**: the compacted vertex position



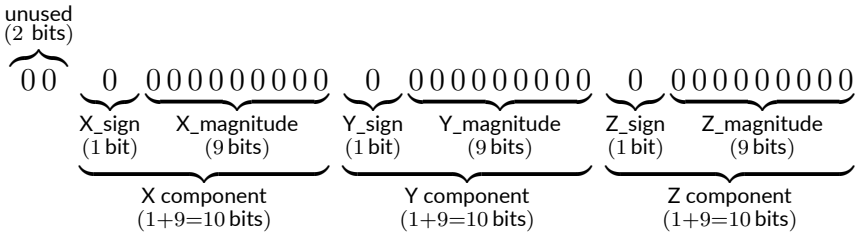
where each component encodes a signed fixed-point value.

- **Compact Colour Data (CCD) (32 bits):** the compacted colour and intensity




storing red, green, blue and intensity (alpha) as 8-bit unsigned integers.

- **Compact Normal Data (CND) (32 bits):** the compacted normal vector




encoding each normal component as a signed 9-bit magnitude.

Each of these three packed floats serves a distinct role: the compacted position (CLD) defines the mesh's shape, the compacted colour (CCD) is derived directly from the voxel's colour, and the compacted normal (CND) feeds into our deferred-lighting pass, enabling per-fragment Phong illumination and sharper visual distinctions between blocks.

Since CLD can only represent coordinates within a limited range (Table 4.5), approximately one chunk's extent plus a small margin, we supply a per-chunk uniform offset to the vertex shader, which first decodes the packed position and then applies the chunk's world offset. By packing into three floats rather than three `vec3`s, we reduce the memory footprint of each mesh to one third of the standard layout.

| Struct | Component(s) | Packed Range | Resolution |
|---|---|---|---|
| **CLD** | $X : \pm(2^4 - 2^{-5})$<br>$Y : \pm(2^6 - 2^{-5})$<br>$Z : \pm(2^4 - 2^{-5})$ | $\pm[0,\ 15.96875]$<br>$\pm[0,\ 63.96875]$<br>$\pm[0,\ 15.96875]$ | $\frac{1}{2^5} = 0.03125$ |
| **CCD** | $R, G, B, A$ integers: $0 \ldots 255$<br>Float range: $0.0 \ldots 1.0$ | $[0,\ 255]$ | $\frac{1}{255} \approx 0.00392$ |
| **CND** | $X, Y, Z$ unit vector components | $[-1.0,\ 1.0]$ | $\frac{1}{2^9 - 1} \approx 0.00196$ |

Table 4.5 Encoding properties of the CLD, CCD, and CND structs.

### 4.2.7 Additional Observations on Mesh Construction

To minimise the number of mesh updates per frame, LODs may be configured to refresh only when the player crosses predefined boundaries. Furthermore, the generation routine can be throttled so that it does not rebuild or update the mesh every frame; instead, it ensures the mesh is refreshed at fixed time intervals. In this way, mesh generation need only occur once every few frames, substantially reducing CPU load.

Our mesh generation pipeline is sufficiently fast to support real-time animations, such as running Conway's Game of Life [63] across the voxel world, one of the default simulation modes included in the project.

## 4.3 Dynamic Chunk Streaming

Rather than keeping the entire voxel world resident in memory, we employ a streaming mechanism that continuously loads nearby chunks and unloads those that stray beyond a configurable radius around the player. This approach ensures smooth navigation through vast terrains without exhausting CPU or GPU resources.

### 4.3.1 High-Level Workflow

1. **Detect Player Movement:** Each update, the player's world coordinates are rounded to the nearest integer and converted into discrete chunk and sector indices. If the player remains within the same chunk and `smart_update` is enabled, no further action is taken.

2. **Unload Distant Chunks:** Chunks whose centres lie outside the square radius $R_{\text{unload}}$ (measured in chunks) are identified and evicted. For each such chunk:

   - It is optionally written to disk for persistence.
   - It is removed from its parent sector's in-memory map.
   - Adjacent chunks are flagged for mesh and lighting updates to fill any newly exposed faces.

3. **Load Nearby Chunks:** Within a loading radius $R_{\text{load}}$ (typically equal to or slightly larger than $R_{\text{unload}}$), all chunk positions are examined in a 2D window around the current chunk. Missing chunks are either:

   - Read from disk, if a saved file exists, or
   - Procedurally generated on the fly via a callback.

Newly acquired chunks are inserted into their sectors and neighbouring nodes are marked for rebuild. When searching for saved chunks to load, we search through all possible chunk ”y” values loading any chunks that fit in the given ”x” and ”z” space.

By decoupling “load” and “unload” phases, and performing only inexpensive index tests until a chunk crosses the threshold, the system minimises stutters and keeps memory usage bounded.

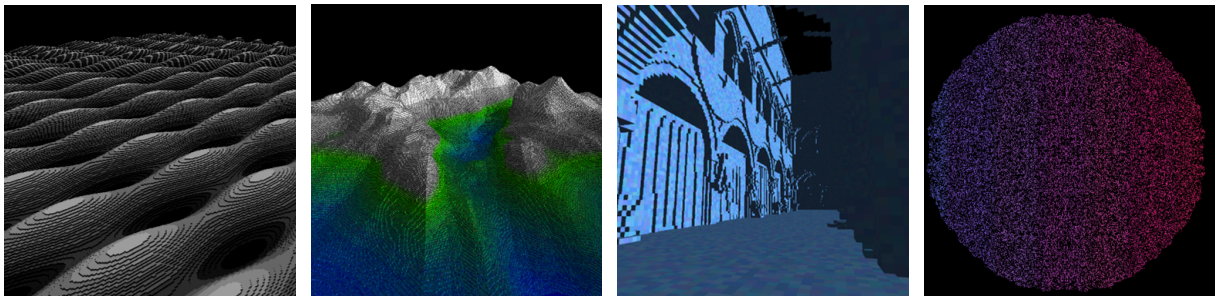
# 5 Evaluation

In the following chapter, we go over some performance statistics gathered to demonstrate the effects of various optimisations and evaluate the system as a whole. As a general note, these tests were performed on two devices:

**M1** **Development Desktop**: Intel i7-12700KF, 32 GB RAM, and an RTX 4070 (16 GB)

**M2** **Test Laptop**: Intel i5-5300U, 8 GB RAM

Unless otherwise specified, it can be assumed that results were obtained using the development desktop.



(a) Sinewave generation

(b) Ported mountain model [64]

(c) Ported sponza model [65]

(d) Conway Game of Life

Figure 5.1 Banner of the 4 different test cases used during the evaluation. Full scale versions can be found in Appendix B

## 5.1 Benchmarking Data Structures

| Sector DS | Chunk DS | Create (ms) | Mesh (ms) | Chunks/s | Faces/s |
|---|---|---|---|---|---|
| AVL | AVL | 1025 | 1155 | 3554 | 2,455,918 |
| UM | AVL | 1099 | 1103 | 3761 | 2,598,743 |
| AVL | UM | 729 | 1083 | 3795 | 2,622,147 |
| UM | UM | 858 | 1204 | 3411 | 2,356,847 |

Table 5.1 Average performance metrics over 100 runs for various data structure (DS) combinations.

Table 5.1 summarises average performance metrics gathered over 100 runs for each configuration tested, combining different data structures (DS) for managing sectors and chunks in our voxel rendering system. The tests used a fixed world setup consisting of 16,244,516 voxels distributed across 4 sectors and 4,096 chunks. Notably, employing a uniform map (UM) for chunks alongside an AVL tree for sectors

delivered the highest throughput in terms of chunks and faces processed per second. The AVL tree performs best for sectors, likely due to its lower calculation overhead and the relatively shallow tree structure when managing just four sectors, which outweighs the cost of using an unordered map. For chunks, the unordered map appears to be more efficient; this may be because the AVL tree's node traversal overhead exceeds the unordered map's faster lookup times, especially given the larger number of chunk nodes involved.

### 5.1.1 Impact of Alternative Voxel Storage Models

| Implementation | Create (ms) | Mesh (ms) | Chunks/s | Faces/s |
|---|---|---|---|---|
| Flat array (baseline) | 729 | 1083 | 3795 | 2,622,147 |
| `unordered_map` storage | 1216 | 1943 | 2113 | 1,459,939 |
| `unordered_map` + iterator mesh gen | 1292 | 1752 | 2342 | 1,618,595 |

Table 5.2 Comparison between flat array and `unordered_map` voxel storage strategies.

These tests used an AVL tree for sectors and an `unordered_map` for storing voxels within each chunk, running 100 iterations on the same world. The current solution stores voxel data in a flat array. Switching to a more dynamic data structure, in this case, an `unordered_map`, significantly slows both chunk creation and mesh generation. Mesh generation time almost doubled from 1083 ms to 1943 ms, while chunks per second dropped from 3795 to 2113 due to the overhead of managing a hash map. The middle test, which uses the `unordered_map` but checks every voxel, represents a worst-case scenario simulating a mostly solid world where every voxel must be processed. Here, the array's memory layout provides a clear advantage.

Modifying mesh generation to iterate only over existing map entries recovers some performance, reducing mesh time to 1752 ms and increasing throughput to 2342 chunks per second. Although still slower than the flat array, this demonstrates better utilisation of the data structure.

Overall, preferring a more storage-efficient data structure results in a noticeable loss of performance during chunk generation. Given that chunks are relatively small, we favour the added performance of a flat array, as the chunking system already effectively limits storage requirements.

## 5.2 Mesh Generation Versions

To evaluate the performance improvements across three mesh generation implementations, we tested each using an input of 4,194,304 voxels. The results are shown in Table 5.3.

| Version | Mesh Generation Time (ms) | Speed-up vs Previous |
|---|---|---|
| Straightforward | 4294.76 | – |
| Optimised | 121.33 | 35.38× |
| Compact | 113.21 | 1.07× |

Table 5.3 Mesh generation times and relative speed-ups for each version (4,194,304 voxels) on **M1**

As shown, there is a significant performance improvement between the ”Straightforward” and ”Optimised” implementations. This can be attributed to the bulk processing approach adopted in the optimised version, which reduces overhead from per-voxel operations.

The real surprise, however, comes from the ”Compact” version. Although this implementation introduces additional bitwise operations, it was initially assumed to run marginally slower. The slight speed-up observed may be due to the shorter code path, which benefits from better compiler optimisations and scheduling, allowing for more efficient, linear execution. During benchmarking, this version consistently shaved off a few milliseconds compared to the previous method (see Table 5.3).

## 5.3 Quantitative Analysis of Face Generation

As explained earlier, the meshing algorithm determines which voxel faces are visible and generates geometry accordingly. To test this, we ran the mesh generator on a single chunk to gather data. The theoretical worst case for the number of generated faces is the checkerboard pattern shown in [Fig. 5.2].

This is because each voxel generates all possible faces since none have neighbours. By contrast, the worst case for voxel access is a solid chunk, as each voxel must check its neighbours, but this results in a lower number

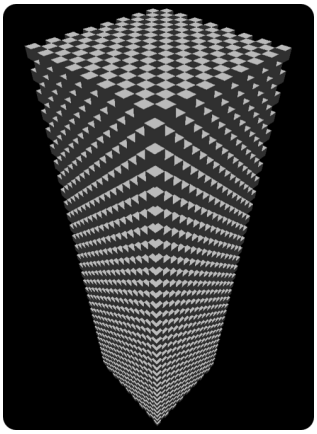


Figure 5.2 Chunk with a checkerboard voxel placement

of faces. These two extremes serve as baselines to compare with the test cases and results.

Table 5.4 shows the number of neighbouring voxel accesses performed on the CPU and the number of faces ultimately generated for the GPU, across various voxel arrangements within a single chunk. This test also includes the previously mentioned baseline cases.

The random distributions demonstrate how sparsity affects face generation: at 10% density, almost all faces are exposed, resulting in a high face count relative to the number of voxels. As density increases, the determining factors become the voxel count and the likelihood of hidden faces. Interestingly, the 75% density case shows a lower face count than the 50% case, suggesting that beyond a certain point, internal occlusion significantly reduces visible geometry, even though more voxels are present.

| Medium | Voxels | CPU Accesses | GPU Faces |
|---|---|---|---|
| Checkerboard | 8192 | 49152 | 49152 |
| Solid Chunk | 16384 | 98304 | 4608 |
| Random 10% | ∼1639 | 9834 | 8891 |
| Random 25% | ∼4090 | 24540 | 18705 |
| Random 50% | ∼8184 | 49104 | 25710 |
| Random 75% | ∼12285 | 73710 | 21038 |

Table 5.4 CPU voxel accesses and GPU face output across different voxel arrangements

To analyse this effect in more detail, a range of percentages was used to generate voxels inside a chunk (Fig. 5.3). This spanned from 0% to 100%, increasing in 5% increments, with 100 samples taken for each step. The average number of voxels and the average number of faces were then plotted against the percentage. The generated plot is surprisingly stable, closely mimicking the curve of $-9.37x^2 + 983.36x - 5.10$, which was obtained using Python.

## 5.4 Level of Detail (LOD) Evaluation

We can evaluate LODs in a similar manner (see Figure 5.4), although this time in the form of faces versus average number of voxels, using the previous plot as a baseline for comparison. It is important to note that each LOD level results in a plot that sits roughly halfway below the preceding one. That being said, the curves do not follow the same shape as the baseline, with LODs beyond the second level flattening out considerably, indicating minimal returns beyond a certain point.

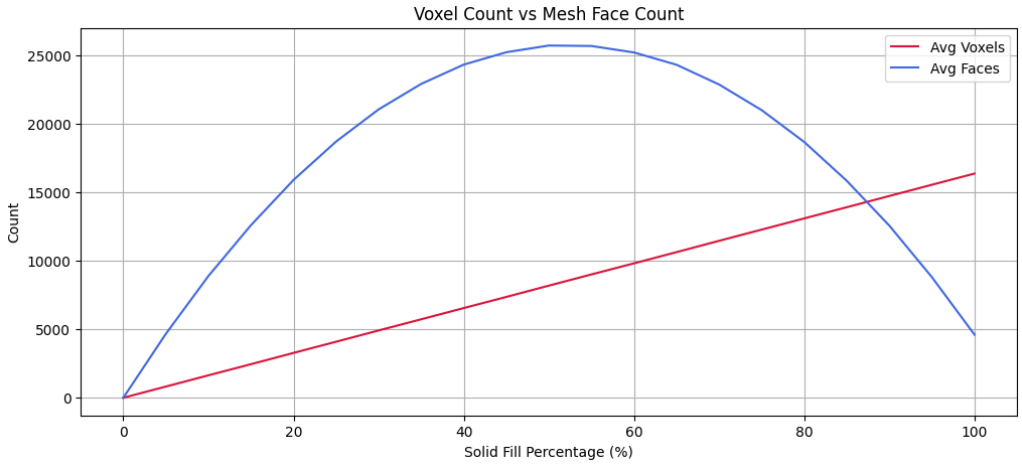



Figure 5.3 Average Voxels & Faces against percentage chance

Naturally, these would yield more noticeable benefits when rendering multiple chunks or when applied to models and environments. However, it remains valuable to examine their behaviour within the context of a single chunk.

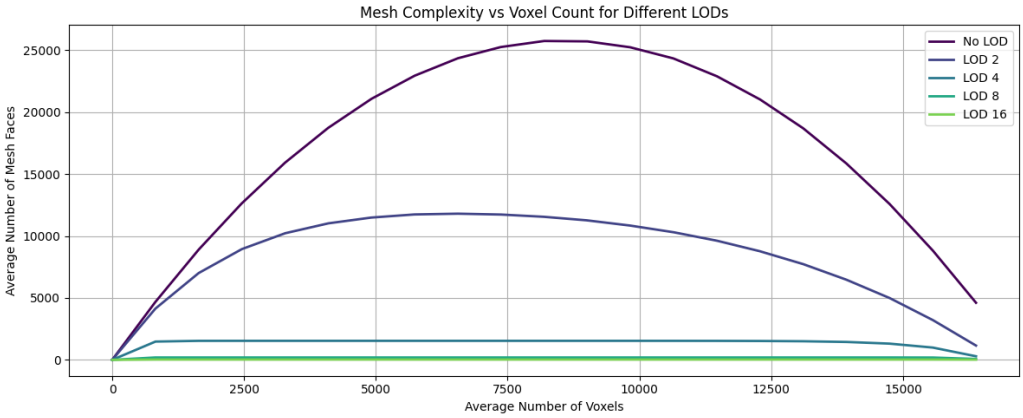



Figure 5.4 Faces generated by LOD level versus average number of voxels

## 5.5 Performance Evaluation on Voxelised 3D Models

For all tests, we used the following level-of-detail (LOD) distance thresholds, defining at what squared distance from the camera each LOD tier is applied: 50 units for `NORMAL`, 100 units for `LOD_2`, 200 units for `LOD_4`, and 400 units for `LOD_8`. These are only recalculated when the camera crosses a chunk boundary to avoid redundant computations. Smart rendering refers to rendering only those chunks that fall within the camera's frustum. We tested on two datasets: the **Sponza** scene (1.7 million voxels across 1,678 chunks and 4 sectors), and the **Mountain** scene (944 thousand voxels across 3,041 chunks and 4 sectors) [see Appendix B]. These two models were converted from `.obj` format into voxel streams using the Python code in Appendix A.

*Abbreviations:* LOD = Level-of-Detail, SR = Smart Rendering, CPU% = CPU Usage, GPU% = GPU Usage, CPU(ms) = CPU Time (ms), GPU(ms) = GPU Time (ms). CPU and GPU times may overlap due to asynchronous command submission, so their sum can exceed the frame time. Additionally, querying GPU times incurs a slight performance cost due to CPU-GPU synchronisation.

| Device | Model | LOD | SR | CPU% | GPU% | CPU (ms) | GPU (ms) |
|---|---|---|---|---|---|---|---|
| M1 | Sponza | Off | Off | 13% | 3% | 0.08 | 0.8 |
| M1 | Sponza | On | Off | 13% | 3% | 0.08 | 0.4 |
| M1 | Sponza | Off | On | 13% | 3% | 0.3 | 0.3 |
| M1 | Sponza | On | On | 13% | 3% | 0.06 | 0.25 |
| M1 | Mountain | Off | Off | 6% | 5% | 1.5 | 0.5 |
| M1 | Mountain | On | Off | 6% | 5% | 1.5 | 0.35 |
| M1 | Mountain | Off | On | 6% | 5% | 0.57 | 0.27 |
| M1 | Mountain | On | On | 6% | 5% | 0.55 | 0.17 |
| M2 | Sponza | Off | Off | 15% | 90% | 7 | 33 |
| M2 | Sponza | On | Off | 25% | 80% | 10 | 14 |
| M2 | Sponza | Off | On | 20% | 87% | 6 | 22 |
| M2 | Sponza | On | On | 25% | 80% | 6 | 12 |
| M2 | Mountain | Off | Off | 25% | 90% | 14.5 | 22 |
| M2 | Mountain | On | Off | 25% | 70% | 10 | 9 |
| M2 | Mountain | Off | On | 25% | 80% | 8.7 | 12.3 |
| M2 | Mountain | On | On | 20% | 70% | 6 | 5.5 |

Table 5.5 Benchmark results for voxelised models on Devices M1 and M2. CPUt and GPUt are measured in milliseconds (ms).

**Device M1** On the high-end desktop (M1), enabling smart rendering clearly improves performance. For instance, on the Mountain scene, enabling smart rendering without LOD drops the CPU time from 1.5 ms to 0.57 ms. However, enabling LOD alone does not bring a significant improvement, the Mountain scene with LOD but without smart rendering still takes 1.5 ms CPU time. This suggests that for powerful devices with ample memory and GPU capacity, the bottleneck is not geometry complexity but rather the number of draw calls. The Sponza model consistently performs better than Mountain across all modes (e.g., 0.25 ms GPU time vs 0.17 ms for best-case Sponza and Mountain respectively), likely due to its lower chunk count. Reducing the number of draw calls further, currently at least two per chunk, could yield additional gains on

this class of hardware.

**Device M2** On the more constrained M2 laptop with an integrated GPU, the introduction of LOD makes a substantial difference. For example, in the Mountain scene, enabling both LOD and smart rendering reduces GPU time from 22 ms (no LOD, no smart) to just 5.5 ms. This is a 4× reduction in GPU time and a 2.6× reduction in CPU time (from 14.5 ms to 6 ms), making the application far more responsive. Even enabling LOD alone (without smart) drops GPU time to 9 ms. This highlights how memory-efficient representations and mesh simplification are critical under bandwidth and memory constraints. The performance impact of smart rendering alone is also evident: on the Sponza model without LOD, enabling smart rendering reduces GPU time from 33 ms to 22 ms.

**Key Takeaways** These results confirm that the engine adapts effectively to different hardware profiles. On high-end systems, reducing draw calls could provide further benefit, though the current performance is already highly efficient. On lower-end systems, mesh simplification through LOD is vital for keeping execution times reasonable. Eliminating mesh compaction might aid performance slightly on desktops, but would likely triple mesh size, degrading performance on mobile-class devices. Overall, the design decisions appear well-balanced: they prioritise general-purpose usability while preserving acceptable performance across a wide range of devices.

## 5.6 Chunk Generation and Updates Stress Test

By simulating a game of Conway's Game of Life [63] [see Appendix B], we can effectively benchmark performance in terms of mesh generation by updating a flat world with a height of just one voxel.

Simulating the Game of Life over a $1024 \times 1024$ area is sufficient to reach the update rate plateau on the desktop system **M1**. Mesh generation begins to cause a noticeable frame drop at around 2000 mesh generations per second, and the generation rate stabilises at around 4300-4500 chunks per second. This high threshold demonstrates how efficiently the mesh system handles sustained load.

On the laptop system **M2**, frame rate degradation begins around 700 chunks per second and plateaus at approximately 1500. Despite the lower performance, the system still maintains a high mesh generation throughput on more constrained hardware.

# 6 Conclusion

## 6.1 Future Work

This project shows that a novel data structure can enhance voxel rendering performance, but there remains scope for several key improvements. By adopting hierarchical spatial structures such as SVOs, the engine could dramatically reduce memory usage and achieve finer level-of-detail control; a GPU-based ray-marching approach might then determine on a per-pixel basis which voxels to draw, further boosting efficiency. Alongside this, integrating the marching cubes algorithm would allow seamless switching between the existing cubic style and smooth, high-resolution meshes, making the engine suitable for both stylised and organic visualisations. Alternatively, geometry shaders could be explored to generate voxel meshes directly on the GPU, potentially simplifying the pipeline and improving performance in dynamic scenes. Further refinements could include advanced lighting and shading, ambient occlusion, real-time global illumination or shadow mapping, to increase realism, with a subtle fog effect smoothing the appearance of loading or unloading chunks. Offloading mesh generation and data traversal to compute shaders would free the CPU and support larger dynamic worlds, while adding physics features such as collision detection, particle interactions or fluid simulations would broaden the engine's applicability to interactive and scientific scenarios. Exploring these areas will help transform this lightweight voxel engine into a versatile platform for a wide range of applications.

## 6.2 Closing Remarks

This project set out to determine whether a novel, lightweight data structure could underpin an effective voxel engine. The implemented system demonstrates that it can, yielding notable improvements in memory access, rendering efficiency, and chunk-level streaming. Benchmarks across various voxel configurations show strong performance for moderately sized scenes, supporting dynamic updates, efficient LOD rendering, and reduced draw sizes via smart mesh compaction. For the defined use case, real-time rendering of procedurally generated, editable voxel worlds, the solution proves robust and scalable. While future work could improve support for extreme scales, smoother surfaces, and more realistic lighting, the current engine provides a solid foundation for voxel-based environments. The results validate the original hypothesis and highlight a promising path for lightweight, modular voxel rendering systems.

# References


[1] Contributors to Wikimedia projects, *Voxel*, https://en.wikipedia.org/w/index.php?title=Voxel&oldid=1283435877, [Online; accessed 7-Apr-2025], Apr. 2025.

[2] *What is mesh 3d? - download*, [Online; accessed 22 Apr. 2025], Apr. 2025. [Online]. Available: https://artgraphic3d.com/179-what-is-mesh-3d.

[3] J. D. Foley *et al.*, *Computer Graphics: Principles and Practice*, 3rd. Addison–Wesley Professional, 2013, ISBN: 978-0321399526.

[4] E. Haines and T. Akenine-Möller, Eds., *Real-Time Rendering, Fourth Edition*. CRC Press, 2019, ISBN: 978-0367332984.

[5] M. Humphreys, D. S. Blythe, C. Engel, R. Franchek, M. Kilgard, and L. Dai, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5*, 9th. Addison-Wesley Professional, 2016, ISBN: 978-0134495497.

[6] J. de Vries, *Getting started – learnopengl*, LearnOpenGL.com, [Online; accessed 5 May 2025], 2020. [Online]. Available: https://learnopengl.com/.

[7] Khronos Group, *Opengl specification, version 4.6 core profile*, [Online; accessed April 2025], 2017. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[8] Khronos OpenGL Wiki, *Rendering pipeline overview*, [Online; accessed 5 May 2025], 2022. [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[9] Khronos OpenGL Wiki, *The rendering pipeline*, [Online; accessed 5 May 2025], 2022. [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline.

[10] StackOverflow user Moose, *At what stage is clipping performed in the graphics pipeline?* StackOverflow, [Online; accessed 5 May 2025], 2019. [Online]. Available: https://stackoverflow.com/questions/60910464/at-what-stage-is-clipping-performed-in-the-graphics-pipeline.

[11] V. Kumar, "Understanding opengl rendering pipeline stages," *Medium*, 2025, [Online; accessed 5 May 2025]. [Online]. Available: https://medium.com/@vinishkumar/understanding-opengl-rendering-pipeline-stages-f85849c63ef3.

[12] Unity Technologies, *Optimizing draw calls*, Unity Manual, version 2023.2, 2023. [Online]. Available: https://docs.unity3d.com/2023.2/Documentation/Manual/optimizing-draw-calls.html.

[13] Arm Ltd., *Draw-call batching best practices*, Arm Developer Documentation, 2020. [Online]. Available: https://developer.arm.com/documentation/101897/latest/Optimizing-application-logic/Draw-call-batching-best-practices.

[14] Wikipedia contributors, *Geometry instancing*, Wikipedia, The Free Encyclopedia, [Online; accessed 22. Apr. 2025], 2024. [Online]. Available: https://en.wikipedia.org/wiki/Geometry_instancing.

[15] M. Kenzel, B. Kerbl, W. Tatzgern, E. Ivanchenko, D. Schmalstieg, and M. Steinberger, "On-the-fly vertex reuse for massively-parallel software geometry processing," *arXiv preprint arXiv:1805.08893*, 2018. [Online]. Available: https://arxiv.org/abs/1805.08893.

[16] C. Cyril, "Gigavoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes," [Online; accessed 8-Apr-2025], Ph.D. Thesis, Université de Grenoble, Grenoble, France, 2011. [Online]. Available: https://maverick.inria.fr/Publications/2011/Cra11/CCrassinThesis_EN_Web.pdf.

[17] Blender Foundation, *Download—blender.org*, [Online; accessed 8. Apr. 2025], Apr. 2025. [Online]. Available: https://www.blender.org/download.

[18] S. E. S. Inc, *Houdini - 3d modeling, animation, vfx, look development, lighting and rendering | sidefx*, [Online; accessed 8. Apr. 2025], Apr. 2025. [Online]. Available: https://www.sidefx.com.

[19] L. Software, *Free 3d modeling software | 3d design online | sketchup free subscription | sketchup*, [Online; accessed 8. Apr. 2025], Apr. 2025. [Online]. Available: https://www.sketchup.com/en/plans-and-pricing/sketchup-free.

[20] Jasper. "The small detail behind breath of the wild's thousands of trees." [Online; accessed 8-Apr-2025]. (2021), [Online]. Available: https://www.youtube.com/watch?v=sh6s17WnWBM.

[21] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, vol. 18, New York, NY, USA: ACM, Jan. 1984, pp. 137–145. DOI: 10.1145/800031.808590.

[22] K. Winner, “Adaptive supersampling using machine learning techniques,” University of Maryland, Baltimore County, Baltimore, MD, USA, Technical Report, 2004, [Online; accessed 9-Apr-2025]. [Online]. Available: https://userpages.cs.umbc.edu/olano/635/winnerk1.pdf.

[23] C. Yang *et al.*, “A survey of deep learning for image super-resolution,” *Computer Vision and Pattern Recognition*, 2020, Accessed: 2025-05-24. [Online]. Available: https://research.nvidia.com/labs/rtr/publication/yang2020survey/.

[24] C. Yang *et al.*, “Deep learning super sampling (dlss) 4: Real-time ai denoising and super resolution,” *NVIDIA Research*, 2023, Accessed: 2025-05-24. [Online]. Available: https://research.nvidia.com/labs/adlr/DLSS4/.

[25] H. Li and S. Kim, “Mesh simplification algorithms for rendering performance,” *International Journal of Engineering Research and Technology*, vol. 13, no. 6, pp. 1110–1119, 2020. [Online]. Available: https://www.ripublication.com/irph/ijert20/ijertv13n6_05.pdf.

[26] H. Hoppe, “Progressive meshes,” in *Proceedings of the 2023 ACM SIGGRAPH Conference*, vol. 2, New York, NY, USA: ACM, Aug. 2023, pp. 111–120. DOI: 10.1145/3596711.3596725.

[27] J. Shepherd, “Quadrilateral mesh simplification,” *ACM Transactions on Graphics*, Jan. 2008. [Online]. Available: https://www.academia.edu/20263146/Quadrilateral_mesh_simplification.

[28] E. Lengyel, *Voxel-based terrain for real-time virtual simulations*, https://docslib.org/doc/12989980/voxel-based-terrain-for-real-time-virtual-simulations, [Online; accessed 9-Apr-2025], Apr. 2025.

[29] M. Aleksandrov, S. Zlatanova, and D. J. Heslop, “Voxelisation algorithms and data structures: A review,” *Sensors*, vol. 21, no. 24, p. 8241, Dec. 2021, ISSN: 1424-8220. DOI: 10.3390/s21248241.

[30] S. W. Wang and A. E. Kaufman, “Volume-sampled 3d modeling,” *IEEE Computer Graphics and Applications*, vol. 14, no. 5, pp. 26–32, Aug. 2002. DOI: 10.1109/38.310721.

[31] J. J. Ryde and H. F. Durrant-Whyte, “Lattice occupied voxel lists for representation of spatial occupancy,” in *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, Taipei, Taiwan: IEEE, Oct. 2010, pp. 3111–3116. DOI: 10.1109/IROS.2010.5650175. [Online]. Available: https://cse.buffalo.edu/~jryde/publications/ryde2010_iros_lattice_occupied_voxel_lists.pdf.

[32] C. Sigg, R. Peikert, M. Gross, and T. Müller, "Signed distance transform using graphics hardware," in *Proceedings of IEEE Visualization 2003 (VIS '03)*, Seattle, WA, USA: IEEE, Oct. 2003, pp. 83–90. [Online]. Available: https://cs.brown.edu/research/vis/docs/pdf/Sigg-2003-SDT.pdf.

[33] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on gpus," *ACM Transactions on Graphics*, vol. 29, no. 6, pp. 1–10, Dec. 2010, ISSN: 0730-0301. DOI: 10.1145/1882261.1866201.

[34] C. Crassin and S. Green, *Octree-based sparse voxelization using the gpu hardware rasterizer*, NVIDIA Research Publication, [Online; accessed 10-Apr-2025], Jul. 2012. [Online]. Available: https://research.nvidia.com/publication/2012-07_octree-based-sparse-voxelization-using-gpu-hardware-rasterizer.

[35] R. E. Bridson, "Computational aspects of dynamic surfaces," [Online; available: https://www.cs.ubc.ca/~rbridson/docs/rbridson_phd.pdf], Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2003.

[36] M. B. Nielsen and K. Museth, "Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets," *Journal of Scientific Computing*, vol. 26, no. 3, pp. 261–299, Mar. 2006, ISSN: 1573-7691. DOI: 10.1007/s10915-005-9062-8.

[37] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Transactions on Graphics*, vol. 32, no. 3, 27:1–27:22, Jun. 2013, [Online; available: https://museth.org/Ken/Publications_files/Museth_TOG13.pdf].

[38] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis, "Spgrid: A sparse paged grid structure applied to adaptive smoke simulation," *ACM Transactions on Graphics*, vol. 33, no. 6, pp. 1–12, Nov. 2014, ISSN: 0730-0301. DOI: 10.1145/2661229.2661269.

[39] J. Wolper and J. Carstenson, "Ray tracing complex objects using octrees," Swarthmore College, Swarthmore, PA, USA, Course Report, https://www.cs.swarthmore.edu/~jcarste1/pdfs/octree.pdf.

[40] S. Laine and T. Karras, *Efficient sparse voxel octrees: Analysis, extensions, and implementation*, NVIDIA Research Publication, [Online; accessed 15-Apr-2025], Feb. 2010. [Online]. Available: https://research.nvidia.com/publication/2010-02_efficient-sparse-voxel-octrees-analysis-extensions-and-implementation.

[41] Y. Pan, "Dynamic update of sparse voxel octree based on morton code," [Online; accessed 17-Apr-2025], Master's Thesis, Purdue University, West Lafayette, IN, USA, 2021. [Online]. Available: https://hammer.purdue.edu/ndownloader/files/27771012.

[42] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, “Interactive indirect illumination using voxel cone tracing,” *Computer Graphics Forum*, vol. 31, no. 7, pp. 1921–1930, Oct. 2012. [Online]. Available: https://www.icare3d.org/research/publications/Cra12/04_crassinVoxels_bps2012.pdf.

[43] Contributors to Wikimedia projects, *Sparse voxel octree*, Russian Wikipedia, [Online; accessed 16-Apr-2025], Apr. 2025. [Online]. Available: https://ru.wikipedia.org/w/index.php?title=Sparse_Voxel_Octree&oldid=144527352.

[44] *What is minecraft? build, discover realms & more*, Official Minecraft Website, [Online; accessed 17-Apr-2025], Apr. 2025. [Online]. Available: https://www.minecraft.net/en-us/about-minecraft.

[45] L. Niu, Z. Wang, Z. Lin, Y. Zhang, Y. Yan, and Z. He, “Voxel-based navigation: A systematic review of techniques, applications, and challenges,” *ISPRS International Journal of Geo-Information*, vol. 13, no. 12, p. 461, Dec. 2024, ISSN: 2220-9964. DOI: 10.3390/ijgi13120461.

[46] M. Fang, C. Mei, and R. Yang, “A survey of real-time voxel-based rendering techniques,” *Journal of Computer Graphics Techniques*, vol. 14, no. 1, pp. 1–28, 2025.

[47] *Enshrouded*, [Online; accessed 17. Apr. 2025], Dec. 2024. [Online]. Available: https://enshrouded.com.

[48] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3d surface construction algorithm,” in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 21, ACM, 1987, pp. 163–169.

[49] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel, “High-speed marching cubes using histopyramids,” *Computer Graphics Forum*, vol. 27, no. 8, pp. 2028–2039, 2008.

[50] F. Goetz, T. Junklewitz, and G. Domik, “Real-time marching cubes on the vertex shader,” in *Eurographics Short Presentations*, Eurographics Association, 2005.

[51] X. Wang, S. Gao, M. Wang, and Z. Duan, “A marching cube algorithm based on edge growth,” *arXiv preprint arXiv:2101.00631*, 2021, [Online; accessed 17. Apr. 2025]. [Online]. Available: https://arxiv.org/abs/2101.00631.

[52] Voxel Tools Contributors, *Overview - Voxel Tools Documentation*, [Online; accessed 14 May 2025], 2025. [Online]. Available: https://voxel-tools.readthedocs.io/en/latest/overview/.

[53] T. Labs, *Teardown on Steam*, [Online; accessed 17. Apr. 2025], Apr. 2025. [Online]. Available: https://store.steampowered.com/app/1167630/Teardown.

[54] P. O. Tvete, “Qtquick3d instanced rendering,” *The Qt Company Blog*, Feb. 2021. [Online]. Available: https://www.qt.io/blog/qtquick3d-instanced-rendering.

[55] C. Garrett, “Voxel performance: Instancing vs chunking,” *Medium*, Dec. 2021, [Online; accessed 18. Apr. 2025]. [Online]. Available: https://medium.com/@claygarrett/voxel-performance-instancing-vs-chunking-9643d776c11d.

[56] mikolalysenko, *Meshing in a minecraft game*, [Online; accessed 18. Apr. 2025], Jan. 2015. [Online]. Available: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game.

[57] S. Jabłoński and T. Martyn, “Real-time voxel rendering algorithm based on screen space billboard voxel buffer with sparse lookup textures,” in *Proceedings of the 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, Diamond Congress Ltd., 2016, pp. 123–130.

[58] B. Zaugg and P. K. Egbert, “Voxel column culling: Occlusion culling for large terrain models,” in *Proceedings of the Symposium on Data Visualisation 2001*, Eurographics Association, 2001, pp. 85–94. DOI: 10.2312/VisSym/VisSym01/085-094.

[59] O. Nousiainen, “Performance comparison on rendering methods for voxel data,” Aalto University, Master’s thesis, 2019. [Online]. Available: https://aaltodoc.aalto.fi/bitstream/handle/123456789/40891/urn_nbn_fi_aalto-201904213115.pdf.

[60] J. Gedge, *Greedy voxel meshing*, [Online; accessed 21. Apr. 2025], Apr. 2025. [Online]. Available: https://gedge.ca/blog/2014-08-17-greedy-voxel-meshing.

[61] GeeksforGeeks, *Introduction to avl tree*, [Online; accessed 20 May 2025], 2021. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-avl-tree/.

[62] GeeksforGeeks, *Introduction to red-black tree*, [Online; accessed 20 May 2025], 2021. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-red-black-tree/.

[63] Wikipedia contributors, *Conway’s game of life — wikipedia, the free encyclopedia*, Accessed: 2025-05-18, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[64] shaiksha35, *Mountain Free 3D Model - Free3D*, https://free3d.com/3d-model/mountain-6839.html, Accessed: May 18, 2025, n.d.

[65] M. McGuire, *G3d innovation engine*, https://casual-effects.com/g3d/data10/index.html, Accessed: May 18, 2025, 2018.

# Appendix A Python Code

```python
import trimesh
import numpy as np
import os

# -- Load and prepare mesh --

# Load the full mesh (handle multi-part scenes)
mesh = trimesh.load("sponge.obj")
if isinstance(mesh, trimesh.Scene):
    mesh = trimesh.util.concatenate(mesh.geometry.values())

# Get bounding box
bounds_min, bounds_max = mesh.bounds

# -- Set voxelisation parameters --

grid_size = 4       # Split into 4x4x4 regions
pitch = 1.0         # Voxel size
all_voxels = []     # Store all voxelised subregions

# -- Split mesh into subregions and voxelise each --

for xi in range(grid_size):
    for yi in range(grid_size):
        for zi in range(grid_size):
            # Compute bounds of current subregion
            min_corner = bounds_min + (bounds_max - bounds_min) * np.array([xi, yi, zi]) / grid_size
            max_corner = bounds_min + (bounds_max - bounds_min) * np.array([xi + 1, yi + 1, zi + 1]) / grid_size

            # Find vertices in region and faces with any such vertex
            vertex_mask = np.all((mesh.vertices >= min_corner) & (mesh.vertices <= max_corner), axis=1)
            face_mask = np.any(vertex_mask[mesh.faces], axis=1)

            if not np.any(face_mask):
                continue

            # Extract submesh and voxelise
            submesh = mesh.submesh([face_mask], only_watertight=False, append=True)
            if submesh and len(submesh.faces) > 0:
                try:
```

```
                    vox = submesh.voxelized(pitch=pitch)
                    if vox and vox.points.any():
                        vox_int = np.round(vox.points / pitch).astype(int)
                        all_voxels.append(vox_int)
                        print(f"Voxelised region {xi},{yi},{zi} with {len(vox_int)} voxels.")
                except Exception as e:
                    print(f"Error voxelising region {xi},{yi},{zi}: {e}")

# -- Combine and save all voxel data --

if all_voxels:
    all_voxels = np.vstack(all_voxels)
    all_voxels = np.unique(all_voxels, axis=0)  # Remove duplicates
    np.savetxt("sponge.txt", all_voxels, fmt='%d')
    print(f"Saved {len(all_voxels)} voxel positions to sponge_split_voxels.txt")
else:
    print("No voxels generated.")
```

Listing A.1 Mesh Splitting and Voxelisation

# Appendix B Models used during evaluation

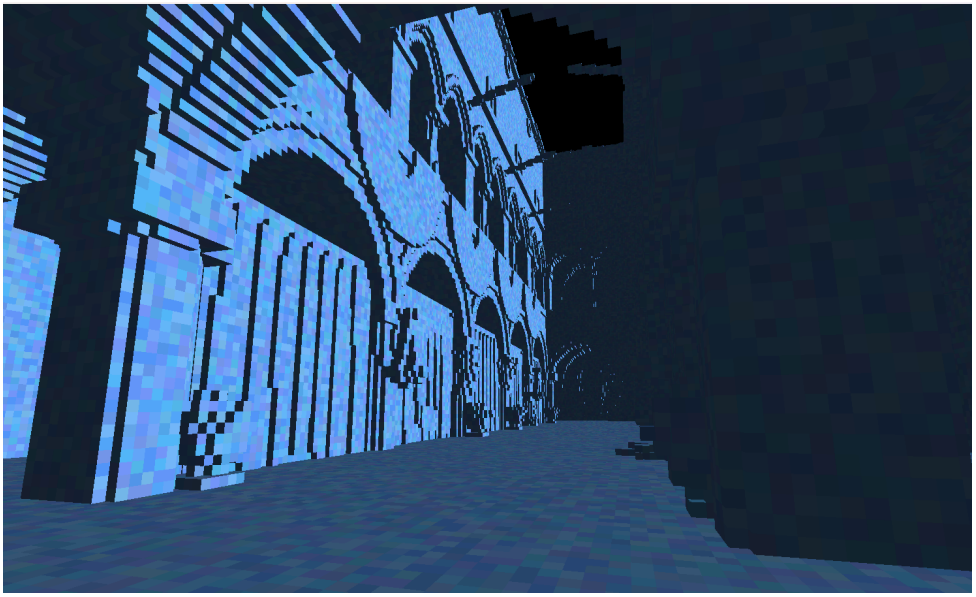


Figure B.1 Sponza model imported and converted from [65].

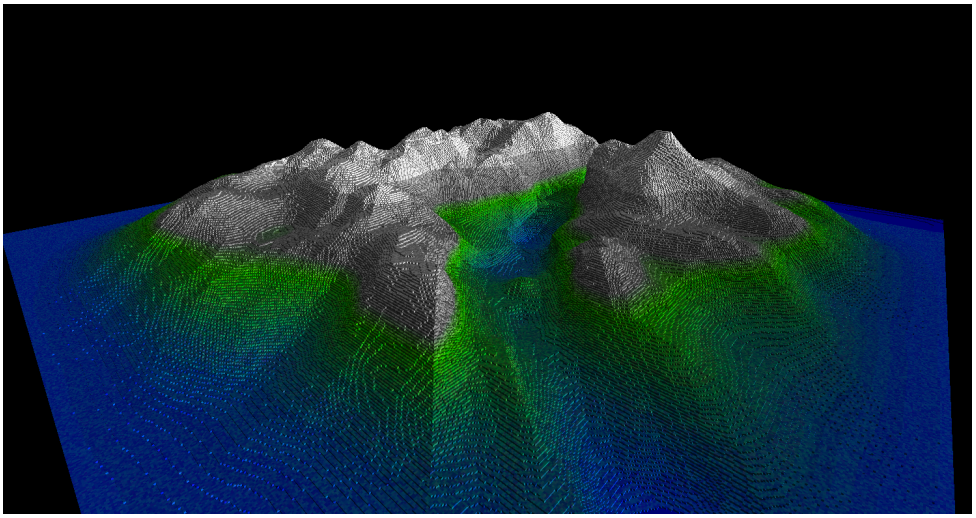


Figure B.2 Mountain model imported and converted from [64].

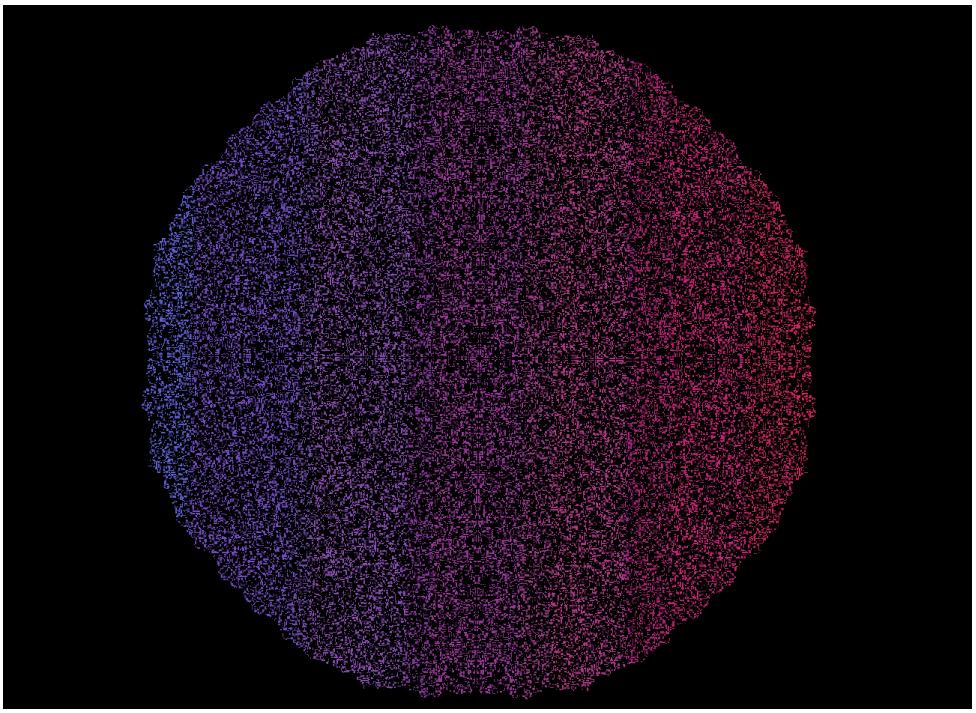


Figure B.3 Conway's Game of Life simulation [63], with a size of $1024 \times 1024$.

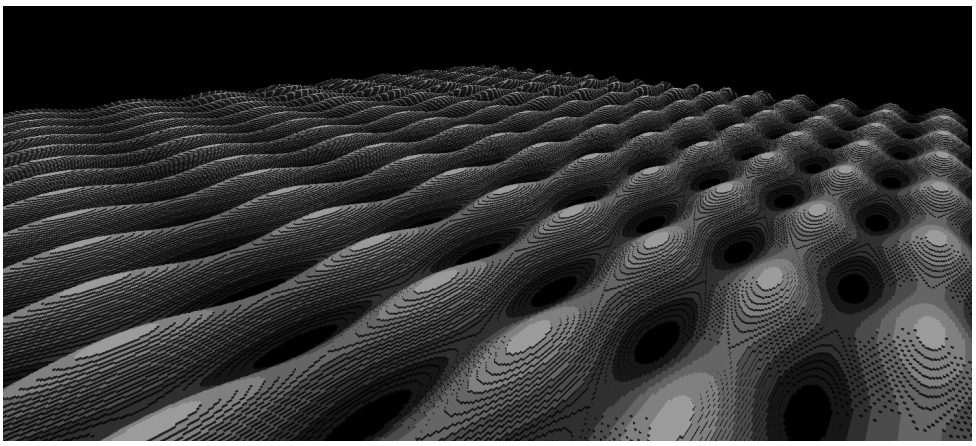


Figure B.4 Voxel world with wavy terrain generated by two sine waves. Used for performance tests in Table 5.1.