BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

A NEW INTERACTIVE ALGORITHM FOR CONVERTING A VOXEL MODEL TO A MESH IN UNREAL ENGINE

NOVÝ INTERAKTIVNÍ ALGORITMUS PRO PŘEVOD VOXELOVÉHO MODELU NA MESH V UNREAL ENGINU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

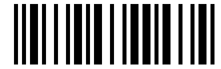
AUTHOR **PAVEL BALUSEK**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment



Institut:	Department of Computer Graphics and Multimedia (DCGM)	162650
Student:	**Balusek Pavel**	
Programme:	Information Technology	
Title:	**A New Interactive Algorithm for Converting a Voxel Model to a Mesh in Unreal Engine**	
Category:	Computer Graphics	
Academic year:	2024/25	

Assignment:

1. Study and familiarize yourself with Unreal Engine and how it can process voxel models. Research algorithms for creating voxel models. Study the algorithms used for converting voxel models to polygonal ones. Find competitive algorithms for converting voxel models to mesh. Identify a tool or design a procedure for measuring the performance of voxel model conversion algorithms in Unreal Engine.
2. Propose your own algorithm capable of converting procedurally generated voxel models to polygonal directly from a compressed representation while allowing interaction with the voxel model.
3. Implement the proposed algorithm in Unreal Engine so that interactions are possible and set up the competitive algorithms.
4. Measure the efficiency of the algorithms from point 3 and compare the measurement results.
5. Write conclusions from the measurements, describe the properties of the algorithms, and select the fastest algorithm. Suggest possible extensions, publish the work, and create a demonstration video.

Literature:

- Cyril Crassin. GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes. https://maverick.inria.fr/Membres/Cyril.Crassin/thesis/CCrassinThesis_EN_Web.pdf

Requirements for the semestral defence:
Points 1 - 3, application prototype, and two implemented meshing algorithms.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Milet Tomáš, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	12.11.2024

Abstract

This thesis presents Run Directional Meshing (RDM), a new voxel meshing algorithm developed in Unreal Engine and designed to enable real-time conversion of voxel models into polygonal meshes using unidirectional traversal and single-pass face merging. Two prototype implementations were created: a grid-based variant serving as a conceptual testbed, and a run-length encoded (RLE) variant capable of operating directly on compressed voxel data. Both variants support real-time editing, but remain unoptimized due to inefficient buffer handling, and in the RLE implementation, a lack of complete face culling. The algorithm was implemented as a Unreal Engine plugin, and compared against the Voxel Plugin alternative. While RDM demonstrated promising memory savings through compression, its meshing performance was inferior in comparison due to its unoptimized state. Despite this, the results validate the potential of direct meshing from compressed voxel representations and provide a foundation for future development of efficient voxel rendering systems in interactive environments.

Abstrakt

Tato bakalářská práce představuje algoritmus Run Directional Meshing (RDM), nový způsob převodu voxelových modelů na polygonální mesh, vyvinutý v prostředí Unreal Engine. Algoritmus je navržen tak, aby umožňoval převod v reálném čase pomocí jednosměrného průchodu a slučování stěn během tohoto průchodu. Byly vytvořeny dva prototypy: varianta založená na mřížce, sloužící jako konceptuální testovací prostředí, a varianta využívající RLE kódování, která je schopna pracovat přímo s komprimovanými voxelovými daty. Obě varianty podporují úpravy v reálném čase, ale nejsou optimalizované kvůli neefektivní práci s buffery a v případě RLE implementace také kvůli neúplnému odstraňování neviditelných stěn. Algoritmus byl implementován jako plugin pro Unreal Engine a porovnán s alternativou Voxel Plugin. I když RDM díky kompresi vykazoval slibné úspory paměti, jeho výkon při převodu na mesh byl kvůli neoptimalizovanému stavu horší. Přesto výsledky potvrzují proveditelnost přímého převodu z komprimované voxelové reprezentace a poskytují základ pro budoucí vývoj efektivních systémů pro vykreslování voxelů v interaktivních prostředích.

Keywords

Run Directional Meshing, RDM, Meshing, Remeshing, Voxel Meshing, Voxel, Voxel Engine, Voxel Model to Mesh Conversion, Voxel Models, Voxel Mesh, Mesh Generation, Real-Time Meshing, Voxel Grid, Voxel Compression, RLE , Collision, Interactive Voxel Models, Quad Merging, Voxel Face, Unreal Engine, UE, C++, Voxel Meshing Performance Profiling, Voxel Plugin, Isosurface extraction

Klíčová slova

Run Directional Meshing,RDM , Meshing, Remeshing, Voxel, Voxel Meshing, Voxelový Engine, Převod voxelového modelu na mesh, Voxelové modely,, Voxelový Mesh, Generace meshů, Real-Time Meshing, Voxelová Mřížka, Komprese Voxelů, RLE, Kolize, Interaktivní Voxelové Modely, Sjednocování Quadů, Voxel Face, Unreal Engine, UE, C++, Měření výkonu voxel meshingu, extrakce isopovrchu

Reference

BALUSEK, Pavel. *A New Interactive Algorithm for*
Converting a Voxel Model to
a Mesh in Unreal Engine. Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

A New Interactive Algorithm for Converting a Voxel Model to a Mesh in Unreal Engine

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Tomáš Milet Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Pavel Balusek
May 14, 2025

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Milet Ph.D., for long hours of consultation, Martin Rybka for motivation, support, and help with simplified formulation of the problem, and all people who express support and interest in my work.

Contents

1 Introduction 7

2 Motivation 8

2.1 Voxel . . . 8
2.2 Voxel models . . . 8
2.3 Meshing . . . 9
2.4 Remeshing and mesh repair . . . 9
2.5 Isosurface extraction . . . 10
2.6 Neural Representations and Hybrid Approaches . . . 10
2.7 Voxel engines . . . 11
2.8 Voxel Meshing . . . 13
2.9 Visual goals . . . 14
2.10 Performance Goals . . . 15
2.10.1 Performance Factors . . . 16
2.11 Interactivity Goals . . . 17
2.12 Conclusion . . . 17

3 Voxel Storage and Visualization 19

3.1 Introduction to Voxel Storage . . . 19
3.2 Interaction with voxels . . . 19
3.3 Voxel grid . . . 20
3.3.1 Flat Array . . . 21
3.3.2 Chunks . . . 22
3.3.3 Flat array indexing . . . 22
3.3.4 Morton Codes (Z-order Curves) . . . 24
3.3.5 Criticism of Voxel Grids . . . 24
3.4 Sparse Voxel Octree . . . 24
3.5 Generation of voxel models . . . 25
3.5.1 Noise . . . 26
3.5.2 Procedural generation from noise . . . 27
3.6 Level of Detail . . . 28
3.7 Compression . . . 28
3.7.1 Run-length Encoding . . . 28
3.7.2 Interval tree . . . 29
3.7.3 Other . . . 29
3.8 Overview of visualization . . . 30
3.9 Surface and mesh definition . . . 30
3.10 Ray rendering overview . . . 30

3.11 Octree Rendering 31

4 Voxel meshing algorithms 33
4.1 Smooth surface meshing 33
4.1.1 Marching cubes 33
4.1.2 Surface nets 34
4.1.3 Dual countoring 35
4.2 Cubic Quad-based Voxel Meshing 36
4.2.1 T-Junction problem 36
4.2.2 Long-thin triangle problem 38
4.2.3 Simple mesher 38
4.2.4 Culled mesher 39
4.2.5 Greedy meshing 40
4.2.6 Naive greedy meshing 42
4.2.7 Sector's Edge run meshing 42
4.2.8 Binary greedy meshing 43
4.2.9 Optimal greedy meshing 44
4.2.10 Monotone meshing 45
4.2.11 Poly2Tri Meshing 45

5 Unreal Engine 47
5.1 Support of voxel models 48
5.2 Project 50
5.3 Assets 50
5.4 Levels 51
5.5 Blueprints 52
5.5.1 C++ 53
5.5.2 Objects 54
5.5.3 Actors 54
5.5.4 Pawns 55
5.5.5 GameMode and player inputs 56
5.6 Static Mesh 56
5.7 Materials 57
5.8 Physics 57
5.9 Raycasting 57
5.10 Collisions 58
5.11 Procedural Mesh Component 58
5.12 Custom logs 59
5.13 Unreal Insights 60
5.14 StartFPSChart 61
5.15 Nanite 62
5.16 Unreal Engine Plugins 63
5.17 Unreal Engine Modules 64
5.18 Fab 65
5.19 Fast Noise Generator 65
5.19.1 RealTimeMesh Component 66

6 Competitive Algorithms in Unreal Engine 69

6.1 Paid . . . 70
6.2 Porism DIMs World Generator . . . 71
6.3 Voxel Plugin . . . 72
6.3.1 Voxel meshing in Voxel Plugin . . . 73
6.3.2 Voxel Plugin Wrapper . . . 77
6.4 Conclusion . . . 78

7 Implementation of Run Directional Voxel Meshing 80
7.1 Voxel Faces . . . 81
7.1.1 Merging of voxel faces . . . 84
7.2 Voxel Grid variant . . . 84
7.2.1 Voxel Meshing . . . 85
7.2.2 Optimization . . . 88
7.2.3 Results . . . 89
7.3 RLE-compressed variant . . . 90
7.3.1 Voxel Meshing . . . 91
7.3.2 Optimization . . . 92
7.3.3 Interaction . . . 93
7.3.4 Results . . . 99
7.4 Implementation of Voxel Storage . . . 100
7.5 Voxel Generators . . . 100
7.6 Unreal Engine integration . . . 101
7.7 Conclusion . . . 102

8 Performance profiling 103
8.1 Scenario . . . 103
8.2 Dataset . . . 105
8.3 Profiling methodology . . . 106
8.3.1 Tools and Procedure . . . 106
8.3.2 Data Processing . . . 107
8.3.3 Scope of Measurements . . . 107
8.3.4 System Specifications . . . 107
8.4 Memory . . . 107
8.5 FPS and Vertices . . . 113
8.6 Voxel Meshing Performance Profiling . . . 117
8.6.1 Phase Breakdown . . . 118
8.6.2 Timing events . . . 121
8.6.3 Chunk Scalability . . . 123
8.7 Performance Profiling results . . . 124

9 Conclusion 127
9.1 Future optimization . . . 127
9.2 Future Work . . . 128
9.3 Closing Words . . . 129

Bibliography 130

List of Figures

2.1 Example of model remeshing taken from Polygon Mesh Processing book [10]. 10
2.2 Examples of voxel-based games 12
2.3 Examples of different surface representations generated using Voxel plugin (6.3). 14
2.4 Types of quad meshes. 15
2.5 Voxel model modification. 17

3.1 3D representation of flat array structure. 21
3.2 Illustration of chunk sparsity. 22
3.3 Voxel coordinates in voxel array transformed to a voxel grid. 23
3.4 Representation of octree sequence. 24
3.5 3D representation of SVO structure. 25
3.6 Noise images generated using FastNoise library [69]. 27
3.7 Cubic voxel terrain generated using noise. 27
3.8 Floating islands generated using multiple noise layers 28

4.1 Screenshots of a T-Junction situations taken from [48]. 37
4.2 Screenshot of a T-Junction seam taken from [7]. 37
4.3 Basic chunk mesh generated using modified version of 7.2. 39
4.4 Culled chunk mesh generated using modified version of 7.2. 39
4.5 Greedy chunk mesh generated using 7.2. 40
4.6 Example of naive greedy meshing not finding optimal solution taken from a blog [7]. 42
4.7 Screenshot of face merging meshing taken from [71]. 43
4.8 Illustration of finding optimal greedy mesh taken from [8]. 44
4.9 Screenshot of optimal greedy meshing taken from [8]. 45
4.10 Example of monotone meshing error [7]. 45
4.11 Example, where T-Junctions may occur when using this method taken from [9]. 46

5.1 Unreal Engine logo. 47
5.2 Rendering of a large volumetric dataset using Heterogeneous Volume rendering in UE 5.4 taken from Schlüter et al. [74]. 48
5.3 Screenshot of Content Browser. 50
5.4 Asset examples in the Unreal Engine content browser. 50
5.5 Screenshot of UE5 Level Editor interface. 51
5.6 A Blueprint example in the Blueprint Editor. 52
5.7 Project screenshot in Rider IDE. 53
5.8 Example of actors with components. 55

5.9 Example of Output log in the editor. 59
5.10 Screenshot of Unreal Insights. 60
5.11 Editor widget. 60
5.12 Example of Nanite activated on an Actor with different views. Left is a mesh generated from Nanite. Right is the original Static Mesh. Image was taken from [21, Nanite Virtualized Geometry]. 62
5.13 Example of module usage inside Unreal Engine plugin. 64
5.14 Fab website screenshot. 65

6.1 Fab screenshot with a voxel filter. 69
6.2 Fab screenshots of paid algorithms 70
6.3 Voxel-based environment generated by Voxel Plugin in a showcase level. . . 73
6.4 Voxel-based environment generated by Voxel Plugin in a showcase level with highlighted LOD transitions. 73
6.5 Configuration of an VoxelWorld actor. 74
6.6 Screenshots of Marching cubes surface generated using Voxel Plugin. 75
6.7 Screenshots of Surface nets generated using Voxel Plugin. 76
6.8 Cubic surface wireframe generated by Voxel Plugin. 76
6.9 Figure showing my voxel model meshed using Voxel Plugin. Left is a voxel model generated using my implementation of voxel meshing. Right is a voxel model generated using Voxel Plugin. 78

7.1 Sphere mesh generated using RDM. 80
7.2 Unreal Engine left-handed coordinates image taken from Unreal Engine documentation [21, Coordinate System and Spaces]. 81
7.3 Voxel faces. 82
7.4 Voxel boundary representation using faces. 83
7.5 Named vertices in a voxel face. 84
7.6 Examples of how vertices may be compared and merged together. 84
7.7 Grid–based RDM traversal and merging. 86
7.8 Screenshot taken during the initial process of face generation and merging phase in Grid–based variant, visualized via a modified version of RDM. . . 86
7.9 Screenshot taken during the initial process of face generation and merging phase in Grid–based variant, visualized via a modified version of RDM. . . 87
7.10 Final mesh after greedy mesh optimization. 88
7.11 Sphere wireframe generated using Grid–based RDM. 89
7.12 Illustration of RLE sequences in RLE–based variant. 90
7.13 Illustration of RLE sequence traversal. It is similar to the voxel grid illustration, but the voxels have become interval-like sequences. 91
7.14 Screenshot taken during the initial process of face generation in RLE–based variant, visualized via a modified version of RDM. 92
7.15 Final mesh generated from RLE–based variant. 92
7.16 Diagram that shows how voxel change is applied to the first run. 95
7.17 Diagram that shows how voxel change is detected at the start of a new run. 96
7.18 Diagram that shows how voxel change is applied when detected mid run. . 97
7.19 Diagram that shows how voxel change is detected when the change is at the start of a new axis. 98
7.20 Sphere wireframe generated using RLE–based RDM. 99

7.21 Screenshot of Unreal Engine Data table with types of voxels 100
7.22 Screenshot from Blueprint on how interaction event is passed from Unreal Engine into a Chunk Spawner 101
7.23 Diagram of implementation with Unreal Engine. 102

8.1 Screenshot of a scenario template. 104
8.2 Screenshot of dataset placed into a scenario scene. 105
8.3 Screenshot of Voxel Model in Scenario 17. 106
8.4 Persistent memory of voxel models in group 1. 108
8.5 Ratio of opaque and transparent voxels in Voxel Models in group 1. 109
8.6 Persistent memory of voxel models in group 2. 110
8.7 Ratio of opaque and empty voxels in Voxel Models in group 2. 110
8.8 Persistent memory of voxel models in group 3. 111
8.9 Ratio of opaque and transparent voxels in Voxel Models in group 3. 111
8.10 Strip voxel models in group 3. 112
8.11 Vertices in Voxel Grid Scenarios. 113
8.12 Vertices in Voxel Plugin Scenarios. 114
8.13 Vertices in Voxel Grid Scenarios. 115
8.14 FPS performance in selected scenarios. 116
8.15 Voxel Plugin LOD artifact in voxel model taken from scenario 17. Left side is a voxel model generated using Grid-based RDM on the right side is a voxel model generated using Voxel Plugin 117
8.16 Phases Voxel Meshing algorithms in Group 1 118
8.17 Phases in scenario 17. 119
8.18 Phases Voxel Meshing algorithms in Group 2. 120
8.19 Phases Voxel Meshing algorithms in Group 3. 120
8.20 Voxel Meshing Speed of Run Directional Meshing from RLE in Group 2. . . 121
8.21 Voxel Meshing Speed of Run Directional Meshing from RLE in Group 3. . . 122
8.22 Screenshot of all voxel meshing algorithms in Scenario 5 123
8.23 Influence of different chunk sizes on Voxel Plugin across scenarios. 123
8.24 Influence of different chunk sizes on Run Directional Meshing from Grid across scenarios. 123
8.25 Influence of different chunk sizes on Run Directional Meshing from RLE across scenarios. 124
8.26 Efficiency of Voxel Meshing Algorithms. 125

Chapter 1

Introduction

Voxel-based worlds, which are made from small cube-like blocks, have become a popular way to create 3D environments, especially in voxel fan communities which is described in chapter 2. This structure makes it easy to generate terrain procedurally, edit it in real time, and simulate complex environments. However, modern game engines like Unreal Engine are built to work with triangle-based models, called meshes. Because of this, voxel models need to be converted into meshes before they can be used or displayed in the engine. This process is called meshing in voxel engines.

This thesis focuses on solving that problem in Unreal Engine. The objective was to study how voxel models are handled, review existing meshing techniques, and develop a new method that could efficiently turn compressed voxel data into triangle meshes. The method also had to support real-time editing so that changes to any voxel model would immediately update the mesh.

To meet these goals, a new algorithm called Run Directional Meshing was created and implemented as a plugin for Unreal Engine. The algorithm serves as a functional prototype that proves this conversion from a compressed representation is possible. The compressed representation also reduces persistent memory storage of a voxel model.

The plugin was built using C++ and fully integrated into Unreal Engine. It supports procedural terrain generation, which is used to substitute complex voxel models, real-time editing such as adding or removing blocks, and instant mesh updates during gameplay. Performance tests were conducted to compare this method with competitive algorithms described in chapter 6. The tests profile meshing speed, frame rate, memory use, and the number of vertices generated. Results can be seen in chapter 8.

In conclusion, this thesis presents a functional Unreal Engine plugin that implements a novel voxel meshing algorithm termed Run Directional Meshing. The plugin serves as a prototype demonstrating the feasibility of converting compressed voxel data into triangle meshes in real time, while preserving full interactivity and editability of the voxel environment. Its support for user interaction and built-in performance profiling framework establishes it as a viable proof of concept. Moreover, the plugin provides a robust foundation for future work and development in voxel-based applications, offering a practical platform for evaluating and extending real-time meshing techniques in interactive environments.

Chapter 2

Motivation

Voxel-based representations are increasingly relevant in real-time applications due to their simplicity, flexibility, and support for procedural generation. As Mileff and Dudra [63] note, voxels store geometry and visual data directly in a 3D grid, simplifying content creation. However, rendering large voxel datasets remains expensive and often lacks support. While GPU raycasting algorithms can be used as described in section 3.10, it is poorly suited for interactions, especially with physics-based systems in game engines.

These limitations pose challenges for real-time, interactive applications, especially in environments like Unreal Engine, which rely on mesh-based rendering and physics. This thesis addresses these challenges by investigating how voxel data can be efficiently converted into polygonal meshes while preserving interactivity and trying to minimize performance overhead.

The goal is to support real-time interaction and ensure compatibility with engine systems and similar frameworks. The following chapter details these challenges and outlines the goals that drive the proposed approach.

2.1 Voxel

Thesis Gigavoxels [15] states:

> *„The name voxel comes from "volumetric elements„ and it represents the generalization in 3D of the pixels. Voxels represent the traditional way to store volumetric data. They are organized in axis-aligned grids subdividing and structuring space regularly."*

According to Real-Time Rendering [1], a voxel (volume element) is the 3D equivalent of a pixel or texel, representing a cube of space in a regular 3D grid. Its position is determined by its index, and it can store various values such as material, color, or flow data. Voxels are useful for representing volumetric data like terrain or smoke and can integrate well with mesh-based scenes.

2.2 Voxel models

Voxel models are represented by voxel data stored in a voxel grid. These models have unique properties and allow operations on the entire object rather than just its surface. For example, the volume of a voxel model can be calculated as the sum of its voxels [1].

A model represented by voxels can be any geometry of an object. The regularity of the voxel grid allows simulations of, for example, erosion or clouds using cellular automata. Due to the finite number of voxels in the grid, the tensile strength of an object can be determined. A new model can be sculpted from the original by removing or altering the material of the voxels. Creating a voxel model from a polygonal one can be done by placing the polygonal model into the voxel grid. Sampling is done by determining which voxels overlap the model [1].

As stated by Lv et al. [53], Voxel models are digital representations of 3D objects using a regular grid of volumetric elements called voxels—the three-dimensional counterpart to pixels. Each voxel implicitly encodes its spatial position via its index in a 3D array and may store properties such as material type, density, or color.

Unlike surface-based representations, voxel models describe the entire volume of an object, enabling operations like internal editing, volume computation, and procedural modification. In this work, voxel models serve as input for mesh generation algorithms, with an emphasis on their structured layout rather than realism or detailed physical accuracy.

In this case, the voxel model serves as a storage structure for storing voxels in a way that represents the original geometry.

2.3 Meshing

Meshing [67], or mesh generation, is a fundamental process in engineering simulations that divides complex geometries into small, discrete elements to enable numerical solutions of partial differential equations (PDEs). These elements include triangles and quadrilaterals for surfaces, and tetrahedra, triangular prisms, quadrilateral pyramids, and hexahedra for volumes. Meshing is essential in Finite Element Analysis (FEA) [51] and Computational Fluid Dynamics (CFD) [79], where analytical solutions are impractical for real-world geometries. The main meshing techniques are structured [12], unstructured [78], and adaptive. Structured meshing techniques use a regular grid layout with evenly shaped elements, which are efficient for simple shapes and commonly used in simulations like heat transfer and structural mechanics. Unstructured meshing techniques use irregularly shaped elements that fit complex geometries more accurately, such as those found in car bodies or airflow simulations, though they require more computational resources. Adaptive methods such as Adaptive Multilinear Meshes (AMM) [6] adjust the mesh dynamically during simulation to refine critical areas and simplify less important regions. Accurate simulations depend on mesh quality, including element shape, size, and alignment. Pre-processing steps like CAD cleaning and surface wrapping address geometry issues, while post-processing includes correcting bad elements and applying prism layers near solid surfaces. Mesh sensitivity and grid independence studies are used to achieve a balance between accuracy and computational cost.

2.4 Remeshing and mesh repair

Remeshing [10] is the process of altering a mesh's connectivity or geometry to enhance its quality, improving factors like regularity, uniformity, and feature alignment, especially when the original mesh is inaccurate or low-quality. It is essential for tasks such as simulation, parameterization, and simplification, and is often performed iteratively. Techniques generally fall into triangle-based methods, which adjust triangle shape via edge flips and vertex

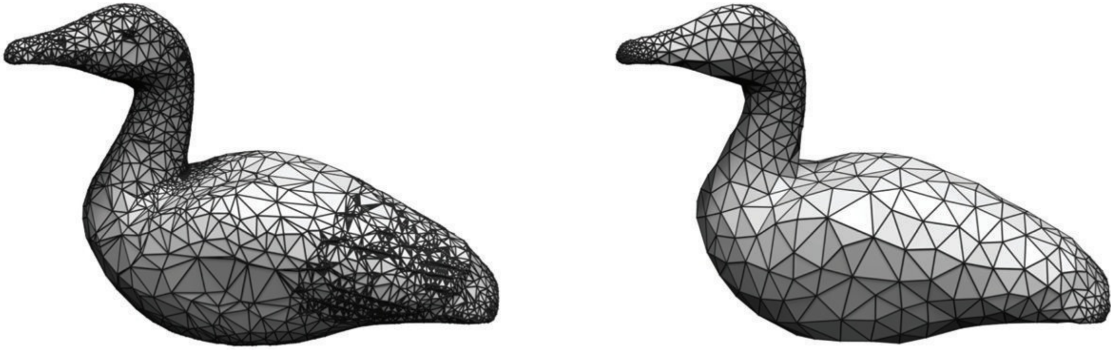


Figure 2.1: Example of model remeshing taken from Polygon Mesh Processing book [10].

relocations, and quad-dominant methods, which generate primarily quadrilateral meshes for greater regularity and subdivision compatibility. Voxel-based remeshing is particularly useful for mesh repair and implicit remeshing [47], transforming noisy or non-manifold meshes into implicit representations like signed distance fields, from which clean, watertight [83], and manifold surfaces can be reconstructed using algorithms such as marching cubes [52] or dual contouring [35]. Guan et al. [30] introduce a voxelization-based approach to generate quadrilateral meshes directly from point clouds without relying on triangle-based methods, producing semi-regular meshes through point remapping and regularization. Similarly, Lv et al. [53] propose a voxel-based pipeline for reconstructing triangle meshes from point clouds by incorporating local resampling, feature detection, and geodesic metrics to guide isotropic remeshing with edge operations that preserve geometric features. A recurring theme in these methods is isosurface extraction, discussed in the following section.

2.5 Isosurface extraction

Isosurface extraction is a key method for converting voxel-based data, such as CT or MRI scans, into surface meshes [95]. Algorithms like Marching Cubes [52] and Surface Nets [29] extract triangle meshes by identifying surface boundaries within voxel grids, forming the basis for many voxel-to-mesh workflows. These methods aim to preserve geometric and topological integrity, generating clean, manifold meshes directly from binary or labeled volumes.

The main motivation behind isosurface extraction is accurate 3D reconstruction, particularly in medical imaging, where it supports diagnosis, surgical planning, and modeling of patient-specific anatomy [29]. Enhancements like gradient-based mesh refinement improve surface quality and reduce complexity [11]. While classical techniques remain widely used, neural approaches are emerging, offering differentiable, resolution-independent alternatives for high-level geometry processing [57].

2.6 Neural Representations and Hybrid Approaches

Recent advancements in voxel-based meshing increasingly leverage neural networks to overcome the limitations of traditional surface extraction methods. Voxel2Mesh [90] introduces a deep learning approach that deforms a template mesh from voxel input, producing topo-

logically consistent surfaces suitable for high-quality medical imaging. VMesh [31] uses neural networks to learn signed distance fields, enabling resolution-independent mesh extraction via differentiable rasterization, which supports advanced rendering and UV mapping. More recently, Vosh [94] presents a hybrid voxel-mesh method that dynamically adapts neural surface representations into mesh geometry for real-time rendering on GPUs, drawing inspiration from NeRF techniques. These methods exemplify a shift toward neural surface modeling, offering improved visual fidelity. Wu et al. [92] address real-time volume rendering through multi-resolution hash grids and CUDA, yet their system remains limited to passive exploration. In the future, integration of such conversion into Unreal Engine could be considered. However, such integration of neural networks into Unreal Engine is out of scope for this thesis, but it represent possible future direction of this problematic.

2.7 Voxel engines

The oldest mention of the term voxel engines comes from a game called Outcast[1]. The engine combined polygons to render objects and height maps to render terrain. The renderer interpreted a 2D map of height values to calculate the boundary between the air and the ground in the scene as made up of voxels. However, according to 80Level blog [81], this was not truly a voxel-based approach as there‘s no volumetric data set in use and the game does not model three-dimensional volumes of voxels. There was an attempt to make hardware-accelerated voxel engines as presented by Verve [40]. Another mention of voxel engines can be found in a chapter of GigaVoxels thesis [15, 1.2 Other applications and usage of voxel representations], where voxel engines are mentioned to render special effects from large voxel grids. The main idea of voxel engines seems to be to render voxels.

The revolution in voxel engines came after the release of the Minecraft[2] game. The revolution is stated in the 0FPS blog [55], by Dusterwald [18], by Arnebäck et al. [4], and Cambronero [13]. The revolution is the idea to convert voxels into polygons and use a GPU rasterization pipeline to render the surface of voxels. Several online tutorials[3] exist regarding the development of voxel engines.

After this, voxel engines split into two branches:

1. Directly rendering voxel representation on GPU. This is the case of GigaVoxels thesis [15] and projects like Atomontage[4] and visual effects.

2. Real-time conversion of voxels into a polygonal mesh representation performed on CPU. This is usually the case with voxel engines used in voxel-based games [54] such as Minecraft[5] or Luanti[6]. A figure 2.2 shows different examples of voxel-based games.

Discussions about the terminology of voxel engines were made across several community forums[7]. This may result in hybrid approaches like in Max Slater blog [60]. But it became

[1] Link to an Archive of the Outcast blog: https://web.archive.org/web/20060507235618/http://www.outcast-thegame.com/tech/paradise.htm

[2] Link to official Minecraft website: https://www.minecraft.net/en-us

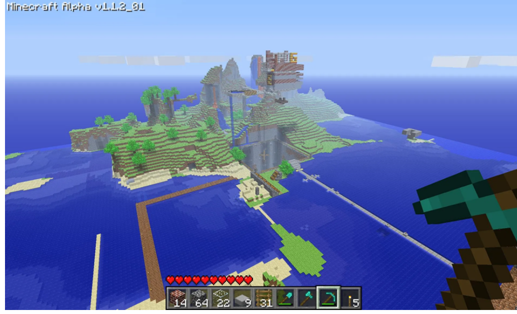
[3] Link to example of an online tutorial on voxel engine: https://sites.google.com/site/letsmakeavoxelengine/

[4] Link to Atomontage official website: https://www.atomontage.com/

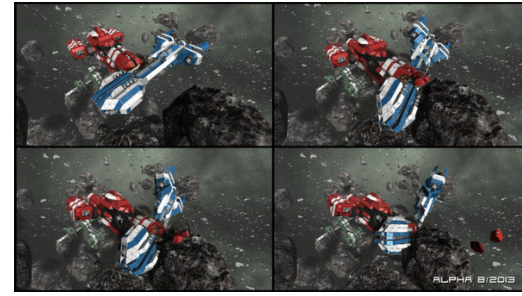
[5] Link to Minecraft official website: https://www.minecraft.net/en-us

[6] Link to Luanti official website: https://www.luanti.org/

[7] Link to stack exchange forum discussion: https://gamedev.stackexchange.com/questions/11976/is-a-voxel-engine-appropriate-for-a-minecraft-like-game



(a) Minecraft screenshot taken from a review [62].



(b) Space Engineers collision image taken from a Stea

(c) ASTRONEER image taken from a Steam page [80].

(d) Teardown image taken from a Steam page

Figure 2.2: Examples of voxel-based games

the voxel community consensus to include this conversion as a part of voxel engines. This essentially created a new branch for voxel engines. There are several discussions on */r/VoxelGameDev* subreddit[8] that discuss the terminology of this new branch of voxel engines. One discussion[9] centers on the definition of voxel games. Another discussion[10] clarifies that voxels represent 3D space, and how they are rendered can vary, including techniques like raycasting or polygons. While many voxel-based games use polygons for efficiency, true voxel rendering can bypass polygons entirely. Most voxel engines today convert voxel data into polygons for speed. Despite debates, the voxel community now accepts this conversion as part of voxel engine design. This history is briefly summarized in the 80Level blog [81]. A common theme of a voxel engine is to provide a framework for managing voxels. This includes processes required to render the voxels and the ability to manipulate voxel models.

The first branch can be seen, for example, in the paper about Poxels [64], a method for high-performance voxel engine rendering. It uses geometry shaders to generate visible voxel faces, with stream-out caching and data compression (run-length encoding, quantization) to minimize memory use. Advanced culling techniques reduce processing load. Unlike raycasting, Poxels fully exploits GPU rasterization hardware, enabling faster and higher-quality rendering of large voxel environments.

Another paper advances voxel engines by introducing real-time fluid simulation into voxel engines [93] by running a Navier–Stokes solver fully on the GPU. It improves typical voxel fluid behavior, avoids costly CPU-GPU memory transfers, and leverages geometry shaders for efficient dynamic voxel rendering. Built on the Poxels framework, this ap-

[8]Link to VoxelGameDev subreddit: https://www.reddit.com/r/VoxelGameDev/

[9]Link to VoxelGameDev subreddit discussion about voxel game definition: https://www.reddit.com/r/VoxelGameDev/comments/lhhe70/voxel_terminology_these_are_not_the_voxels_you/

[10]Link to VoxelGameDev subreddit discussion about voxel engine definition: https://www.reddit.com/r/VoxelGameDev/comments/tpmtn4/what_is_a_voxel_based_game/?tl=pt-pt

proach dramatically boosts performance and suggests extensions like obstacle interaction and multi-chunk simulation. This paper also emphasises the resource intensity of voxel engines.

The second branch became popular in a blog writers community after the release of Mikola Lysenko's 0FPS blog [55]. In this blog, the conversion of voxels to a polygonal mesh is called **meshing in voxel engines**. This started community-driven development[11] of these voxel engines across community blogs, as seen in John Lin's blog [50]. Blog post examples of code implementation [85, 2]. The revolutionary idea is to introduce meshing and remeshing to voxel engines. Similarly, the Real-Time Rendering book [1] describes a technique that transforms a voxel grid into a polygonal representation, rendering the model as a polygon mesh. While rendering polygonal meshes is fast, the conversion process introduces additional costs during mesh creation. Because this bachelor's thesis focuses on this second branch, most of the sources for this bachelor's thesis are community-driven blogs. Resulting polygonal meshes can be rendered in a scene and create complex colliders (5.10).

An academic representative example of the second category is the VoxBox engine, developed by Dusterwald [18] in Unity3D for procedural generation of voxel worlds and architectural elements like castles. VoxBox utilizes a chunk-based design with 16^3 voxel chunks, using flat arrays and hashmaps to improve performance while preserving spatial resolution. It also implements a layered voxel system, separating terrain and structure layers for non-destructive editing. Procedural generation is driven by coherent noise for terrain and rule-based placement for structures, with all heavy operations—including lighting and meshing—performed in background threads to maintain responsiveness. This architecture highlights modern trends in voxel engine development: separation of data and rendering, thread-based performance optimization, and support for interactive, procedurally generated environments.

Even if there is a lack of academic articles, many students wrote works or theses about this topic, often implementing their own solution. One such example can be a custom-made game engine focused on voxel support made by Wilder [91]. For example, Bloxel [4] provides a complete overview on how to create a voxel-based game. Similarly, DigBuild [13] provides a framework for the creation of voxel games.

2.8 Voxel Meshing

In some blogs [77, 61], the term voxel meshing refers to converting a 3D voxel grid into a mesh of quads or triangles for rendering in games. Voxels represent the smallest space unit in voxel-based games, and the mesher converts them into meshes. Across blogs, this process is called meshing in voxel engines, voxel generation, or voxel meshing. Voxel mesher is then called a specific implementation of the voxel meshing algorithm. This can be seen reflected in official Godot[12] Documentation [13] or in Voxel Plugin (6.3).

The use and meaning of the term are not fully unified across different fields. In Cinema 4D plugin[14], it is used to convert meshes into cubic meshes. In biomechanics and medical imaging, voxel meshing typically describes the direct conversion of volumetric scan data, such as CT or MRI images, into finite element models, often requiring additional smoothing

[11]Link community-driven development YT video: https://www.youtube.com/watch?v=CJ94gOzKqsM
[12]Link to Godot official website: https://godotengine.org/
[13]Link to Godot's VoxelMesher: https://voxel-tools.readthedocs.io/en/latest/api/VoxelMesher/
[14]Link to VoxelMesher plugin for Cinema 4D: https://www.rocketlasso.com/voxelmesherinfo

techniques to correct jagged surfaces [89]. In composite materials modeling, voxel meshing focuses more on automating the assignment of material phases within complex fiber structures, prioritizing computational efficiency over geometric precision [39, 24]. In stress and damage analysis, voxel meshes are often criticized for causing artificial stress concentrations due to their stepped surfaces, necessitating special treatments like stress smoothing or local refinement [17]. Although the basic concept of voxel meshing—dividing space into regular elements—is consistent, its specific application and interpretation vary significantly between disciplines, reflecting a lack of a fully standardized definition. This is also reflected in Altair [15] documentation[16].

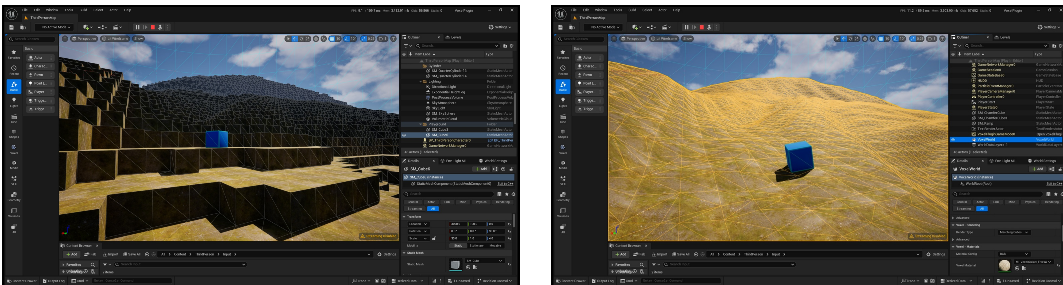
In this thesis, the term voxel meshing is used to specifically denote the process of converting a voxel model into a polygonal mesh within the context of voxel engines. Because *meshing* is a broader term that generally refers to mesh generation, **voxel meshing** here refers exclusively to the *conversion process of voxel data into a mesh*. In contrast, voxelization (see Section 3.5) refers to the sampling of a continuous space using discrete voxels.

The conversion from voxels to polygonal geometry is a necessary preprocessing step before rendering, especially in engines or frameworks such as Unreal Engine (see Section 5), which focus on the rendering of polygonal meshes. This thesis proposes a novel voxel meshing algorithm designed to operate efficiently within such voxel-based environments.

For the sake of simplicity, the term *meshing* algorithm may occasionally be used interchangeably in this work. This is particularly true when referring to voxel meshing algorithms, where simplified naming conventions are preferred. Some algorithms may be referred to simply as meshing and implemented under a common base class called `VoxelMesher`, in the source code. However, to ensure consistency across the implementation, all classes and directories involved in the voxel meshing process will adopt the standardized name `VoxelMesher`.

From this point onward, the term voxel meshing will be used consistently throughout this bachelor's thesis to refer to the voxel-to-mesh conversion process.

2.9 Visual goals



(a) Cubic surface. (b) Smooth surface.

Figure 2.3: Examples of different surface representations generated using Voxel plugin (6.3).

Many of the previously mentioned voxel engines prioritize generating surfaces that better represent voxel surface compared to cubic type because the priority is to produce a smooth

[15] Link to official Altair website: https://altair.com/

[16] Link to Altair documentation page: https://2021.help.altair.com/2021/hwdesktop/hm/topics/pre_processing/meshing/meshing_voxel_c.htm#concept_byq_bcl_yfb

surface representation as can be seen in figure 2.3. Alternative surface types can be achieved by adapting algorithms originally developed for remeshing and applying them to real-time voxel meshing. Notable examples of such approaches are documented in the 0FPS blog [58]. The selection of a particular algorithm is therefore often influenced by the visual objectives of the voxel engine in question.

The primary objective of this thesis, however, is the efficient conversion of voxel models rather than the rendering of complex, smooth surfaces. The algorithms introduced can generate only cubic surfaces, and not produce high-fidelity or organically shaped meshes. Although surface reconstruction techniques such as Marching Cubes [52] can yield detailed and smooth topography, this thesis does not propose any enhancements to this class of algorithms, as such efforts fall outside the scope of the current work. Instead, the methods developed are evaluated primarily on their performance and impact on real-time processing efficiency, rather than visual fidelity.

In accordance with these visual objectives, only algorithms that support cubic mesh generation are used in this thesis and for comparison. This ensures a consistent basis for evaluating performance metrics. Consequently, the visual goal is to maintain a uniform and consistent cubic surface representation across all test cases and implementations.

2.10 Performance Goals

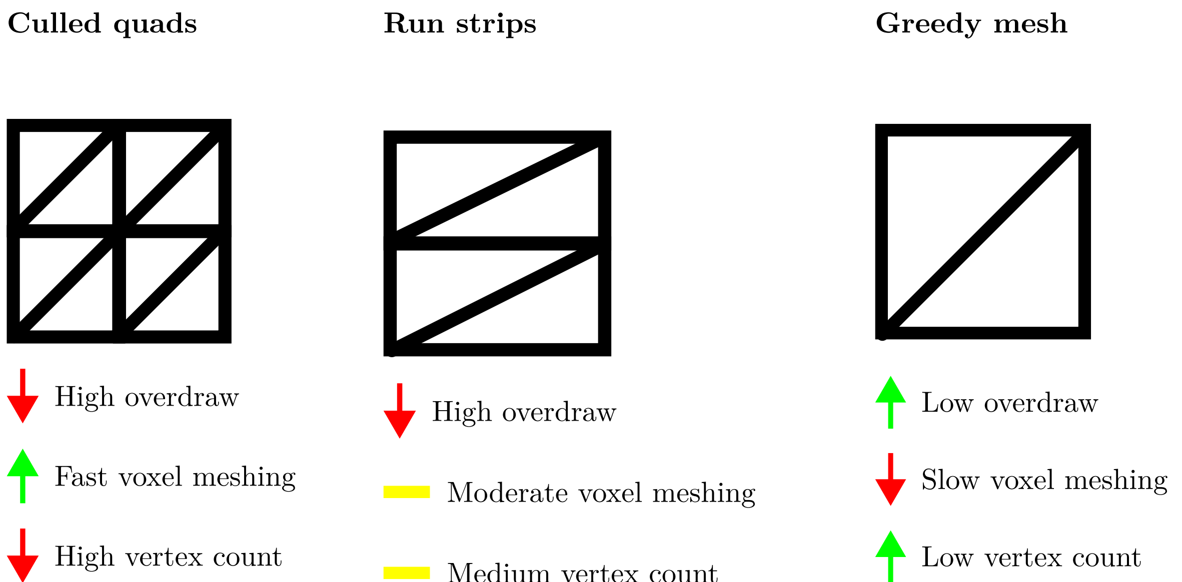


Figure 2.4: Types of quad meshes.

For cubic mesh types, I identified three distinct categories of quad-based meshing techniques, each of which is examined in detail throughout this thesis. These are: (1) culled quad meshes, (2) run strip meshes, and (3) greedy-meshed meshes.

The most commonly used quad mesh types in voxel engines are illustrated in Figure 2.5, along with their respective impacts on performance. This figure provides an overview, while the detailed characteristics and trade-offs of each mesh type are discussed in the corresponding sections of the thesis. As shown, when implemented efficiently, the greedy meshing technique offers the highest performance benefits among the three.

The performance implications of each mesh type, as depicted in Figure 2.5, are further analyzed in Chapter 4. This thesis specifically focuses on minimizing iteration overhead during voxel meshing while maintaining interactive performance, guided by well-established performance criteria.

Based on this established performance criteria:

- **Smaller number of triangles improve rendering speed**: Reducing triangle counts enhances rendering speed and reduce memory use.
- **Low voxel meshing latency is critical**: Delays exceeding two frames disrupt real-time interactions, necessitating rapid mesh updates.

Therefore, the objective is to develop an algorithm that produces compact, optimized meshes with minimal latency, ensuring smooth interaction. Furthermore, voxel surface density and modifications are stored within the voxel model, allowing real-time updates to be reflected in the rendered mesh. The hardware this algorithm aims for are devices able to support nanite (5.15).

2.10.1 Performance Factors

Achieving efficient voxel-to-mesh conversion requires optimizing several performance-critical aspects. The primary observed bottleneck in voxel-to-mesh conversion algorithms is the iterative traversal of large voxel datasets during voxel meshing. Excessive iteration significantly impacts conversion speed and overall system performance. A key optimization strategy is to reduce voxel dataset size through compression, reducing iteration by a compact representation, thereby improving performance in real-time applications.

- **Reducing iteration overhead**: Minimizing the number of iterations required for voxel meshing accelerates processing and decreases computational load.
- **Balancing voxel meshing speed vs. minimal quad generation**: The conversion algorithm is faster the more unnecessary quads it leaves and slower the more it tries to reduce the number of quads. The goal is to balance the number of generated quads so that the conversion speed does not decrease significantly while ensuring the number of quads generated is small enough for subsequent rendering but large enough to represent the model accurately. Additional quads may also cause overdraw.
- **Maintaining real-time interactivity**: Ensuring minimal voxel meshing latency for seamless user interaction.
- **Algorithmic scalability**: The voxel meshing process should efficiently scale with increasing voxel complexity while maintaining low computational costs.
- **Considering voxel sparsity**: The density of voxels within a grid significantly impacts voxel meshing performance. Denser models enable faster voxel meshing with fewer triangles, while sparse distributions lead to fragmented geometry and increased computational overhead.
- **Supporting volumetric interaction**: Enable real-time volumetric changes to the solid opaque surface of a mesh by dynamically updating voxel model density and structure. Dynamic update of BREP.

Since Unreal Engine already manages fundamental game engine tasks such as rendering, physics, and collision detection, this thesis focuses on refining the voxel meshing process to minimize performance bottlenecks while maximizing responsiveness.

2.11 Interactivity Goals

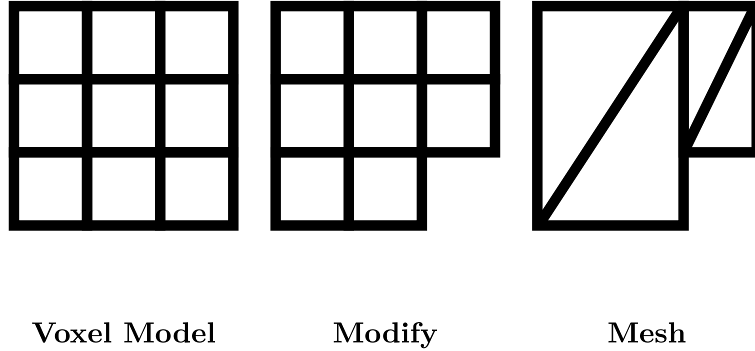


Figure 2.5: Voxel model modification.

Real-time voxel interactions require immediate visual updates in the rendered scene. Since Unreal Engine does not natively support collisions for voxel models, all interactions must be translated into polygonal representations. The conversion process follows steps:

1. Modifying the voxel model.
2. Converting the modified model into a polygonal mesh and render it.

This transformation must be near-instantaneous for an optimal user experience, reinforcing the need for an efficient voxel meshing algorithm. While interactivity may be much broader, like, for example, the introduction of cellular automata to voxel grids, it is out of the scope of this thesis, and interaction for this purpose is limited only to changing voxel material. Still, more complex *interaction* could follow the same steps. Another goal is to achieve destructive interaction, meaning volumetrically defining the geometry of an object.

Interaction is considered surface volume deformation, sculpting, and density changes.

A notable example of an interactive voxel editing system is VoxMorph [25]. It is a CPU-based tool designed for high-resolution voxel grids, emphasizing real-time deformation through a 3-scale pipeline. Users manipulate a low-resolution polygonal control cage using an as-rigid-as-possible (ARAP) method, with changes propagated to a mid-resolution visualization envelope via Green Coordinates. These intermediate representations enable immediate visual feedback during modeling. However, the deformation of the full-resolution voxel grid is deferred to a post-processing step, preserving performance while maintaining high output quality. Crucially, VoxMorph never converts voxels into polygon meshes for rendering or output, keeping the data in voxel form throughout.

2.12 Conclusion

This thesis aims to reduce iteration overhead and accelerate real-time voxel meshing by enabling direct processing of compressed voxel models. To this end, a new algorithm (chapter 7) has been developed for rendering opaque, cubic voxel surfaces, with an emphasis

on alignment with the stated goals. The main motivation is to improve voxel meshing performance in voxel engines.

A performance analysis (chapter 8) compares the proposed method with existing solutions by profiling performance and comparing results across different voxel models. The algorithm is designed as a prototype and proof of concept and demonstrates the feasibility of working directly with compressed voxel data while maintaining accurate voxel model representation and interactibility. The thesis also outlines potential directions for further optimization and development of this idea. The goal is eventual widespread integration into voxel engines and voxel-based games.

Also my personal note is that some chapter may have lower number of figures than expected. This is because my university limits size of submitted PDFs to 20MB and given scope of this thesis, I had to make reduction in some chapters to fit in this limit.

Chapter 3

Voxel Storage and Visualization

Efficient storage and visualization of voxel data are essential for real-time performance and interactivity in voxel-based systems. This chapter outlines the most common methods for representing voxel models, with an emphasis on data structures, compression techniques, and rendering strategies applicable to mesh-based environments. The focus is on formats and algorithms relevant to voxel meshing and integration with Unreal Engine, as used in this thesis.

3.1 Introduction to Voxel Storage

This section outlines common formats for storing voxel models and introduces the voxel formats used in this thesis. As stated by Arbore et al. [3], voxel storage formats define how volumetric data is organized and accessed, with each format offering different trade-offs between memory usage, performance, and complexity. Basic approaches like uniform grids are easy to implement but memory-intensive, while more advanced methods such as sparse voxel octrees (SVOs), sparse voxel DAGs (SVDAGs), and distance fields improve efficiency by exploiting sparsity and redundancy. Hybrid voxel formats combine these strategies to better adapt to the structure of the data. The following sections provide a brief overview of these formats and explain the choices made for this work. Only formats relevant to this thesis are presented; for additional information, refer to Hybrid Voxel Formats for Efficient Ray Tracing [3].

3.2 Interaction with voxels

Game engines such as Unity[1] and Unreal Engine (5) offer the possibility to support efficient real-time interaction with voxel-based models. The VoxSculpt [73] project illustrates this by implementing real-time volume sculpting in Unity, leveraging key engine features including ray casting for user interaction, collision detection for sculpting precision, GPU instancing for high-performance rendering, and multithreaded execution through Unity Jobs and the Burst compiler. These technologies collectively enable responsive and stable interaction within virtual reality environments. Notably, the system was built with medical imaging in mind, enabling interactive modification of voxel models derived from tomographic data such as CT and MRI scans. VoxSculpt demonstrates that game engines can support interactive editing of volumetric medical datasets, confirming their applicability beyond entertainment

[1]Unity official website: https://unity.com/

to domains such as medicine and scientific visualization. VoxSculpt highlights the advantages of integrating voxel systems into established game engines, particularly regarding interactivity, extensibility, and performance. This thesis may serve as a similar foundation, but within Unreal Engine.

VoxMorph [25] demonstrates an alternative approach to voxel interaction, a 3-scale freeform deformation system designed for high-resolution voxel grids. Unlike game engine-focused meshing algorithms that convert voxel data to polygonal meshes for real-time rendering and interaction, VoxMorph retains the voxel format throughout the entire deformation pipeline. It operates in three stages: user-driven deformation of a low-resolution control cage using as-rigid-as-possible (ARAP) methods; interactive deformation transfer to a mid-resolution mesh via Green Coordinates for smooth visual feedback; and a high-resolution post-process deformation applied to the voxel grid using a tetrahedral mesh and local linear interpolation. This approach enables high-fidelity editing of volumetric data, such as medical scans, with minimal visual artifacts and without immediate conversion to a mesh. While not designed for real-time physics or collision integration within game engines, VoxMorph exemplifies an alternative method for volumetric interaction, favoring deformation quality and voxel data preservation over interactive mesh reconstruction.

3.3 Voxel grid

Storing voxels in a voxel grid has high memory requirements. The memory usage grows with the resolution at which voxels are sampled, with a complexity of $O(n^3)$. To improve this, Voxel models are streamed around the player in games with nearly infinite world generation, like Minecraft. Each voxel contains a unique material identifier value. Each block can have its material or a polygonal representation. Data stored in voxel models exhibit coherence, meaning there is a high likelihood that nearby locations will contain the same or similar values. A large portion of the voxel grid can be empty. These empty areas are called sparse volumes. Coherence and sparsity allow for creating a compact representation [1].

A voxel grid is a structured three-dimensional array used to represent volumetric data in discrete, uniformly spaced elements known as voxels. Each voxel typically holds scalar or vector values that define the properties of the modeled volume at that location, such as density, color, or material type. Voxel grids are fundamental to a variety of applications, including 3D graphics, simulation, and volumetric rendering. In this thesis, voxel grids serve as the foundational structure for generating 3D meshes from voxel data. Denisova et al. [16] describe how a voxel grid can be created and used on the GPU.

In VoxSculpt [73], voxel grids are implemented as permanently dense 3D arrays. This design supports features like restoration tools, which require access to removed voxels for collision detection and interaction. Each voxel in the grid stores its index-space position, an 8-bit grayscale color, and a state flag that can be Visible, Solid, or Removed. The dense array is supplemented by sparse storage via hash maps to track visible voxels in active chunks, enhancing performance. To optimize GPU compatibility and cache performance, all voxel data is stored as 1D arrays, with index-to-position conversions calculated at runtime.

A voxel grid, where voxels are stored, can be imagined as a three-dimensional texture. Such data can be conceptualized in several ways [1].

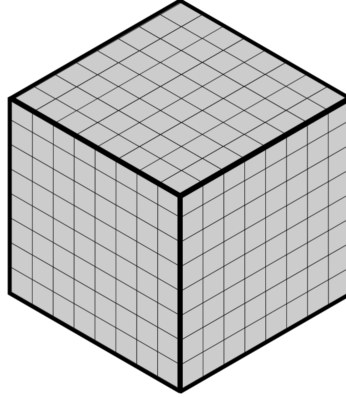


Figure 3.1: 3D representation of flat array structure.

3.3.1 Flat Array

A flat array model stores all voxels in a large fixed-size one-dimensional array. The position of the voxel is given by its index in the array. This type of storage has the advantage of constant-time read and write access for new voxels, but it consumes a vast amount of memory, and the list size is fixed in advance to allow index calculation. The position and size of each voxel are determined by its index [54]. Value in the array can be a material index that references its properties in a material table.

The importance of flat arrays as a voxel storage is also stated by Wilder [91].

Voxel grids can also be created from a collection of images, often used in medicine, where scanned X-ray images are stacked up. Similarly, mesh models can be sampled into slices, and the voxels within these slices are sampled in a voxel grid. If we want to voxelize models on modern graphics cards, it is possible to use, for example, a depth buffer as storage for individual voxel samples. Different types of voxelization vary based on how they determine whether a voxel is inside or outside the model [1].

Flat arrays will be used in this thesis.

3.3.2 Chunks

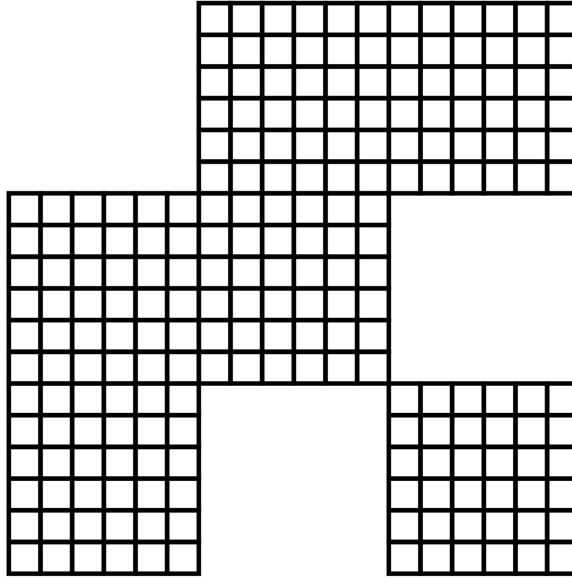


Figure 3.2: Illustration of chunk sparsity.

Chunks are uniformly sized, cubic regions of the voxel grid that enable efficient spatial partitioning and memory management. In VoxSculpt [73], chunks use a sparse data representation and are indexed via a 3D hash map based on world-space coordinates, allowing constant-time access and preserving spatial locality. The 0FPS blog [54] describes chunks as fixed-size flat arrays (k voxels) stored in a hash table, where a chunk's position in world space serves as its hash key. Voxel data is indexed within the flat array based on local position, allowing for sparse representation and lazy initialization—chunks are only instantiated when needed and can be written to disk under their hash key if modified.

In the VoxBox engine, as noted by Dusterwald [18], chunks are fixed-resolution cubic sections of the world grid. This structure allows flexible vertical scaling without the constraints of power-of-two dimensions. Each chunk's position in chunk space serves as the key in a hash map, while its voxels are stored in a flat array where spatial coordinates are inferred from array indices, optimizing memory use and simplifying access.

Chunks provide several practical advantages: they enable dynamic loading/unloading of world sections, support localized updates, and facilitate efficient frustum culling—only active or visible chunks are processed. In this thesis, chunks define cubic regions around voxel models for integration into a sparse voxel grid. Models can either fully occupy a chunk or be split across multiple chunks, with data loaded selectively based on the player's position, which must be tracked within the chunk grid. Figure 3.2 demonstrates the sparsity in the voxel model that chunks are able to create based on dynamic loading. This chunk-based approach is also used to generate multiple complex models for profiling.

3.3.3 Flat array indexing

In the context of voxel data storage, a flat array is a one-dimensional array where each voxel's position is implicitly represented by its index in the array, rather than storing explicit 3D coordinates. Advantages are stated in Dusterwald's thesis [18].

The algorithms implemented in this bachelor's thesis operate on a voxel grid, where each position can contain a voxel ID that marks a distinct material, enabling the construction of voxel-based models. The grid is stored as a one-dimensional flat array, with voxel positions mapped to array indices using the equation (3.1), where n represents the number of voxels along each grid dimension, and (x, y, z) are coordinates of the voxel in the grid.

$$index = y + z \cdot n + x \cdot n^2 \tag{3.1}$$

A similar equation for a flat array to 3.1 can be found in Tustvold's blog [84], but this one is adjusted for Unreal Engine's left-handed system, which uses a Z-up axis [21, Coordinate System and Spaces].

By default, the traversal runs from left to right, bottom to top, and back to front—that is, in the positive directions of the x-, y-, and z-axes. This direction was chosen in the current implementation because it aligns naturally with increasing coordinate values. However, a run can begin at any corner and proceed in any direction. Since the mesh is generated only after the entire traversal is complete, the choice of direction has no visual effect on the final result.

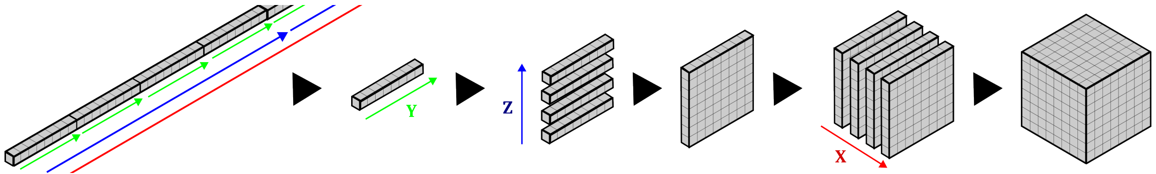


Figure 3.3: Voxel coordinates in voxel array transformed to a voxel grid.

Figure 3.3 illustrates the indexing process during traversal. The leftmost section depicts a continuous one-dimensional flat array of voxels. Each voxel in the array has its index. The first voxel has index 0, the second voxel has index 1, and so on. The array is then gradually decomposed and reconstructed into a three-dimensional voxel grid. Arrows represent different runs. As indicated by them, the traversal proceeds in a fixed sequence: first along the y-axis, then the z-axis, and finally the x-axis. The end of each run represents an increment of the next axis. Because we use 3 axes, and the z-axis is in the middle, we can state that a run terminates whenever the z-coordinate changes. This reconstruction demonstrates the directionality introduced through linear flat array indexing, with subsequent steps showing how the voxels are reoriented along different axes. A voxel gains its three-dimensional position by this reorientation. Chapter 7 provides a more detailed discussion of how this traversal order influences the implementation of the new meshing algorithm.

Figure 3.3 shows how the array is indexed during traversal. The left most part represents continuous flat array of voxels which is broke down and rebuild into a voxel grid. This represent the direction which is created by indexing the flat array and further steps illustrate how each axis rebuilds voxel grid in a new direction. As the arrows indicate, the iteration follows a consistent order: it moves first along the y-axis, then along the z-axis, and finally along the x-axis. A run ends whenever the z-coordinate changes. Chapter 7 explains in more detail how the traversal order affects the implementation of the new meshing algorithm.

3.3.4 Morton Codes (Z-order Curves)

Morton codes [66], or Z-order curves, that interleave the binary bits of 3D voxel coordinates to produce a 1D index that preserves spatial locality, which can improve cache coherence and benefit data structures like octrees. However, Tustvold's blog [84] criticizes that in voxel meshing applications, particularly in dynamic environments, this encoding introduces fragmentation: small changes to the voxel space can significantly disrupt the Z-order, leading to inefficient remeshing and memory access. As a result, this work uses linear indexing with RLE-based compression and optional interval trees to maintain performance and simplify data access.

3.3.5 Criticism of Voxel Grids

As stated in an extended report by Laine and Karras [45], while voxel grids offer a straightforward way to represent volumetric data, they are significantly limited by poor memory efficiency. Representing a full volumetric dataset at resolution N requires $O(N^3)$ memory, even when much of the space is empty. In cases where only surfaces are encoded, the memory footprint can be reduced to $O(N^2)$, but this is still inefficient compared to hierarchical approaches. Furthermore, voxel grids lack adaptability; their fixed resolution enforces uniform detail across the entire space. This results in unnecessary memory usage in sparse regions and prevents level-of-detail optimizations, making voxel grids less suitable for large or complex scenes with varying spatial frequency.

The goal of this thesis is to find a way to address this criticism by using a compression on a flat array of a voxel grid.

3.4 Sparse Voxel Octree

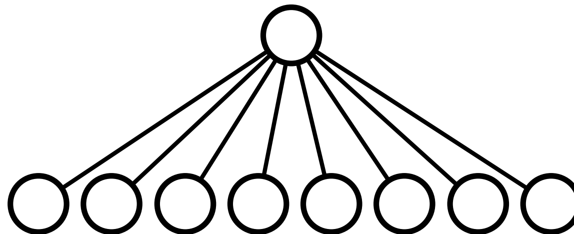


Figure 3.4: Representation of octree sequence.

Sparse Voxel Octrees (SVOs) [1] are hierarchical data structures that store only non-empty voxel regions, offering major memory and scalability benefits over dense voxel grids.

Laine and Karras (2010) [45] developed a GPU-friendly SVO with 64-bit node descriptors encoding child masks, shading data, and either child links or compressed contours. Their stackless kd–restart traversal boosts GPU throughput, and mipmapped voxel bricks (16^3 or 16^3) improve cache usage. Their system achieved over 120 million rays/sec at 5 mm voxel resolution in dense scenes like Sibenik Cathedral.

Later enhancements [46] introduced hierarchical contour compression, asynchronous streaming, and beam traversal, enabling out-of-core rendering of large datasets while preserving performance and visual fidelity. They also compressed per-voxel gradients and normals for efficient shading.

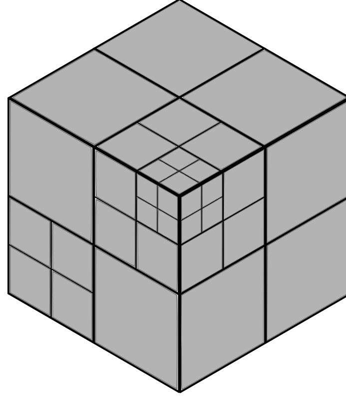


Figure 3.5: 3D representation of SVO structure.

However, SVOs perform poorly in interactive or dynamic environments, as voxel updates often require costly tree modifications. Despite being more compact than dense grids, SVO memory still scales with resolution and must be carefully managed.

Kähler et al. [36] highlight SVO inefficiencies in sparse or irregular data, advocating for Adaptive Mesh Refinement (AMR) trees, which adaptively partition space for better performance.

The 0FPS blog [54] criticizes SVOs' log(n) access time, which hinders real-time voxel engines like Minecraft, where constant-time access is preferred. Octree traversal causes latency, cache issues, and fragmented memory access.

As a result, many engines use chunked 3D arrays with hashable keys, trading structural complexity for faster interaction and simpler memory management.

See section 3.11 for rendering specifics and chapter 6 for a competitive octree-based method.

3.5 Generation of voxel models

Voxelization [14, 1] is the process of converting a model into a voxel-based representation. This technique can be applied to various data structures, including point clouds, polygonal models, and spatial locations. Voxelization and voxel generation are not the primary focus of this thesis; the voxel model uses a flat array, allowing individual voxels to be modified through direct indexing, as described in section 3.3.3. For instance, filling a voxel model involves setting all voxel IDs in the array to a specific value. This simple indexing approach is sufficient for generating basic voxel models, which are sufficient enough for performance profiling. As such, the voxelization of advanced complex models is unnecessary. The most complex voxel models will be generated using noise as described in section 3.5.2.

The generation of voxel models in this work is primarily based on procedural methods using 3D Perlin noise, with parameters such as scale and thresholds controlling terrain variation. However, as noted by Lv et al. [53], voxel models can also be constructed from real-world 3D data using voxelization, which converts point clouds or meshes into a regular grid of voxels. This process begins by defining a bounding box that encloses all input points and discretizing the enclosed volume into a voxel grid, with each point assigned to its corresponding voxel based on spatial coordinates. These voxel structures effectively capture spatial relationships and serve as structured input for mesh reconstruction.

Guan et al. [30] further emphasize the practical advantages of voxelization in meshing workflows. To improve model quality, noise and outliers can be filtered out during voxelization through suppression and refinement steps. The resolution of the voxel model is governed by the voxel size, where smaller voxels yield higher fidelity representations at the cost of increased memory consumption. Importantly, voxel-based representations are inherently well-structured and eliminate the need for complex triangulation or remeshing. Although the voxel models used in this thesis are synthetically generated, the same meshing approach is directly applicable to voxelized real-world data, underscoring its adaptability to various types of input.

The key factor in voxel model generation is the ratio of transparent to opaque voxels, which determines the surface that must be generated. The distribution and ratio of transparent and opaque voxels thus define voxel sparsity. Multiple voxel models with varying voxel sparsity are required to measure voxel meshing performance accurately. Chapter 8 provides examples of voxel models with different sparsity levels.

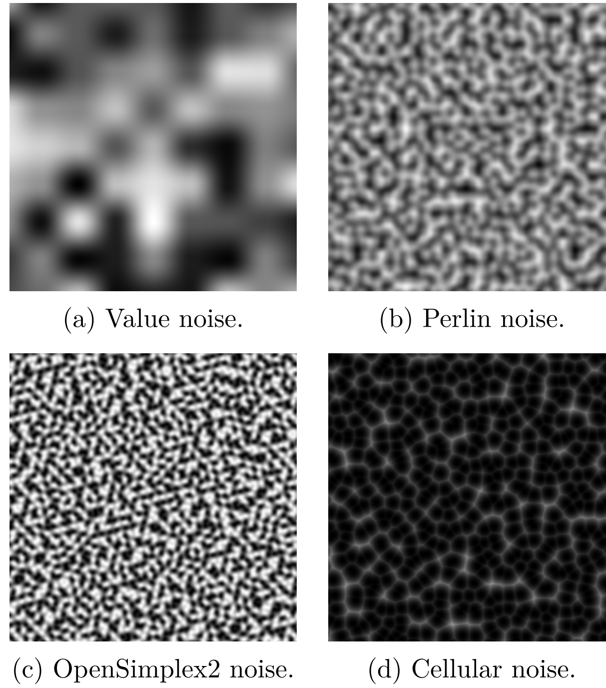
3.5.1 Noise

Article A Survey of Procedural Noise Functions [44] defines noise as:

> *„A noise is a stationary and normal random process. Control of the power spectrum is provided, either directly, or through the summation of a number of independent scaled instances of (typically band-limited) noise.“*

As described in [19] and implemented in the FastNoiseLibrary [69], the noise function generates values by aggregating outputs from pseudorandom functions. These functions produce values within the range from 0 to 1. Key properties of such noise functions include the absence of visible repeating patterns and deterministic behavior. Determinism ensures that the output remains consistent when using the same initial seed for any given input, enabling reproducible noise generation, which is essential in procedural content creation.

The common noise algorithms combine and blend different signals to generate seemingly random yet structured patterns. These algorithms often use techniques such as Perlin, Simplex, or Value Noise, each producing distinct visual patterns and varying computational performance. Examples of various generated noise patterns can be seen in figure 3.6. Noise is commonly layered into octaves, combining signals at different frequencies and amplitudes to create more complex patterns. The result mimics natural randomness. A community library described in section 5.19 can be used to implement noise generation in Unreal Engine 5.



(a) Value noise.

(b) Perlin noise.

(c) OpenSimplex2 noise.

(d) Cellular noise.

Figure 3.6: Noise images generated using FastNoise library [69].

3.5.2 Procedural generation from noise

Noise is a fundamental concept in procedural generation. Procedural terrain generation involves creating a 2D grid where values vary but are smoothly interpolated between neighboring cells. This grid of 2D values is known as a heightmap [76]. Noise functions such as Perlin noise or Simplex noise can be utilized to generate heightmaps. The terrain produced from a single layer of noise is illustrated in figure 3.7. In the implementation, an independent noise generator is used for each voxel material, creating terrain layers. This technique enables the generation of complex features like floating islands, as depicted in figure 3.8.

While procedural terrain and noise generation are unnecessary for this thesis, voxel models generated using heightmaps allow performance testing on a use case with varying voxel distributions.

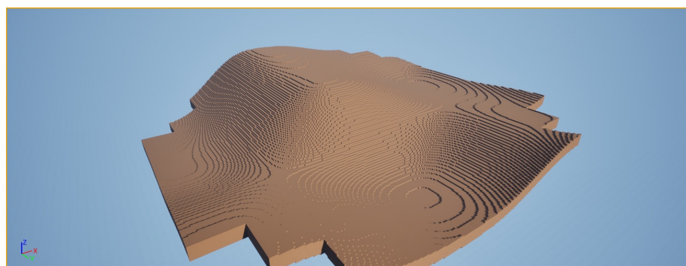


Figure 3.7: Cubic voxel terrain generated using noise.

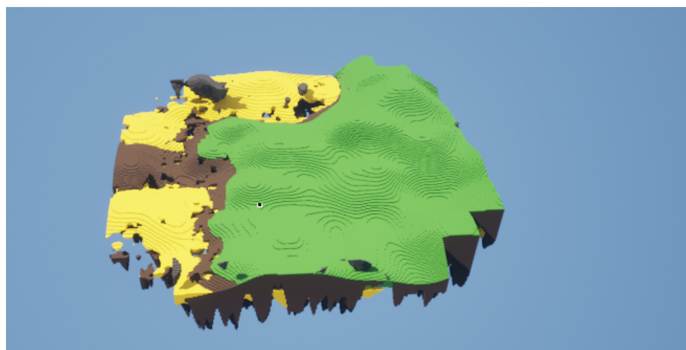


Figure 3.8: Floating islands generated using multiple noise layers

3.6 Level of Detail

According to He et al. [33], Level of Detail (LOD) reduces 3D model complexity based on visual importance, using detailed geometry for nearby objects and simplified forms for distant ones. In voxel systems, LOD is implemented via multi-resolution voxel grids generated through sampling and filtering.

SimLOD [75] constructs sparse voxel octrees directly on the GPU by streaming point cloud data and subdividing nodes when a capacity threshold is exceeded. After populating the tree, inner nodes are voxelized into local 128^3 grids, storing simplified geometry at higher levels. LOD selection during rendering is distance-based, and a spill buffer handles node overflows to maintain streaming performance.

Lengyel [49] describes LOD as essential for balancing performance and visual quality. Voxel-based systems often use SVOs or lower-resolution grids, but transitions between LOD levels can cause popping and misalignments due to mismatched mesh resolutions. The Transvoxel algorithm [49] solves this by generating transition cells that stitch coarse and fine meshes along block boundaries. Each block may generate up to six transition meshes, aligning triangle vertices across LOD levels.

To ensure smooth transitions, vertex placement in low-resolution blocks is guided by high-resolution sampling, and vertex offsets are projected onto tangent planes to preserve surface continuity. The system uses an octree, selecting LODs via visibility culling and projected screen size for scalable terrain rendering.

3.7 Compression

Only compression relevant to this thesis is RLE. Alternatives are mentioned in others.

3.7.1 Run-length Encoding

Run-length Encoding (RLE) compression [2] is a compression technique in which consecutive identical elements, such as voxel types, are represented as a single value paired with a count. For example, the string *aaaaabbbbacaa* becomes $5a4b1a1c2a$, reducing the total storage of characters.

As stated in [54], RLE is usually used during storage or , in voxel systems, RLE can be applied by flattening a 3D chunk into a 1D array and encoding sequences of repeated voxel

[2] Link to Wikipedia: https://en.wikipedia.org/wiki/Run-length_encoding

types. This transformation reduces the storage footprint and can improve data locality and iteration speed during mesh generation or neighborhood traversal.

RLE also supports efficient traversal by allowing iteration over runs rather than individual elements. This can optimize algorithms that process voxel data sequentially. However, standard RLE implementations may exhibit poor performance for random-access operations, as retrieving a single voxel can require scanning multiple runs.

Runs can be imagined as intervals of the same voxel values.

Implementation of RLE sequence in this thesis is described in section 7.3.

Usually, chunks are only stored as compressed RLE sequences according to forum posts [3].

While Run-Length Encoding (RLE) significantly reduces memory usage in sparse voxel regions by storing sequences of identical voxel types, it suffers from a key drawback: inefficient random access. A naive implementation stores runs sequentially, which necessitates linear-time iteration to resolve voxel identity at arbitrary positions. This becomes prohibitive in real-time applications requiring fast voxel lookups.

As mentioned on a forum[4], RLE is often used as a persistent storage for voxels. However, it requires decompression.

An alternative approach to RLE usage is described by Forstmann and Ohya [27], where RLE is employed in the described ray-casting approach to accelerate volume traversal by reducing the number of elements that need to be checked for intersections, thereby improving rendering performance.

3.7.2 Interval tree

A practical enhancement is to store voxel runs within an interval tree as stated by 0FPS blog [54]. Interval trees enable logarithmic-time random access by organizing runs as intervals keyed by voxel indices. This structure supports efficient queries without scanning the entire run list, improving responsiveness during voxel edits or dynamic meshing tasks.

To complement this, as proposed by Tustvold blog [84], spatial hashing and flat array indexing may still be employed for spatial locality, while interval trees handle compressed access. For example, an access query (x, y, z) is first flattened to a 1D index (using standard grid indexing), then resolved against the interval tree for that chunk.

This bachelor's thesis tries to remove the necessity of using interval trees and by leveraging run directions.

3.7.3 Other

Lossy compression can reduce memory usage by allowing small differences between similar parts of a voxel structure. One method creates a Lossy Sparse Voxel DAG (LSVDAG) [43] by finding subtrees that look alike and replacing them with a single shared version. This is done by measuring how different the subtrees are (using Hamming distance), grouping similar ones together with a clustering algorithm, and then keeping just one version for each group. This reduces the number of unique nodes in the DAG, saving space while keeping the overall shape mostly the same.

[3] Link to /r/VoxelGameDev subreddit: https://www.reddit.com/r/VoxelGameDev/comments/15bycu3/method_to_compress_chunks_for_voxel_engine/

[4] Subreddit link: https://www.reddit.com/r/VoxelGameDev/comments/15bycu3/method_to_compress_chunks_for_voxel_engine/

Additional compression of voxel models can be found in Voxel Compression documentation[5]

3.8 Overview of visualization

This section overviews all algorithms capable of rendering a voxel model or transforming it into a polygonal representation, which can be rendered as a polygonal mesh. This thesis will focus solely on visualizing the voxel grid as meshes. Each voxel will contain only information about its identification. The material will always represent a surface that is either transparent, opaque, or empty.

In rendering, objects are represented by their surfaces and broken down into basic primitives. Rendering operates as a view-dependent sampling process, where the idea is to sample a 3D scene to generate a 2D image. The 3D scene is the continuous source signal, while the 2D image represents the reconstructed signal. Sampling below twice the highest frequency leads to aliasing [15].

3.9 Surface and mesh definition

According to Encyclopedia of Multimedia [28]:

> *„Mesh is a 3D object representation consisting of a collection of vertices and polygons.“*

A surface [10] in computer graphics is a smooth, continuous 2D shape in 3D space that represents the boundary of a solid object. It is typically defined as an orientable, continuous 2D manifold embedded in 3D space. When creating surfaces from samples, it's important to ensure the surface is continuous and smooth. These ideas apply to both continuous surfaces and polygon meshes.

A 3D object is often represented by a polygon mesh, which consists of vertices and polygons that define its shape in 3D space. It is one of the fundamental representations of objects in computer graphics. For a more advanced mathematical definition of polygonal meshes, refer to [10]. To optimize for modern GPU, polygon meshes usually contain only triangles, as the hardware is optimized for triangle mesh computation. A mesh typically includes vertices with specific positions and edges connecting them, which allows game engines and graphical applications to define, manipulate, render, and perform collision detection on 3D objects. An overview of methods and mesh rendering details can be found in [1]. However, depending on the implementation, additional information is often required to describe the object. Mesh handling in Unreal Engine is further described in section 5.6.

3.10 Ray rendering overview

Ray-based techniques such as ray casting and ray marching are fundamental for visualizing 3D volumetric data on GPUs [1]. In Direct Volume Rendering (DVR), rays are emitted from the camera into the volume, sampling interpolated voxel values and computing gradients for lighting models like Phong [5]. Ray marching advances the ray in small steps, sampling values at each position. These samples are mapped to color and opacity using transfer

[5] Link to Voxel Compression documentation: https://eisenwave.github.io/voxel-compression-docs/

functions and composited to form the final image. To manage large data sets efficiently, the volume is partitioned into bricks and rendered using optimizations like empty space skipping and early ray termination.

GPU-based implementations such as GigaVoxels DP [70] demonstrate real-time performance through dynamic parallelism, LOD mipmapping, and brick streaming, avoiding CPU-GPU synchronization issues present in earlier approaches of [15]. Similarly, the Antara framework [37] applies ray casting to medical data visualization, supporting interactive adjustments of transfer functions and internal structure exploration.

However, these methods lack the interactivity required in game engines, particularly regarding collision and physics. Since most implementations rely entirely on GPU shaders, they cannot interact with world entities. Even advanced CPU methods using BVH and SIMD [41] handle visibility, not gameplay. Shader-based ray rendering also lacks practical voxel grid collision.

While ray-based methods enhance visual fidelity, their limitations with interaction, dynamic geometry, and physics make them unsuitable for engines like Unreal Engine [20]. Therefore, this thesis adopts a mesh-based voxel representation to enable real-time updates, physics, and gameplay features.

3.11 Octree Rendering

Sparse Voxel Octrees (SVOs) serve as acceleration structures for rendering voxel-based scenes, utilizing the hierarchical and sparse characteristics of octrees for efficient ray traversal and visibility determination. Laine and Karras [45] present a stack-based traversal algorithm in which a ray descends into intersected child nodes. Each node encodes child pointers and voxel attributes, such as color, into compact 64-bit words to reduce memory usage and enhance GPU memory coherence through dense packing.

Their method employs cone tracing, where cones, rather than discrete rays, are traced to approximate global illumination effects including diffuse interreflection, ambient occlusion, and soft shadows. Cone tracing is performed against precomputed voxel data containing both color and lighting information. This data is filtered across octree levels according to the cone's footprint, allowing accumulation of lighting information over a spatial region.

Level-of-detail (LOD) selection is integrated into traversal by varying octree depth based on cone angular size or distance from the camera. Coarser octree levels are used for broader cones or distant regions to reduce sampling cost, with a corresponding decrease in detail.

To address limitations in traditional octree traversal, SparseLeap [32] separates empty space skipping from ray traversal. It generates view-independent occupancy geometry using a hybrid object-order/image-order method and rasterizes it into per-pixel ray segment lists. These lists enable linear ray iteration without hierarchical traversal and support efficient rendering of large, sparse, and dynamic scenes.

The extended implementation of SVO rendering [46] introduces GPU-focused optimizations such as persistent threads and spatial indexing to reduce branch divergence. Lighting data is stored in a format conducive to filtering and interpolation. Refinements to cone tracing include adjustments to aperture, step size, and sample accumulation techniques to reduce artifacts such as light leaking and aliasing.

The method has specific constraints. The use of precomputed lighting data does not support fully dynamic lighting or geometry, as changes require recomputation of the octree and lighting content. Cone tracing approximates light transport and does not capture phenomena such as caustics or specular reflections. Visual quality depends on voxel grid

resolution and octree depth; higher detail requires deeper trees, increasing memory and computational costs. Transitions between LOD levels may introduce visual discontinuities if not filtered appropriately.

Chapter 4

Voxel meshing algorithms

Voxel meshing represents a separate branch of voxel rendering, distinct from ray-based and volume-based techniques discussed in the previous chapter. While all rendering approaches aim to visualize voxel data, meshing focuses on converting discrete voxel representations into polygonal geometry. This enables compatibility with mesh-based rendering pipelines and supports engine-level features such as lighting, collision, and physics, which are essential in platforms like Unreal Engine.

Because of its critical relevance to the topic of this thesis, voxel meshing is addressed in this dedicated chapter. The ability to transform voxel models into mesh structures is fundamental to achieving real-time interaction and efficient visualization in environments that do not support volumetric data.

This chapter presents a survey of voxel meshing algorithms developed by the voxel engine fan community, beginning with smooth surface extraction methods and continuing with cubic, quad-based approaches that align with the visual and performance goals defined earlier. Since no comprehensive summary of these algorithms was available at the time of writing, this survey also makes additional research effort undertaken as part of this thesis. The overview serves as the conceptual foundation for the design my new voxel meshing algorithm introduced in chapter 7.

The primary objective of this thesis is to introduce this new voxel meshing algorithm as a contribution to these existing methods.

4.1 Smooth surface meshing

The following sections contain algorithms that can be used for the voxel meshing process. They are either algorithms used for remeshing pr isosurface extraction as stated in the 0FPS blog [58]. These algorithms will not be used, but a competitive algorithm is able to produce smooth surfaces using these methods. The competitive solution is described in a section 6.3

4.1.1 Marching cubes

Marching Cubes [1] is a high-resolution isosurface extraction algorithm introduced by Lorensen and Cline in 1987 [52]. It was originally developed for medical imaging and remains a widely used method for generating smooth polygonal meshes from volumetric data such as voxel grids.

The core concept is based on treating each point in the voxel data not as a cube center, but as a sample. A single cube is defined by eight such samples (voxels) at its corners. The algorithm compares each voxel's value to a chosen surface threshold to determine whether it lies inside or outside the surface. Based on this binary classification (0 for outside, 1 for inside), an 8-bit index is formed that uniquely identifies the cube's configuration.

This index is then used to access a precomputed lookup table of 256 possible surface configurations, which describe how the surface intersects the cube edges. To simplify the table construction, complementary and rotational symmetries are applied, reducing the number of unique triangulation patterns to just 1416.

Each intersected edge's vertex position is estimated using linear interpolation between the scalar values at the edge's endpoints. This estimation improves the accuracy and smoothness of the resulting mesh. Normals at the triangle vertices are calculated using gradients derived via central differences, and are then interpolated to each triangle vertex to enable smooth shading through Gouraud interpolation.

The algorithm follows these steps:

1. Load four slices of the volume into memory.
2. Form a cube using four neighbors from one slice and four from the adjacent slice.
3. Compare voxel values at the cube's eight corners with the surface threshold to compute an index.
4. Use the index to retrieve intersected edges from the lookup table.
5. Perform linear interpolation along edges to determine vertex positions.
6. Calculate gradients and interpolate normals.
7. Output triangle vertices and associated normals.

Marching Cubes provides advantages over simpler voxel-to-mesh techniques due to its ability to approximate curved surfaces and produce smooth transitions. However, because of its reliance on edge-based lookups, it may produce ambiguous topologies and uneven triangle quality in some configurations.

Several enhancements have been introduced to the original algorithm. Efficiency improvements leverage data coherence to reduce redundant edge interpolations across adjacent cubes, improving performance by a factor of three. Functional enhancements enable Boolean operations (e.g., union, intersection) and solid modeling features like cutting and capping, which are critical for interactive medical applications.

Despite its limitations, Marching Cubes remains an influential algorithm in voxel meshing and serves as a foundational reference for evaluating newer techniques.

4.1.2 Surface nets

Based on the work of Gibson [29] and the extensions proposed by de Bruin et al. [11], Surface Nets are a technique for generating a polygonal surface from volumetric voxel data. A vertex is placed at the center of each voxel cell that contains a material boundary. These vertices are then connected with their neighbors to form quadrilateral faces, which are later triangulated.

The process starts by identifying voxel cells in which the binary material occupancy differs among the eight corners. Each such cell is considered a surface cell. A vertex is placed at the center of the cell, and edges are formed by linking this vertex to those in adjacent surface cells, using a 6-connected neighborhood.

To improve the quality of the resulting surface, the positions of the vertices can be adjusted through constrained relaxation. In this step, each vertex is moved toward the average of its neighboring vertices, while remaining within the bounds of its original cell. This constraint ensures that vertices do not drift across boundaries and preserves the fidelity of the original volume data.

Vertex relocation may use additional scalar information, such as greyscale voxel values or a precomputed distance field, to better approximate the underlying surface. This adjustment is performed iteratively to reduce surface noise and irregularities. The combination of averaging and gradient-based displacement allows the surface to achieve both smoothness and accurate conformance to iso-surfaces.

After vertex relaxation, the mesh is finalized by converting each quadrilateral face into two triangles. This is typically done by selecting the diagonal that produces the better-shaped triangles, either through length comparison or Delaunay triangulation criteria.

Compared to the Marching Cubes algorithm, Surface Nets offer several advantages. Marching Cubes relies on a lookup table of local configurations, which can lead to ambiguous topologies and poorly shaped triangles. Surface Nets avoid these problems by using a regular connectivity scheme and by deferring topology resolution to later processing stages. Furthermore, Surface Nets with relaxation, particularly the SNERT variant, produce higher quality triangles and smoother surfaces while preserving geometric features. Although Marching Cubes may provide slightly better vertex-to-surface accuracy in some configurations, the difference is minimal when grayscale information is used for gradient correction.

4.1.3 Dual countoring

Dual contouring [35] is an isosurface extraction algorithm developed to address some of the limitations inherent in methods such as Marching Cubes and Surface Nets. It preserves sharp features more accurately by leveraging Hermite data—specifically, the positions and normals of intersections between voxel edges and the implicit surface function.

In contrast to traditional algorithms that place mesh vertices at grid corners or edge midpoints, dual contouring computes optimal vertex positions inside each cell using Quadratic Error Functions (QEFs). The QEF is formulated to minimize the squared distances from a candidate vertex position to tangent planes defined by normals at Hermite samples. The solution to the QEF yields a vertex location that best approximates the local geometry of the implicit surface. To ensure numerical stability, the method applies QR decomposition with pivoting or SVD (Singular Value Decomposition) when solving the QEF.

One of the notable advantages of dual contouring is its ability to maintain mesh continuity while capturing both smooth and sharp features of the original scalar field. This makes it particularly suitable for applications where geometric fidelity is critical, such as scientific visualization and 3D modeling.

Furthermore, dual contouring can be extended to an adaptive version by utilizing an octree data structure. This adaptive dual contouring refines the mesh only where additional detail is required, significantly reducing the polygon count while maintaining visual accuracy.

The algorithm consists of three primary steps:

1. **Edge Intersections**: Compute Hermite data by finding surface-edge intersections and estimating the corresponding normals.
2. **QEF Optimization**: Solve the QEF for each cell to find the optimal vertex position that approximates the zero-crossings of the scalar field.
3. **Mesh Construction**: Connect the computed vertices across cell faces to generate the final surface mesh.

Dual contouring offers a significant improvement over both Surface Nets and Marching Cubes by more accurately preserving sharp features and adapting to surface complexity. While Marching Cubes relies on edge-based vertex placement and lookup tables—often leading to ambiguous topologies and smoothed geometry—dual contouring positions vertices freely within voxel cells by minimizing a Quadratic Error Function (QEF) derived from Hermite data. Surface Nets, though topologically sound and simpler than Marching Cubes, typically places vertices at cell centers and lacks the geometric precision needed to capture sharp details. In contrast, dual contouring incorporates surface normals directly into vertex computation, producing meshes that are both more accurate and more efficient. Additionally, its support for adaptive resolution through octree-based refinement enables localized mesh detail without unnecessary polygon density—something neither Surface Nets nor Marching Cubes provide natively.

4.2 Cubic Quad-based Voxel Meshing

Because cubic voxel meshing that uses quads is used in this thesis, an overview of possible and found algorithms will be presented here from 4.2.3 section onwards. Before that, in the next two sections, I introduce 2 main problems that arise from using this method. These problems align with stated performance goals.

4.2.1 T-Junction problem

In computer graphics [48], a T-junction refers to a situation where two polygons share an edge composed of duplicated vertices. When the vertices have identical coordinates and belong to the same object, modern graphics hardware is typically designed to rasterize complementary pixels, ensuring no gaps appear between the polygons. This compatibility arises because the rasterization process generates pixels that precisely fill the shared edge.

However, complications arise when adjacent polygons sharing an edge are transformed into a different space. Each vertex may be defined in a local coordinate system, and when these vertices are transformed into world space, floating-point round-off errors can result in slightly different coordinates for the shared vertices. Consequently, the vertices no longer align perfectly, potentially leading to visible seams during rasterization. The characteristic „T“ shape formed at the junction gives rise to the term „T-junction.“ This issue is one of the primary causes of visible seams in real-time rendering engines that do not employ techniques to mitigate it. One such technique may be checking z-buffer depth anomalies.

T-junction issues can manifest in two primary scenarios:

1. Polygons share an edge, and duplicated endpoint vertices of the polygons share the same coordinates.

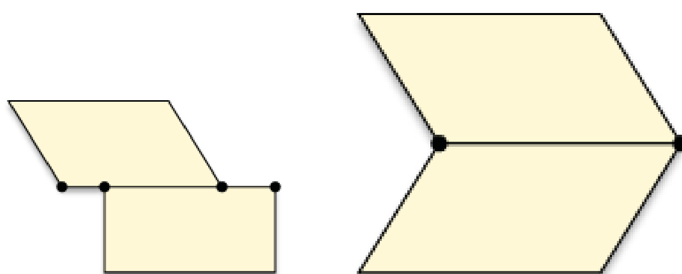


Figure 4.1: Screenshots of a T-Junction situations taken from [48].

2. Duplicated endpoint vertices of the polygons lie on the same edge but have differing coordinates in a single axis.

The first scenario should yield slightly better results. However, duplicating vertices in culled chunk meshes can still lead to the formation of T-junctions. These T-junctions typically cover a smaller surface area, making the gaps less noticeable. The likelihood of T-junction formation increases with the distance of misalignment between the duplicated endpoint vertices, as more significant misalignments produce a larger area where gaps may appear.

Existing meshing techniques have been criticized for rendering surfaces with quads [56]. This approach often generates redundant vertices, exacerbating the potential for T-junctions. As previously mentioned, limiting the transformation of vertices or reducing the distance between vertices in the edge axis can help mitigate the problem. With advancements in modern GPU hardware, the precision of floating-point arithmetic has significantly improved, reducing the likelihood of T-junction-related issues [7]. Enhanced precision helps maintain better alignment of vertices even after transformations, making the problem increasingly less common in contemporary rendering pipelines. Creating a static exporter may be better when working with less-powerful hardware. This means converting voxel models to mesh format before visualization and using only exported mesh assets during runtime. This would create a static exporter, and it would prevent dynamic interaction with voxel models.

Figure 4.2: Screenshot of a T-Junction seam taken from [7].

The technologies employed within Unreal Engine (5) impose specific per-vertex data constraints while simultaneously reducing the necessity for accommodating obsolete hardware. Notably, Unreal Engine requires extensive data for each polygon. These include attributes such as a normal vector for each vertex to enable accurate mesh shading and lighting, material indices for shader specification, and texture coordinates. Even when vertices occupy identical spatial coordinates, variations in other data, such as a normal vector orientation or parent polygon material, can differentiate them. Quads are particularly advantageous in this context, as they can support the additional data required by Unreal

Engine, such as embedded support for textures in Unreal Engine materials. Given this thesis's Unreal Engine requirement and the decreased importance of detail, I have chosen a quad-based approach. This approach aligns with the requirements of most modern gaming engines. I deem the impact of possible T-junctions negligible on hardware compatible with Unreal Engine.

The issues that result from T-Junctions and floating point error may also influence complex colliders, but further testing is needed.

This can be solved by anti-aliasing special techniques described in [49] or in section 4.2.10 or changing resolution or target hardware.

4.2.2 Long-thin triangle problem

Overdraw [26] occurs when multiple fragments are shaded for the same screen pixel due to overlapping geometry. One of the primary causes of this inefficiency is the presence of long, thin triangle shapes with a high aspect ratio that intersect many pixels but contribute minimal coverage to each.

As stated in GPU Performance for Game Artists [68], modern GPUs shade in 2×2 pixel quads. Even a minimal intersection with one pixel of such a quad triggers execution of the fragment shader across the entire group. Consequently, thin triangles or long geometry significantly increase fragment shader workload without improving visual quality, a phenomenon known as overshading. This inefficiency stems from the GPU shading entire 2×2 quads even if only one pixel within them is touched by a sliver of a triangle, wasting up to 75% of shading work.

In voxel meshing, long quads resulting from aggressive merging can lead to similarly shaped long, thin triangles after triangulation. When chunks are too large, these quad strips can span many pixels, worsening the problem. Moreover, because GPUs rasterize using square or rectangular regions, these long, narrow triangles force the rasterizer to cover a much larger area than necessary, further increasing the rendering cost.

Although the precise performance threshold needs to be measured, my supervisor approximates that aspect ratios exceeding 1:100 may be problematic. I recommend the following:

- **Adjust chunk size**: Chunks with fewer voxels naturally break up long quad strips, resulting in more compact and efficient triangles.
- **Maintain good triangle aspect ratio**: Limit the maximum length of merged quads to avoid producing extreme triangle shapes.
- **Implement Level of Detail**: Lower-resolution meshes at distance reduce both triangle count and shading cost per fragment.
- **Increase number of quads**: Prefer evenly shaped triangles even if this increases the polygon count slightly. They offer better rasterization efficiency and load balancing on the GPU.

4.2.3 Simple mesher

The most straightforward approach to voxel meshing involves iterating over each voxel in the volume and generating a cube composed of six quads for every non-empty voxel. Its main advantage lies in its ease of implementation, as described in the 0FPS blog [55] and

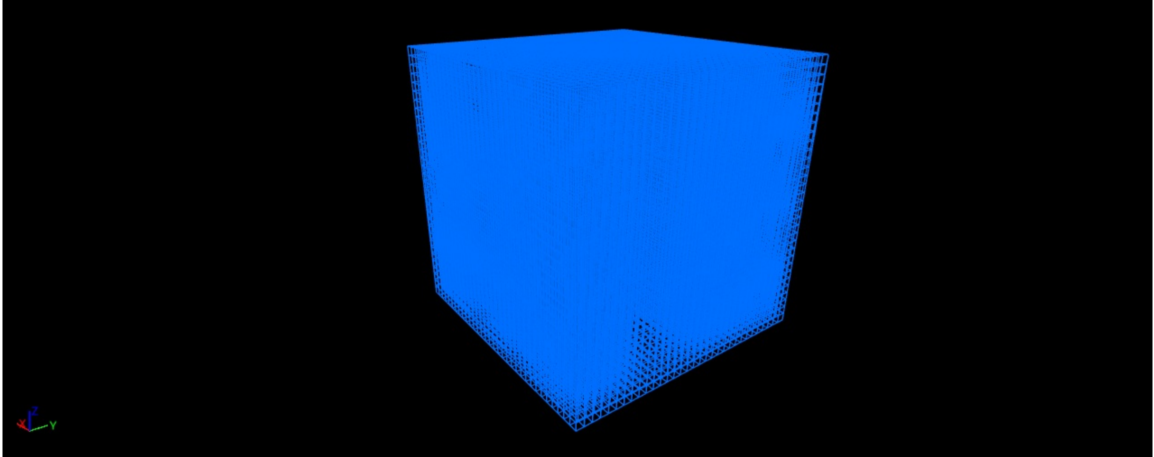


Figure 4.3: Basic chunk mesh generated using modified version of 7.2.

fast iteration complexity through a voxel grid which is $O(n)$ where n is a number of voxels in the grid. However, it is also highly inefficient because it produces a very large number of quads and vertices. The resulting overhead significantly impacts rendering performance, often leading to severe frame rate degradation. As a result, this method is considered impractical for real-time applications.

Despite its limitations, this method is often used as a foundational concept when developing more advanced voxel meshing algorithms. These more sophisticated approaches aim to optimize this naive method by reducing the number of generated faces, ideally producing only the visible quads that contribute to the final rendered surface.

This basic form of voxel meshing directly maps the sampled voxel material to each face of the cube, as illustrated in Figure 4.3.

4.2.4 Culled mesher

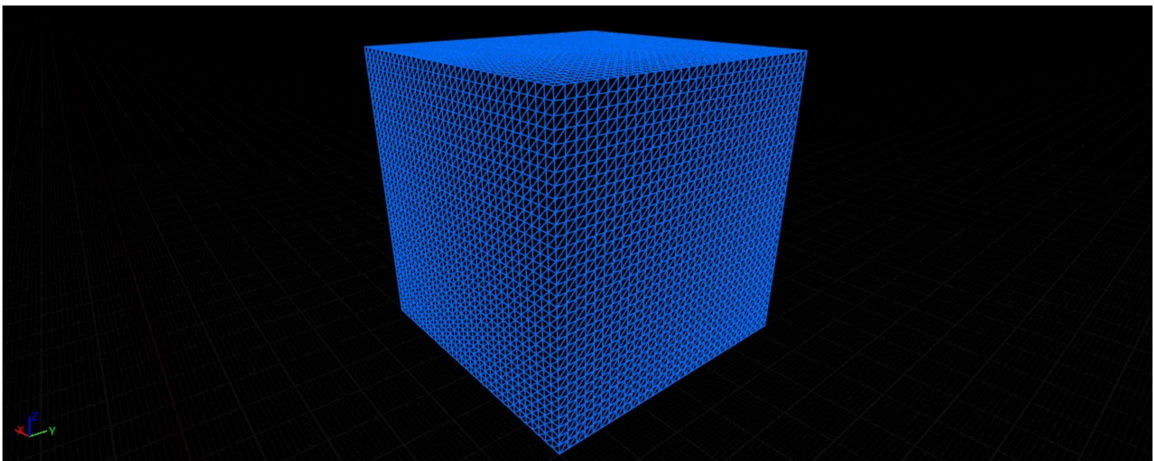


Figure 4.4: Culled chunk mesh generated using modified version of 7.2.

An enhancement to the original algorithm involves eliminating internal faces within regions of opaque material, preserving only the surface quads. While conceptually similar to

the original approach, this variation requires tracking whether the current voxel lies within opaque material, this means transitions between empty (invisible) and opaque (visible) voxels. By examining neighboring voxels, a algorithm can identify boundaries between opaque and non-opaque regions and generate quads only at those boundaries. Since checking neighboring voxels is a constant-time operation, this asymptotic complexity remains unchanged. However, the number of generated quads is significantly reduced, as only surface quads are retained.

When two adjacent voxel cubes are opaque, the shared face between them can be discarded, as it will never be visible. This process yields a mesh composed solely of exterior-facing quads, leaving the internal volume empty [1].

0FPS blog [55] compares this to the naive approach, this method and notes an improvement. The practical complexity drops from $O(n^3)$ to $O(n^2)$ a gain in complexity of $O(n)$ where n, represents number of voxels in a voxel grid. However, this improvement is highly dependent on the structure of the voxel model. The blog also highlights that the primary advantage of this optimized algorithm lies in its ability to retain only the quads that are potentially visible to the observer resulting in an efficient optimization of vertex count. It allows rendering of only the outer surfaces of visible voxels, with each quad corresponding to a single exposed face. Internal faces, which are obscured from view, are culled. This is why this process may be called `face culling`. The reduction in the number of generated vertices significantly improves performance, making the method suitable for real-time applications.

For this reason, I refer to the resulting quads produced by this technique as culled quads.

Culled mesher is increasingly more used because several enhancements in LOD generation and binary bitwise optimizations have been made. Binary variant description can be found in a YouTube video[1]. Another shift to binary optimizations can be seen later in section 4.2.8.

4.2.5 Greedy meshing

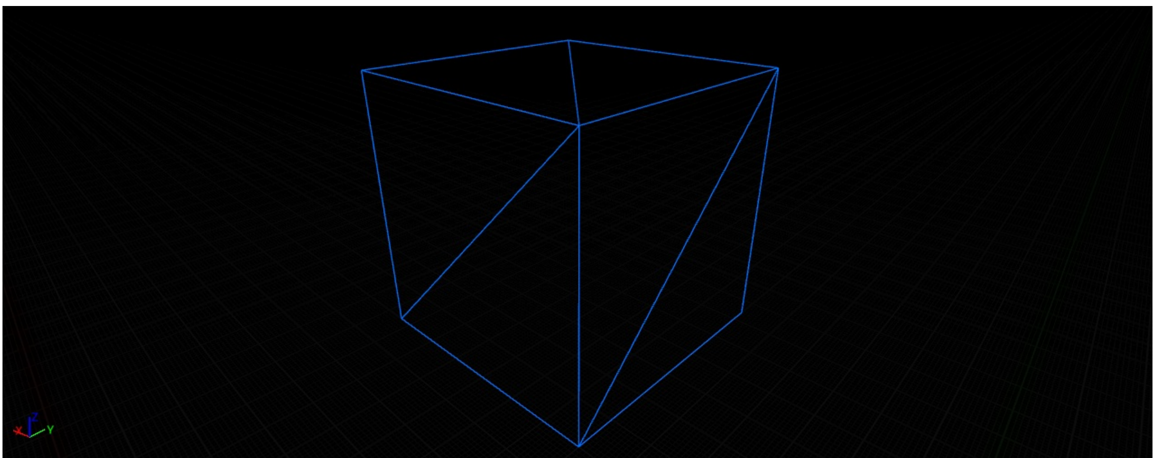


Figure 4.5: Greedy chunk mesh generated using 7.2.

[1]Link to Binary meshing YouTube Video: https://www.youtube.com/watch?v=4xs66m1Of4A

The greedy meshing algorithm builds upon the previous culling algorithm and excludes all invisible quads. The difference is that it combines all adjacent quads representing the same material.

Since the chunk has a regular cubic shape, one cube face would contain 64 voxels. With only culling meshing applied, this face would consist of 64 quads. However, after merging the quads, only 1 quad would be generated for each chunk face. Although this was an extreme case, it resulted in an improvement by a factor of 64 and removed all the unnecessary vertices as stated in 0FPS blog [55].

Quads, representing the 2D faces of a generated voxel mesh cube, reduce the merging problem from a three-dimensional (3D) spatial challenge to a two-dimensional (2D) surface problem. This simplification is grounded in the certainty that voxel samples maintain a consistent distance from one another, resulting in the cube's faces forming unique 2D surfaces. A quad ordering should allow a merge of the quads on the same surface level for each surface.

The objective is thus to determine the optimal ordering of quads for each surface while minimizing the total number of quads and, consequently, the number of vertices required for each mesh. This reduction should improve computational efficiency and simplify the mesh structure. An example of a total ordering of quads can be found in [55].

The capability to order quads using a total order enables their lexicographical comparison. This property ensures the existence of a unique least quad within the ordering. This quad is then used to construct greedy mesh so that each quad fully covers at least one edge in the region, making the number of quads directly proportional to the number of edges. In the worst-case scenario, such as a collection of independent cubes scattered in space, the greedy meshing process will yield the same number of quads as a culled mesh. Despite this, the greedy meshing algorithm remains linear in computational complexity. However, its efficiency and the extent of its benefits diminish as the number of edges in the polygonal model increases. This decrease in effectivity applies to every very sparse voxel model. The estimated efficiency of greedy meshing, in terms of reducing the number of quads (and vertices) compared to culled mesh, is $O(\sqrt{(n)})$, offering substantial improvements over the latter method in most practical scenarios [55].

This algorithm is the most influential contributions to voxel meshing algorithms, frequently referenced in community-driven discussions and blog posts. The main contribution of this algorithm was the introduction to greedy–meshed quad meshes.

4.2.6 Naive greedy meshing

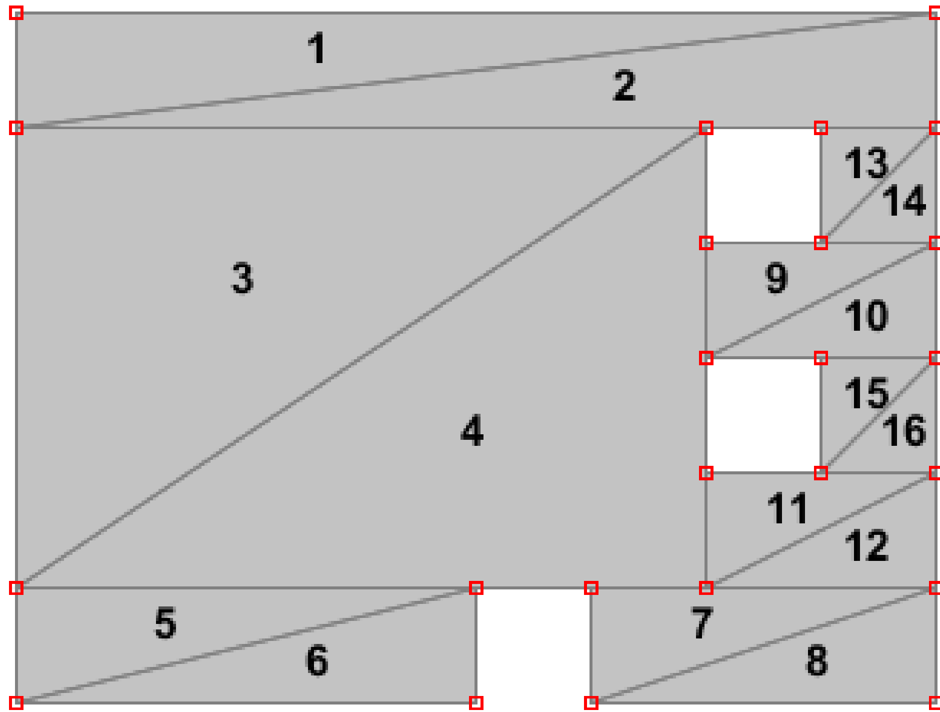


Figure 4.6: Example of naive greedy meshing not finding optimal solution taken from a blog [7].

This approach is introduced by Blackflux blog [7]. It tries to simplify process of greedy meshing by performing a more naive approach to generating greedy meshes. It's goal is to improve greedy meshing speed at a cost of worse quad merging. This can be seen in figure 4.6.

4.2.7 Sector's Edge run meshing

This is a quad-based algorithm that combines voxel blocks into structures referred to as runs. A run represents a linear segment of voxels along a single axis, where all voxels share identical coordinates on the remaining axes and are composed of the same material. Every visible non-empty voxel must be part of a run. If a voxel is not part of an existing run, the algorithm initializes the initial coordinates of a new quad and continues to check subsequent voxels in the model. This process continues until the chunk's end is reached or the voxel material changes. The last voxel with the same material determines the final coordinates of the quad. Quad is initialized based on the visibility of the first voxel in the run. The run is not divided, even if some of its subsequent voxels are hidden. This is done to reduce

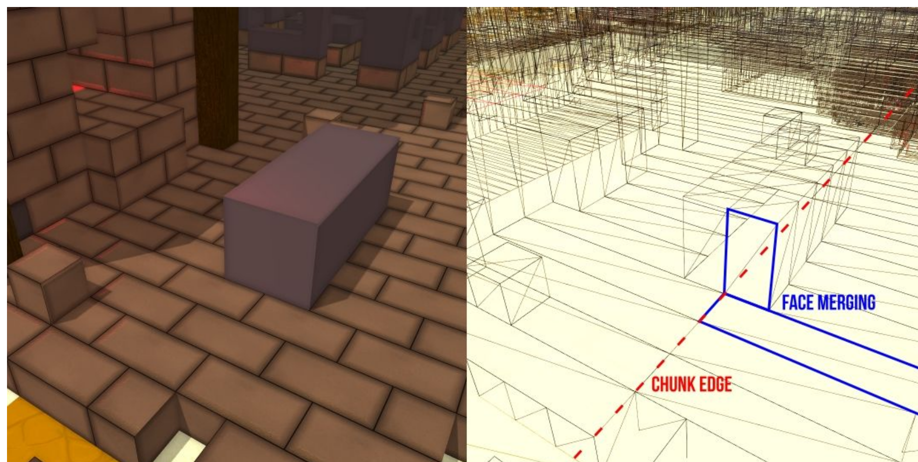


Figure 4.7: Screenshot of face merging meshing taken from [71].

the amount of triangles in the mesh. Voxels used in a run are stored in a special structure and are ignored [71].

This algorithm is implemented within a custom game engine. This means there is no overhead compared, for example, to Unreal Engine. Following optimization, the meshing of a chunk achieves a processing time of `0.48 ms` as stated in [72]. The exact number of generated triangles is not explicitly specified. Comparison in Unreal Engine is not feasible, as no direct implementation of this algorithm within Unreal Engine has been found. The primary advantage of this approach is its ability to regenerate meshes faster than conventional greedy meshing techniques (4.2.5). However, the increased amount of long, thin triangles may impact mesh rendering performance. The fundamental idea of merging voxels into runs was also used in my algorithm in section 7 and, of all listed algorithms, the two algorithms share the most similarities.

The main contribution of this algorithm is the utilization and handling of run strip meshes which can be seen in figure 4.7.

4.2.8 Binary greedy meshing

Binary greedy meshing is a community-developed technique presented in a YouTube tutorial[2]. This method leverages binary operations to accelerate voxel model traversal by creating binary grid planes for each block type, where each binary grid is a 2D bitmask representing whether a voxel face should be meshed. The algorithm performs greedy meshing on these binary slices, generating greedy meshed quads. In total, three binary grids are generated (one per axis), each stored in a flat array. These grids are populated by iterating over the voxel data and setting the appropriate bits based on voxel face visibility.

Before greedy meshing is applied, binary slices undergo a face-culling phase. During this phase, voxels are evaluated for visibility: if a face is not culled, its corresponding bit is set to 1 in the binary grid. The goal of this approach is to reduce the number of times the voxel model is sampled, thereby improving performance. A key innovation lies in the use of bitwise operations to perform both face culling and the construction of binary grids efficiently. For each axis, two directional binary masks are generated, one fo a direction and one for a reverse direction, using different bitwise operations.

[2]Binary Greedy Meshing YouTube Video: https://www.youtube.com/watch?v=qnGoGq7DWMc

The resulting visible face masks are then separated by block type, using a hash map to retrieve the type based on position. The final step involves running a binary-optimized greedy meshing pass on the culled data.

The most computationally expensive part of this algorithm is the generation of the binary sampling grids. As such, its primary optimization is the use of bitwise operations to perform fast face culling. Notably, this approach can also be used independently of greedy meshing as a standalone culled mesher.

The implementation has a fixed chunk size of 32 voxels per chunk dimension. The algorithm's implementation is publicly available on GitHub[3].

This is the most recent advancement in voxel meshing algorithms I could find. One of the more general trends seems to be rewriting voxel meshing algorithms into Rust[4], as can be seen for example in another community YouTube video[5].

4.2.9 Optimal greedy meshing

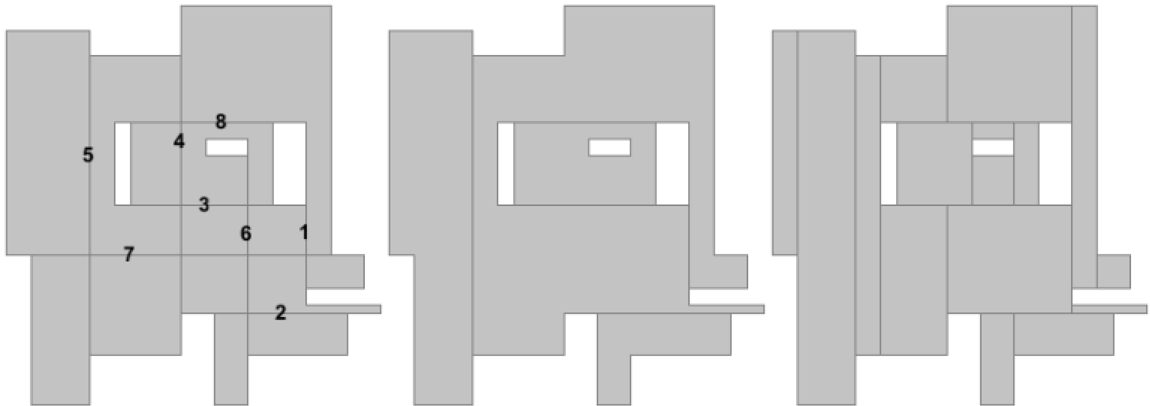


Figure 4.8: Illustration of finding optimal greedy mesh taken from [8].

Optimal Greedy Meshing is introduced in a blog [8] and it is a quad-based algorithm using the bipartite graph described in [23]. The core idea relies on finding good polygon edges based on concave vertices in a polygon. The resulting edges can be represented as a bipartite graph, where two entries are connected exactly if they intersect. This graph can be used to find a maximal independent set, which is the inverse of a minimal vertex cover. Details and sweep line algorithm for generating triangles from polygon and the set of edges can be found in the blog. This process is illustrated in 4.8.

---

[3] Binary Greedy Mesher GitHub Repository: https://github.com/TanTanDev/binary_greedy_mesher_demo

[4] Rust official website: https://www.rust-lang.org/

[5] Link to voxel meshing in Rust YouTube video: https://www.youtube.com/watch?v=meup98Okxwk

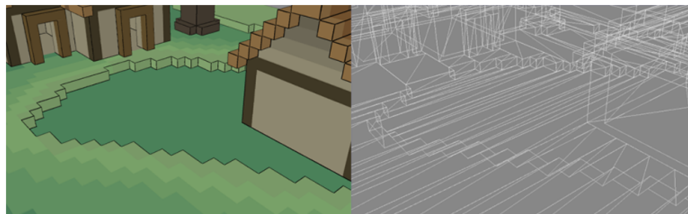


Figure 4.9: Screenshot of optimal greedy meshing taken from [8].

This algorithm serves primarily as a demonstration to identify the minimal number of quads required in quad-based meshing. As outlined in [9], performance evaluations revealed that the algorithm exhibits significantly slow processing times. The optimal application of this algorithm would be for pre-runtime conversion of a mesh derived from a voxel-based model. This makes it unsuitable for applications requiring dynamic interactions. Nevertheless, it establishes a valuable baseline for comparing the minimal achievable reduction of quads. Experimental results indicate a few percent triangle reduction (less than 5%) over other greedy meshing techniques. This result can be seen in figure 4.9.

4.2.10 Monotone meshing

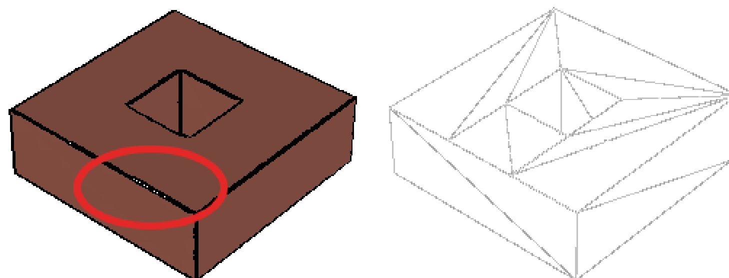


Figure 4.10: Example of monotone meshing error [7].

This algorithm was proposed to eliminate redundant vertices created by quads and resolve the T-Junction problem. The algorithm is based on the principle that monotone polygons can be easily converted into triangles. The original approach is described in [8]. However, a limitation of this algorithm is that it can address the T-Junction problem only for 2D surfaces. When these surfaces are combined into a 3D mesh model, the T-Junction problem persists, as demonstrated in figure 4.10. Fixes for monotone meshing are discussed [7].

4.2.11 Poly2Tri Meshing

This design is presented in [8] and was developed as an alternative to monotone meshing. The main principle of this meshing approach is to use an algorithm or library to generate the polygon's outline from voxels and then triangulate it. In the original design, the Poly2Tri library [6] was used (hence the name). This open-source library can triangulate any

[6] Poly2Tri library: https://github.com/greenm01/poly2tri

weakly simple polygon, requiring only its outline as input. The algorithm for generating the polygon's outline is described in [87]. In the original design, interior holes or the polygon outline must not touch each other. The advantage of this library is that it does not create any new vertices, thus preventing the occurrence of duplicates.

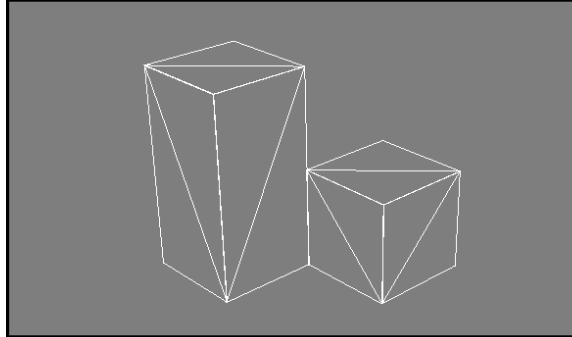


Figure 4.11: Example, where T-Junctions may occur when using this method taken from [9].

However, the result could still generate T-Junctions, as seen in the figure 4.11. As mentioned in the previous section about T–Junction (4.2.1), this can be mitigated by adding additional vertices to the place where the other polygon touches the edge. Also, it is necessary to use neighboring chunks or world segments when performing the meshing, and based on stated performance, this method is slow, and its primary proposed usage is for a static mesh exporter [9]. While modern algorithms could potentially enhance their speed, this approach focuses on eliminating the T-junctions rather than achieving the fastest possible real-time conversion. Consequently, optimizing this method for integration into a new interactive algorithm is impractical.

Chapter 5

Unreal Engine

Figure 5.1: Unreal Engine logo.

Unreal Engine 5 (UE5)[1] is a widely used modern game engine [59] developed by Epic Games [34]. It is applicable across various industries, including the gaming industry[2], the film industry[3], architectural visualizations[4], simulation[5], and the automotive industry[6]. The engine is renowned for its comprehensive support of key technologies, such as Nanite, and its strong community, extensive marketplace, and widespread adoption across industry sectors. The utilization of this engine is an assignment of this thesis. Unreal Engine technologies, such as Unreal Insights (5.13), are also used as a framework for performance profiling of implemented algorithms. Logo from figure 5.1 was taken from Unreal Engine website[7]

[1]For more information, visit the Unreal Engine website: https://www.unrealengine.com/en-US

[2]For details on Unreal Engine's gaming applications, see https://www.unrealengine.com/en-US/uses/games

[3]For more on its use in film, visit https://www.unrealengine.com/en-US/uses/film-television

[4]For information on architectural applications, refer to https://www.unrealengine.com/en-US/uses/architecture

[5]Learn about simulation usage at https://www.unrealengine.com/en-US/uses/simulation

[6]For insights into its automotive applications, visit https://www.unrealengine.com/en-US/uses/automotive

[7]Unreal Engine branding website: https://www.unrealengine.com/en-US/branding

Krüger et al. [42] present an in-depth case study on how Aachen University transitioned from their in-house VR framework to Unreal Engine, describing its viability for scientific visualization.

Based on over two decades of experience in immersive visualization, the authors identified six key requirements for software frameworks used in such contexts:

- Wide Adoption
- Large Feature Set
- Performance
- Accessibility
- Extensibility/Adaptability
- Flexibility

Unreal Engine was selected as the lab's new development platform due to its strong alignment with these criteria: fully satisfying requirements for adoption, extensibility, and flexibility; partially meeting others under specific conditions [42].

This project uses Unreal Engine **5.4** to support all the necessary plugins (5.19.1). At the time of writing, the latest version of Unreal Engine is 5.5.

5.1 Support of voxel models

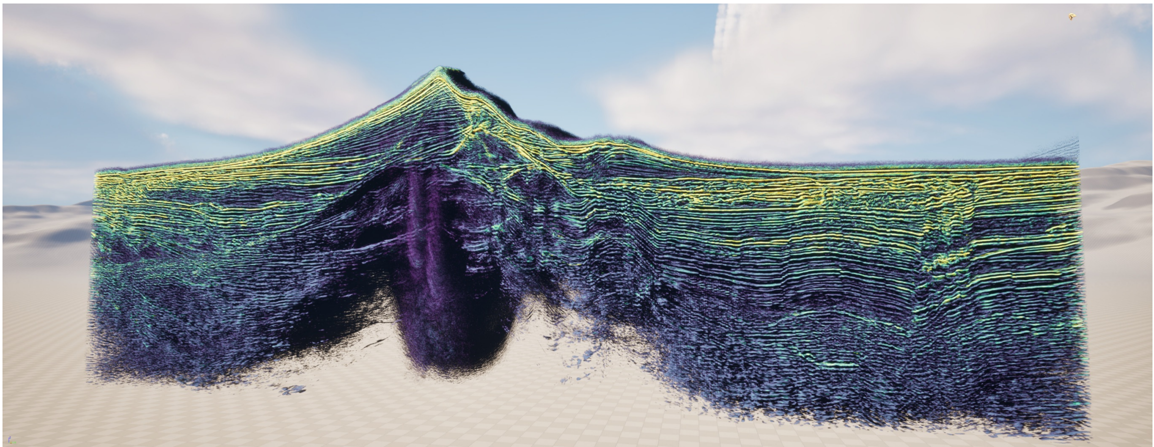


Figure 5.2: Rendering of a large volumetric dataset using Heterogeneous Volume rendering in UE 5.4 taken from Schlüter et al. [74].

Although Unreal Engine is a leading platform for real-time rendering and interactive 3D content, it does not natively support voxel models as physically interactive objects. This includes the absence of built-in systems for editable volumetric data structures, real-time voxel collision, or physics-based interactions such as deformation, destruction, or granular simulations. Developers seeking such functionality must rely on community-maintained plugins or entirely custom implementations.

Schlüter et al. (2025) [74] provide a comprehensive evaluation of various methods for visualizing large-scale volumetric data in Unreal Engine 5, including the use of custom raymarchers, Niagara Fluids [21, Niagara Fluids in Unreal Engine], and the engine's experimental Sparse Volume Texture (SVT) [21, Sparse Volume Textures] and Heterogeneous Volume (HV) [21, Heterogeneous Volumes] systems. While their work demonstrates how to efficiently render massive datasets up to 32k $\times$ 32k $\times$ 16k voxels, as seen in 5.2, they do not address voxel interaction in a physics-driven context. Their approach focuses on in-core visualization and the performance limits of GPU memory, not dynamic manipulation or collision-aware interaction.

The Niagara system [21, Niagara Overview], which is primarily designed for GPU-accelerated fluid simulation, is repurposed in their study to visualize volumetric textures. This uses UE's particle systems, which may render visual effects like fog, but is not a framework for interacting with voxel grids using rigid-body or soft-body physics.

Upon researching all possible support on Subreddits like /r/VoxelGameDev[8] or /r/unrealengine[9], asking developers on official Unreal Engine Discord[10], Unreal Engine Forum[11] and reading official Unreal Engine documentation [21], I have not found any direct official or standardized support for interactive voxel model physics in Unreal Engine. Despite the strengths in rendering, developers must still engineer their solutions for physical interaction with voxel models, whether for deformable terrain, destructible environments, or physically reactive simulations. Such capabilities remain unsupported by the Unreal Engine core API.

Instead, all voxel support is provided by community-developed plugins available on the Fab (5.18). The community plugins (6) and my implementation use Unreal Engine systems described in the following sections.

The following sections will outline the specific implementation and integration strategies used to support voxel meshing, its visualization, and physics-based interaction. Unless stated otherwise, all information and terms were taken from official Unreal Engine Documentation [21]. Specific sections of the documentation are in brackets.

---

[8] Link to a Subreddit discussions: https://www.reddit.com/r/VoxelGameDev/comments/12qceqy/is_unrealunity_efficient_for_a_voxel_game_that/

[9] Link to a Subreddit discussions: https://www.reddit.com/r/unrealengine/comments/19cnl42/voxels_where_do_i_start/

[10] Invite link to Unreal Engine discord https://discord.com/invite/unreal-engine-978033435895562280

[11] UE Forum posts: https://forums.unrealengine.com/t/voxel-support-and-procedurally-generated-terrain/90555; https://forums.unrealengine.com/t/procedural-world-generation-ue-4-27/1284251; https://forums.unrealengine.com/t/node-based-infinite-world-creation-tools-with-biome-support/1966703; https://dev.epicgames.com/community/learning/tutorials/k8am/procedural-voxel-mesh-generation?locale=de-de

5.2 Project

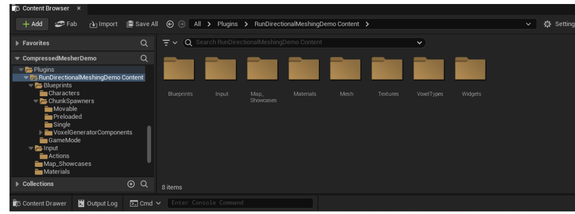
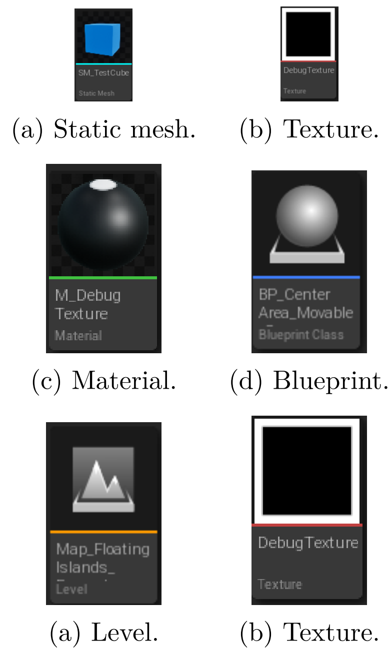


Figure 5.3: Screenshot of Content Browser.

A Project in UE5 contains everything needed for a game or an application, including Assets (5.3). The Assets are organized into folders on a disk, and the same folder structure is reflected in the **Content Browser** [21, Content Browser] within the Unreal Editor. The folder structure of content browser as can be seen in figure 5.3. Arranged Asset folders are located in `Content/` folder and C++ code in `Source/` folder. Each project has a `.uproject` file is used to create, open, or save the project. The name of this bachelor's thesis project is *CompressedMesherDemo.uproject* and is located directly in the attached repository.

5.3 Assets



(a) Static mesh. (b) Texture.

(c) Material. (d) Blueprint.

(a) Level. (b) Texture.

Figure 5.4: Asset examples in the Unreal Engine content browser.

Assets [21, Assets and Content Packs] are building blocks of this project. Any piece of content in Unreal Engine is an asset. Assets can be of many different types. All assets are accessible from the Unreal Engine Content Browser, where each asset has a specific color coding and filter. Examples of assets used in this project can be seen in figure 5.4. Almost all Unreal Engine assets are stored on a disk in a specific file format [21, Working with

Assets] as `.uasset` files. Assets with this format should be manipulated only using Unreal Engine, and the files are saved in `Content/` folder of a project or a plugin. *Content* may also refer to all assets available or used in a project.

5.4 Levels

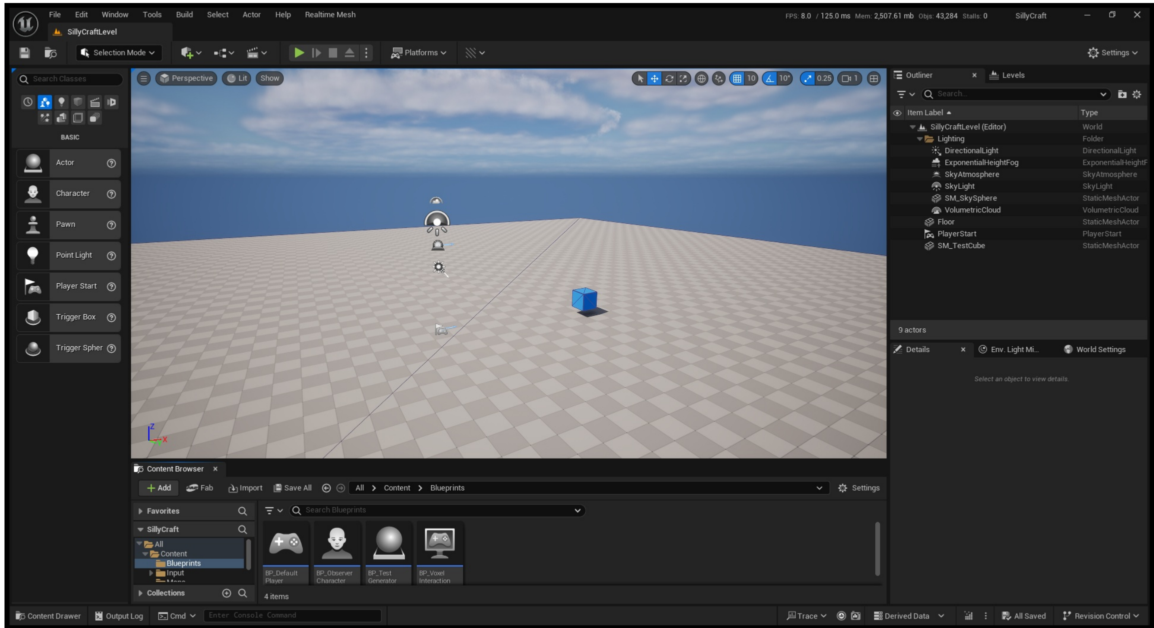


Figure 5.5: Screenshot of UE5 Level Editor interface.

Unreal Engine documentation [21, Levels] describes levels as:

> *„A Level is all or part of your game's 'world'. Levels contain everything a player can see and interact with, like environments, usable objects, other characters, and so on.“*

In Unreal Engine, Levels create different environments or scenes, each saved as a `.umap` file. A Level may be referred to as a map. Usually, a Unreal Engine project contains multiple Levels. Levels may have a transition between them or just be used as showcases, as is the case with this project. A Level usually contains assets in its scene, like terrain, lights, sounds, and a player character. In this project, two types of Levels are used: showcase Levels to demonstrate meshing and voxel generation, and profiling Levels to test the performance of the algorithms as further described in chapter 8.

A Level is still considered an asset and can be created in Unreal Engine. The main interface for creating and modifying Levels in Unreal Engine is Level Editor [21, Level Editor] as seen in 5.5. Upon launching Unreal Engine, the Level Editor is the initial tool presented to the user. Each level has a Play in Editor (PIE) [21, In-Editor Testing (Play & Simulate)] option within a project. This allows a player to be spawned into the Level. The Level Viewport [21, Unreal Editor Interface] serves as the primary visualization interface, displaying the contents of the currently active Level. Opening a project in Unreal Engine

automatically loads its default Level into the Level Viewport. This viewport enables users to inspect and modify the elements within the active Level.

5.5 Blueprints

Figure 5.6: A Blueprint example in the Blueprint Editor.

The Blueprint Visual Scripting system [21, Introduction to Blueprints] in Unreal Engine is a node-based visual programming language designed to implement game functionality. Scripts and user-defined objects created with this system are commonly referred to as **Blueprints**. The interface for creating and editing them is the Blueprint Editor [21, Blueprint Editor Reference], shown in Figure 5.6.

Blueprints enable rapid iteration, accessibility, and scripting of game behaviors, making them ideal for designers and prototyping. They also provide a safer memory model and enhanced discoverability of APIs and assets. In this project, Blueprints serve as an interface between C++ code and Unreal Engine systems, enabling player control, collider interaction, and event handling such as raycasting. Configurable properties can also be exposed in Blueprints to influence underlying C++ behavior.

5.5.1 C++

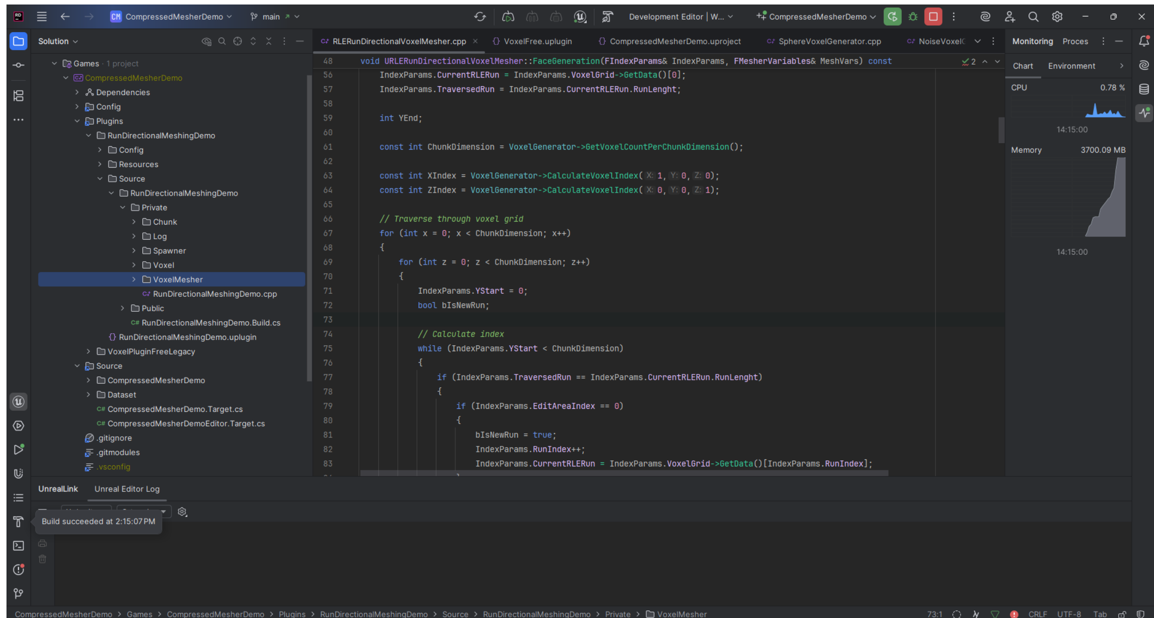


Figure 5.7: Project screenshot in Rider IDE.

Unreal Engine supports Blueprint and C++ [21, Coding in UE: Blueprint vs. C++], and most projects benefit from using both. While Blueprint is ideal for scripting behavior and interaction, C++ excels at building performant core systems, accessing low-level engine features, and enabling advanced debugging and extensibility. Programming in C++ allows better scaling and collaboration due to its text-based nature. JetBrains Rider, selected via a university student license, was used for C++ development, with most of the project's core systems implemented in C++.

C++ outperforms Blueprint because it compiles to native machine code, whereas Blueprint executes as bytecode on a virtual machine. This performance difference is especially significant in core systems, tight loops, extensive data processing, tick-heavy classes, and multi-threaded scenarios, which Blueprint does not support. To optimize Blueprint performance, it's recommended to use timers or delegates instead of Tick and to profile performance with Unreal Insights (5.13) before converting systems to C++.

The optimal workflow is to implement core systems in C++ and expose them to Blueprints using metadata specifiers like `UPROPERTY(BlueprintReadWrite)` or `UFUNCTION(BlueprintCal`
This exposure is used as the interface between C++ and Blueprints. Alternatively, static utility functions can be created via `UBlueprintFunctionLibrary`. When converting from Blueprint to C++, references may require updating, which can be automated using Core Redirects.

When working with C++, Unreal Engine `.sln` solution must be generated from the `.uproject` file and the editor should be launched from an IDE. JetBrains Rider IDE[12], activated via a university student license, was used for development as can be seen in figure 5.7. Most of the implementation of the bachelor's thesis project was done in C++. The

[12]Link to Rider website: https://www.jetbrains.com/rider/

following classes may be used in both C++ and Blueprints and serve as building blocks upon which this project is built. Documentation for all the Unreal Engine classes can be found in [21, Unreal Engine C++ API Reference].

5.5.2 Objects

In Unreal Engine, Objects [21, Unreal Engine Terminology] are the most fundamental class, serving as building blocks that provide essential functionality like garbage collection. Almost everything in the engine inherits from the base `UObject` class [21, Objects]. `UObject` allows metadata exposure via `UProperty` for Unreal Editor integration, garbage collection, and serialization for saving and loading data in C++ code.

5.5.3 Actors

In Unreal Engine, an Actor [21, Actors] is any object that can be placed in a Level, such as a Camera, Static Mesh, or player start. Actors support 3D transformations—location, rotation, and scale—and can be created or destroyed during play using C++ or Blueprints. All actors derive from the `AActor` base class in C++. During a Level creation, you place and arrange Actors within a Level to build the scene and add scripts to control their behavior as stated in [21, Actors and Geometry]. Adding new Actor instances into a Level is known as spawning. Chunks are spawned into a Level as Actors through Chunk Spawners (7.6). A guide on creating an Actor spawner can be found in [21, Spawning and Destroying an Actor]. Ticking refers to how Actors are updated in Unreal Engine. All Actors can be ticked at user-defined times; by default, the tick interval is each frame.

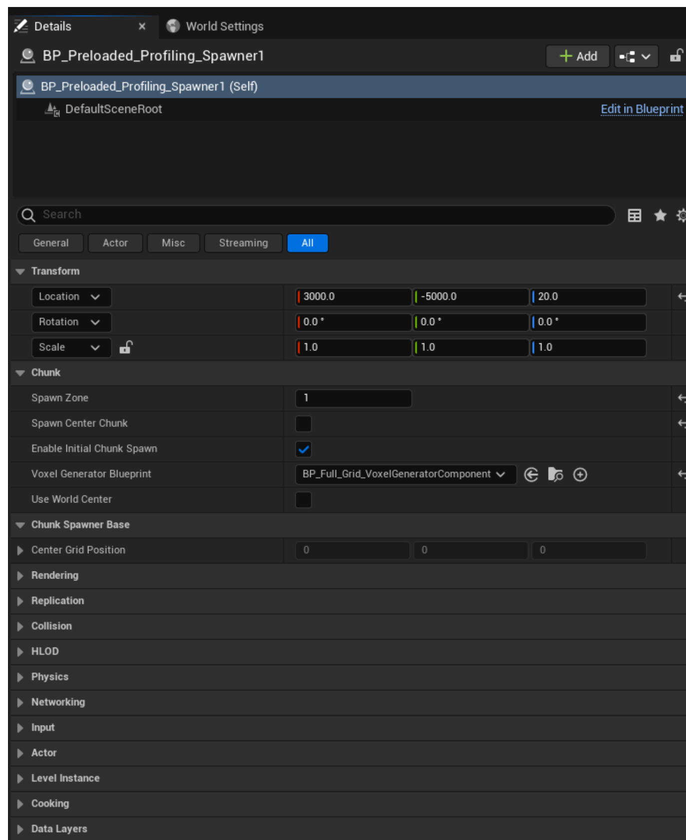


Figure 5.8: Example of actors with components.

Actors can also be used as containers that hold Components[21, Components]. Components are special Objects that define behavior and functionality, such as movement, rendering, audio, and collision. Components must be attached to an Actor and cannot exist independently. There are three main types: `UActorComponent` (abstract logic like inventory or input), `USceneComponent` (adds world position, rotation, and scale), and `UPrimitiveComponent` (adds a visual or physical presence, like meshes or collision volumes). An Actor's location and transformation come from its root *SceneComponent*. Components created as part of an Actor are automatically registered, but manual registration is required for those created during gameplay. Components can be used to give Actors specific capabilities—for example, emitting light, rotating, or playing sounds. Voxel Generators are dynamically added to Chunk Spawner Actors during gameplay as `UActorComponent`. When Chunk Actor is spawned, it also has its Mesh Component. An example can be seen in figure 5.8.

5.5.4 Pawns

A Pawn [21, Pawn] is a type of Actor in Unreal Engine that represents a controllable entity in the game, either by a player or AI, and serves as their physical presence in the world, including location, rotation, collisions, and interactions—even if it lacks a visible mesh. When controlled, it is "Possessed"; otherwise, it is "Unpossessed". Each Controller typically manages only one Pawn at a time, and newly spawned Pawns are not automatically possessed.

A Character [21, Characters] is a type of Pawn designed for player control, with built-in movement features like walking, running, jumping, flying, and swimming. It includes components like `CapsuleComponent` for collision and a `CharacterMovementComponent` that handles movement without using physics simulation. This component allows speed, gravity, and friction customization. This project uses custom Characters to test interactions with voxel models. A detailed description of the interaction implementation is described in 7 chapter.

5.5.5 GameMode and player inputs

The GameMode is a blueprint type in Unreal Engine and defines core gameplay rules, including player spawning, pause behavior, and victory conditions [21, Unreal Engine Terminology]. While the default GameMode can be specified globally via the Project Settings [21, Project Settings], it may also be overridden on a per-level basis through the World Settings [21, World Settings]. However, only one GameMode can be active per level at any given time. This project leverages custom GameModes for each level to initialize characters and PlayerControllers that support voxel-based interaction.

The RDM plugin includes two distinct GameModes, one of which is dedicated to terrain generation in the vicinity of a spawned player character and the other only allows interactions with voxel model. The latter is used in almost all levels. Each GameMode references a custom PlayerController. PlayerController is type of blueprint that acts as an intermediary between the player and their in-game representation (Pawn or Character), translating user inputs into corresponding gameplay actions [21, Player Controllers]. It is tasked with interpreting input events and orchestrating the behavior of the associated Pawn.

In this implementation, all player inputs are processed via Unreal Engine's Input Mapping Context system, part of the Enhanced Input framework [21, Enhanced Input]. The supported input mappings include:

- Jump — triggered by pressing the Spacebar.
- Movement — controlled using the W, A, S, and D keys.
- Voxel Removal (Pick) — initiated with the left mouse button.
- Voxel Placement — initiated with the right mouse button.
- View Rotation — controlled through mouse movement.

I implemented these inputs and custom PlayerControllers as Unreal Engine Blueprints in a folder `Plugins/RunDirectionalMeshingDemo/Content/Input`.

5.6 Static Mesh

Static Mesh [21, Static Mesh] are fundamental assets for creating world geometry in Levels developed with Unreal Engine. The meshes described in section 3.9 are usually saved as 3D models and typically created in external modeling applications such as Blender[13], and are then imported into the Unreal Editor through the Content Browser. Most Levels created in Unreal Engine consist of Static Meshes, typically in the form of Static Mesh Actors.

[13] Blender website: https://www.blender.org/

The mesh is considered static if its geometry does not change, but the Actor itself can be moved or altered during gameplay. Static Mesh Actors are commonly used to create game worlds or other environments, and Unreal Engine includes default Static Mesh Actors such as Cube, Sphere, Cylinder, Cone, and Plane. Additionally, users can import custom Static Mesh Actors created in other 3D applications. Since Static Meshes are cached in video memory, they offer flexibility in translation, rotation, and scaling, and can be more complex than other types of geometry. The meshes can be procedurally generated as described in section 5.11. Unreal Engine also provides Skeletal Meshes [21, Skeletal Meshes] for animated meshes, which are not used in this bachelor's thesis.

5.7 Materials

Unreal Engine documentation [21, Materials] describes materials as:

> *„Materials in Unreal Engine define the surface properties of the objects in your scene. In the broadest sense, you can think of a Material as the 'paint' that is applied to a mesh to control its visual appearance."*

Materials [21, Materials] are an asset that integrates directly with the render engine of Unreal Engine. Materials tell the engine how light should interact with a surface in a scene through properties such as color, reflectivity, bumpiness, transparency, etc. Properties are calculated from textures, node-based material expressions, and inherent material settings.

Material assets are created in a visual scripting interface called the Material Editor [21, Material Editor Guide]. As described in the documentation [21, Essential Material Concepts], Material node graphs are silently translated into shaders behind the scenes. Shaders are programs that define how each vertex or pixel should be rendered on GPU hardware in a rendering pipeline. In the Unreal Engine, the shader code is written in High Level Shading Language (HLSL), but it is generated from Material nodes. The generated code is then converted to Assembly Language instructions that the GPU hardware can execute and used in the rendering pipeline.

In this thesis materials used to render voxel surface are stored in Unreal Engine Data tables [21, Data Driven Gameplay Elements] are a type of Blueprint asset designed for data-driven elements. The data are used in the implementation to store various information about a voxel. More details are described in the future section 7.4.

5.8 Physics

Chaos Physics [21, Physics] is Unreal Engine's lightweight, next-gen physics simulation system designed for high performance and flexibility in game development. It supports real-time destruction, networked physics, fluid and cloth simulation, physical animation, ragdolls, vehicles, hair and flesh physics, and tools for visual debugging and controlling simulations through physics fields.

5.9 Raycasting

Raycasts [21, Traces with Raycasts], also known as Traces, are used in Unreal Engine to trigger an event. These send out invisible rays that detect geometry between two points and return information about whether geometry is hit, and return what was hit so it may

be processed accordingly. Traces can be performed in two main ways: either by detecting any object in the scene, or by using a specific Trace Channel that only interacts with objects configured to respond to that channel through their collision settings. Traces can detect either a single hit (Single Trace) or multiple hits along the ray's path (Multi Trace). Different ray types can also be specified, for example: a box, a capsule, or a sphere. Line Traces are used to intersect with procedurally generated mesh geometry during player interaction input events.

5.10 Collisions

Collisions [21, Collision Overview] in Unreal Engine use two types of collision: simple and complex [21, Simple versus Complex Collision]. Simple collision refers to basic geometric shapes such as boxes, spheres, and capsules, while complex collision uses the actual triangle mesh of an object. By default, Unreal Engine generates both types, and the one used depends on whether a simple or complex query is being executed. The editor treats cameras differently based on the simulation mode: Play In Editor camera follows Pawn collisions settings, but Simulate in Editor allows the camera to pass through everything.

As stated in [21, Setting Up Collisions With Static Meshes] and [21, Simple versus Complex Collision], each object in Unreal Engine, has a collision type and response rules that determine how it interacts with others (either blocking, overlapping, or ignoring them). The same logic applies to raycasting using trace channels. For a collision to occur, both objects must block each other; if one is set to overlap or ignore, the interaction follows the less restrictive rule. Overlap events only happen if both objects have `Generate Overlap Events` enabled, and to trigger Blueprint collision events, `Simulation Generates Hit Events` must also be active. Static Meshes require manual collision setup using the Collision menu, where simple shapes can be added and adjusted, or more complex collisions can be generated using K-DOP or convex decomposition for better accuracy. Once collision is set, enabling `Simulate Physics` allows meshes to respond to forces like gravity or push, and customizable `Collision Presets` control how they interact with various elements in the game world.

5.11 Procedural Mesh Component

Procedural Mesh Component (PMC) [21, UProceduralMeshComponent] is an official experimental support for Unreal Engine's generation of procedural meshes. It allows for specifying custom triangle mesh geometry. It has functions that accept vertices, triangles, normals, etc., like `CreateMeshSection`, which will create procedurally generated meshes. Thanks to this it is known what information about a mesh we need per procedurally generated quad at minimum. As tested for correct lighting and shading, Unreal Engine needs information about the normals of the quads. This was described in detail in the previous section about T-Junction 4.2.1. There are community tutorials[14] on how to operate this tool; however, it is poorly documented. Even if no support for voxel models was found, creating dynamic meshes and colliders from those meshes in Unreal Engine is possible. Instead of this component, a community library described in section 5.19.1 was used.

[14] Community tutorial links: https://dev.epicgames.com/community/learning/livestreams/W7y/getting-started-with-procedural-mesh-generation-inside-unreal

5.12 Custom logs

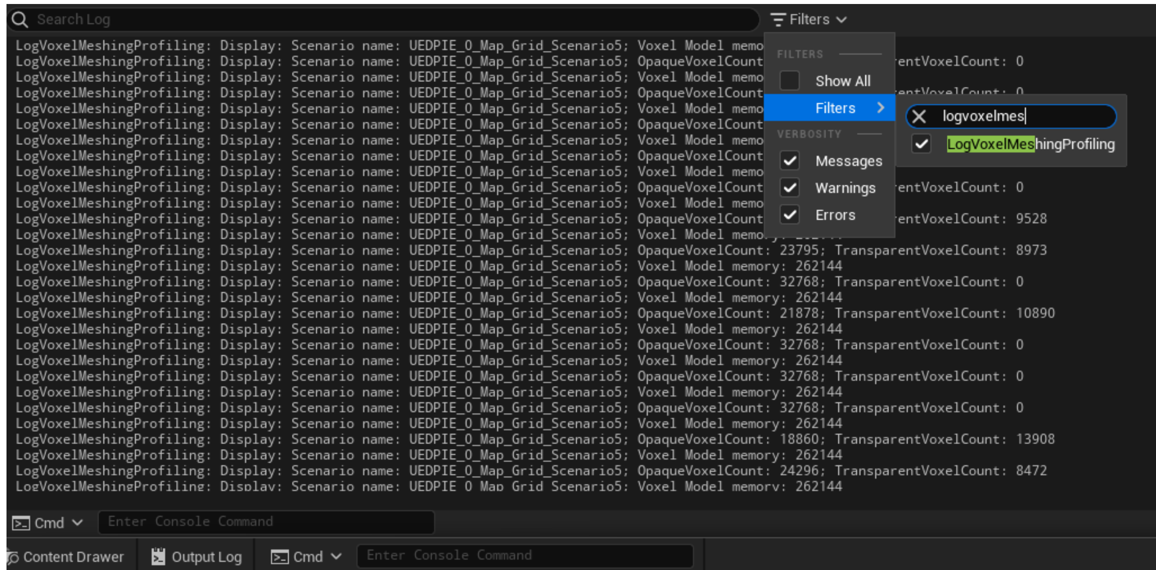


Figure 5.9: Example of Output log in the editor.

Unreal Engine provides a logging system [21, Logging in Unreal] that supports categorized and verbosity-controlled output, which can be used to track runtime information and store diagnostic data. Developers can define custom log categories and assign default verbosity levels using the `ELogVerbosity` enumeration. Logged messages are visible in the `Output Log` window (figure 5.9) of the Unreal Editor and are also saved to text files in the `Saved/Logs` directory. The system supports runtime filtering and redirection of logs, making it suitable for structured data collection.

In this project, a custom log category named `LogVoxelMeshingProfiling` was declared in the header file `VoxelMeshingProfilingLogger.h` using the macro `DECLARE_LOG_CATEGORY_EXTERN(Lo` `Display, All);` and defined in the corresponding source file with `DEFINE_LOG_CATEGORY(LogVoxelMeshi` This category is used by a utility class to log profiling data such as memory usage, vertex count, and voxel ratio. The recorded log files are used in later chapter (8) to support performance analysis of voxel meshing implementations.

5.13 Unreal Insights

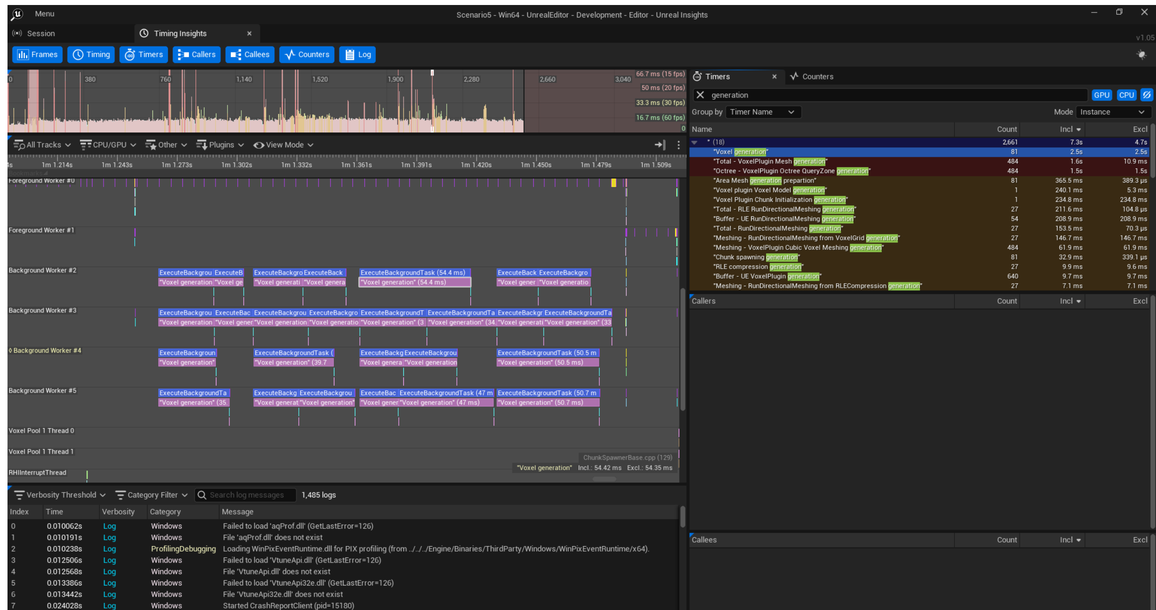


Figure 5.10: Screenshot of Unreal Insights.

Unreal Insights [21, Unreal Insights] is a tool for capturing and analyzing performance data from Unreal Engine projects. It helps identify areas needing optimization by recording detailed trace events during runtime. The system consists of three key parts:

- **trace events**: System to define and record specific actions.
- **Unreal Trace Server**: System to capture and save recorded data.
- **Unreal Insights**: Application to visualize and analyze recorded data.

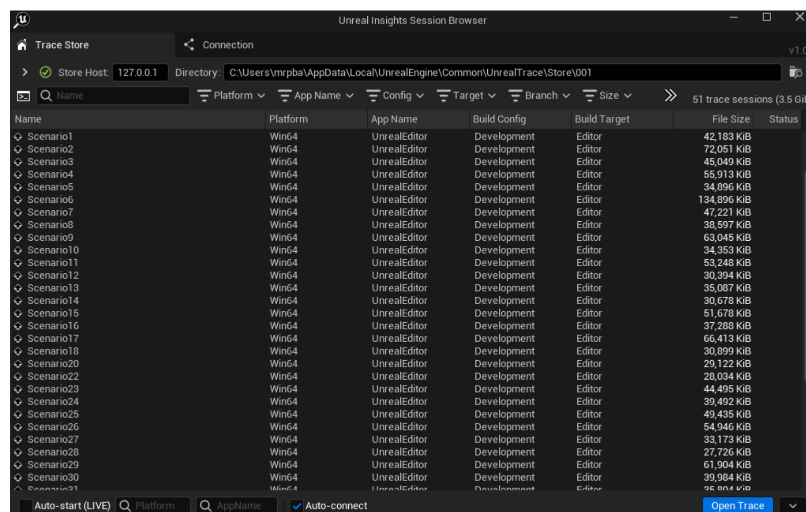


Figure 5.11: Editor widget.

The data is stored in `.utrace` files, with any additional info saved in `.ucache` files. Unreal Insights can be launched directly from the Editor through the `Trace/Insights Status Bar` Widget as can be seen in figure 5.12. Depending on the build and OS, different workflow options are available. A compiled version of Unreal Insights is included if the binary engine build is used. Otherwise, it is built from source using an IDE and Unreal Build Tool. *Trace* is Unreal's logging system for monitoring runtime behavior. The Unreal Trace Server runs quietly in the background, listening for data on port 1981 and saving it in a folder watched by Unreal Insights. It has two components: the Trace Recorder, which captures the data, and the Trace Store, which manages and exposes stored sessions.

The server can be configured to watch specific directories, such as a Downloads folder, and store trace data persistently across reboots. From Unreal Insights, you can manage trace storage locations and watch folders using the Manage Store Settings menu. The Session Browser in Unreal Insights lets you view and manage live and recorded trace sessions. Live sessions appear in real time and are marked as `LIVE`. Live sessions can be loaded and analyzed just like saved sessions. Unreal Insights also includes several specialized analysis tools:

- **Timing Insights**: Shows CPU and GPU activity, helping you visualize performance over time.
- **Memory Insights**: Tracks memory allocations and usage patterns.
- **Networking Insight**: Helps debug and optimize network traffic
- **Slate Insights**: Focuses on UI performance in Slate and UMG.
- **Asset Loading Insights**: Analyzes how long different assets take to load.
- **Cooking Insights**: Breaks down the time spent cooking game content into packages.

All features can be accessed through the menu in the Insights interface, which also allows import of tables, the opening of trace files, and the auto-loading of live sessions. Macros and command-line options can tailor how a project logs and outputs telemetry data for deeper customization. Timing Insights are used in this bachelor's thesis to measure and compare the performance of different algorithms. Instead of memory insights, custom logs are already used to measure a persistent memory of voxel models. Timings are measured from macros placed in the code, and results are exported to CSV files and processed into graphs. Further information on measured performance can be found in chapter 8.

5.14 StartFPSChart

Unreal Engine offers built-in console commands for VR performance profiling [21, VR Performance Testing] that are also effective in general interactive 3D applications. Although VR is not the focus of this thesis, its profiling tools, designed to ensure consistent frame pacing and responsiveness, are well suited for exporting real-time frame rate performance evaluation.

The `StartFPSChart` and `StopFPSChart` commands were used to collect time-bucketed frame rate data, producing CSV reports that summarize frame time distribution. While intended for monitoring VR frame targets, these reports are equally useful for identifying frame pacing issues in non-VR contexts. However, in this thesis, the bucketed data were

not used, instead, only the aggregated log output, produced alongside the detailed CSV data, was used for performance evaluation, as it provides a clearer summary of average frame time and frame rate without requiring additional post-processing.

All measurements were conducted in `Standalone Game` mode to avoid editor-related overhead. Once the voxel environment was fully initialized, the recording is enabled using `StartFPSChart`, a player movement was performed. After a sufficient duration, the results are saved in the **Saved/Logs** directory by executing `StopFPSChart` command.

This methodology is applied in chapter 8 to compare how meshes produced by different voxel meshing algorithms affect frame rate performance.

5.15 Nanite

Figure 5.12: Example of Nanite activated on an Actor with different views. Left is a mesh generated from Nanite. Right is the original Static Mesh. Image was taken from [21, Nanite Virtualized Geometry].

Nanite [21, Nanite Virtualized Geometry] is Unreal Engine 5's virtualized geometry system for real-time rendering of high-detail assets, removing limits on polygon count, draw calls, and manual LODs. It decomposes meshes into compressed triangle clusters streamed based on camera view, with visibility and screen-space error guiding fine-grained LOD and aggressive occlusion culling. Performance depends on screen resolution, not geometric complexity. Nanite supports static meshes and, experimentally, skeletal meshes and foliage, but does not support translucency, deformation, or forward rendering. Unsupported features use fallback meshes generated from the original mesh with customizable simplification settings. Although not used in this thesis, Nanite's potential for procedural mesh generation

is discussed in the conclusion 9. More information is available in a SIGGRAPH YouTube video[15].

5.16 Unreal Engine Plugins

Plugins [21, Plugins] in Unreal Engine are collections of code and data that add features and functionality to a Project or extend the Editor with new tools and interfaces. They can be enabled or disabled per project and are structured similarly to projects, making them easy to share or distribute. Plugins must be included in a `.uproject` file to be used. Plugins can be distributed on Fab, the Unreal Engine Marketplace (5.18).

Plugin Editor can be opened from Unreal Editor. It shows all installed plugins, which can be enabled or disabled using a checkbox. Categories can be used to filter Plugins. Each Category shows how many plugins it contains. Each plugin shows its name, icon, version, description, author, and whether it's enabled. Some changes to the Plugins may require a restart of the Unreal Editor. Plugins with code have a `Source/` folder with the code, a `Binaries/` folder with compiled files, and an `Intermediate/` folder for temporary files. They can also have a `Content/` folder for assets. Config files are in a `Config/` folder; the names depend on whether the plugin is for the engine or a project.

Unreal Engine must find Plugins in specific folders. Engine plugins go in the `Engine/Plugins` folder. Project plugins are in the `Project/Plugins` folder. Unreal Engine finds plugins using `.uplugin` files, which describe the plugin in JSON format using a plugin descriptor. Only project-specific files are used in this bachelor's thesis. A new plugin can be created from the Plugin Editor or an IDE. If a plugin has code, it will be added to the IDE project and compiled with it. Plugins can have multiple modules, for runtime or editor tools. They can also define Unreal classes and structures. Plugins can depend on others, but only in one direction: lower-level code can't depend on higher-level code. For example, engine plugins can't depend on project plugins. Engine plugins are built into Unreal and work in any project.

Plugins can also include assets if allowed in the `.uplugin` file. Project plugins are loaded when the engine or editor starts. If they have code, it's added to a C++ project. If they only have binaries, they'll still load. Config files aren't packaged by default and must be copied manually. A package can be created using the Plugin Editor to share a plugin, or files can be directly copied from the `Plugins/` folder. Plugin descriptor files (.uplugin) are in JSON and include info like version and settings. Each plugin should have an icon called Icon128.png in a `Resources` folder.

The thesis includes my plugin demonstrating Run Directional Meshing (7), further refered to as RDM plugin, and the legacy Voxel Plugin Free. These are located in the `Plugins/` folder of the Project. They are not intended for public distribution but can be downloaded from the files attached to this thesis and imported into other Unreal Projects for use. The current implementation is meant as a tech demo upon which further implementation should be built and improved before distribution.

[15] Nanite YouTube video link: https://www.youtube.com/watch?v=eviSykqSUUw&list=PLuAAwN0nwIkotcBiwm8u-pks0zMHi5mUo

5.17 Unreal Engine Modules

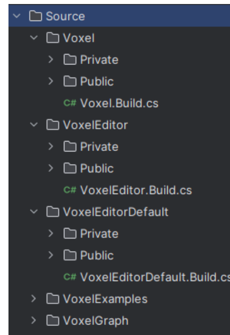


Figure 5.13: Example of module usage inside Unreal Engine plugin.

Modules [21, Unreal Engine Modules] are the building blocks of Unreal Engine's software architecture. They help organize code into isolated, manageable, standalone code units, such as runtime features, editor tools, or libraries in a UE Project. Every project and plugin has a default primary module, but additional ones can be created to improve code organization and build performance. Modules support encapsulation, code reuse, and reduced compile times by compiling only changed parts. They follow the Include What You Use (IWYU) principle, so unused modules are excluded from compilation, which is done using a dependency graph. Only modules inside this dependency graph are compiled. Modules can also be conditionally included based on platform or runtime needs, and their loading behavior is customizable in `.uproject` or `.uplugin` files.

A new directory named after the module is created when creating a new Module. Each module has Private and Public subfolders for internal and external headers. Source files are in the Private folder. Source files in a Private folder should mirror header files in a Public folder to maintain the same structure. `[ModuleName].Build.cs` file is created in a module's root. This file makes it possible for the Unreal build system to discover this module, and it defines dependencies via `PrivateDependencyModuleNames` and `PublicDependencyModuleNames`. Public dependencies for modules are accessed in public headers, and private for internal use. A file `[ModuleName]Module.cpp` is added to the Private folder to provide methods for starting up and shutting down a module. To use a Module in a Project, an entry must be added to `.uproject` or `.uplugin` files. All modules on which this project depends can be found in those files. After changes to the module dependency structure are made, the project files must be regenerated. Modules can be added manually or created automatically using IDE tools. Only primary modules were created for this project, but some used community libraries contain code separated into modules, as can be seen in figure 5.13.

5.18 Fab

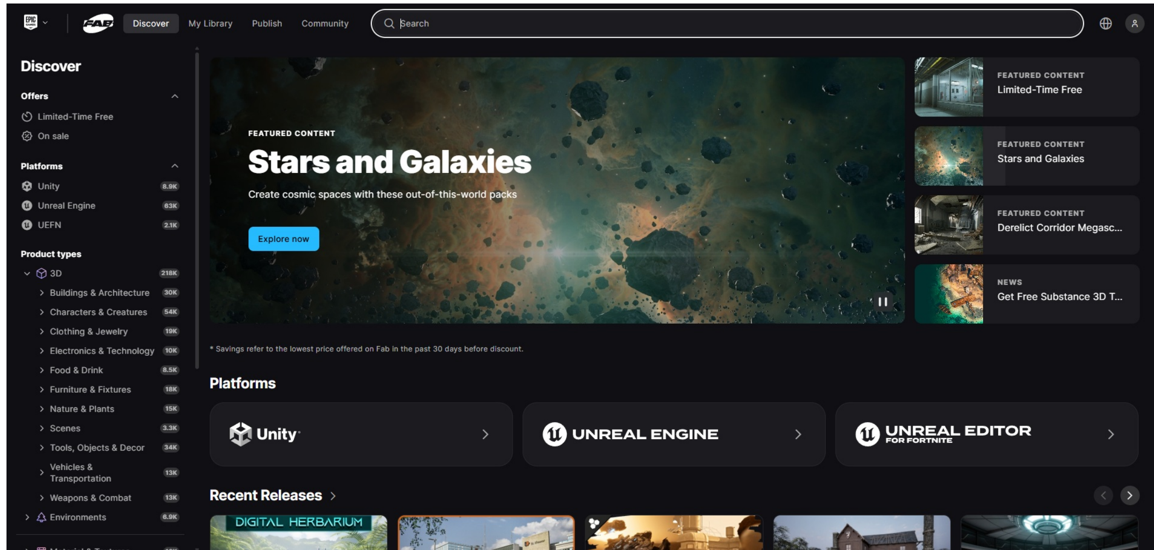


Figure 5.14: Fab website screenshot.

Fab [16] is the unified digital asset distribution platform. As stated in Epic Games blog [22], it replaced the Unreal Engine Marketplace, Sketchfab Store, and offers additional hosting of Quixel Megascans. It supports creators who use Unreal Engine, Unity, and UEFN, offering a wide range of high-quality, real-time–ready assets such as environments, animations, audio, VFX, and plugins.

Fab is critical for this thesis because it is the primary source of Unreal Engine-compatible community plugins, including the ones evaluated and integrated here, specifically Fast Noise Generator 5.19 and RealTimeMesh Component [82]. All competitive algorithms analyzed in the next chapter 6 are required to be freely available and distributed on Fab to ensure fair comparison and compatibility with Unreal Engine 5. Fab's distribution model ensures accessibility, consistency, and long-term support for the plugins used in voxel meshing and profiling.

A screenshot of the Fab interface is shown in figure 5.14.

5.19 Fast Noise Generator

Fast Noise Generator[17] published by Rockam[18] is a Plugin available on Fab. According to the documentation [65], the Fast Noise Generator (FNG) plugin is a noise generation tool for Unreal Engine that offers both C++ and Blueprint compatibility. Developed by Víctor Hernández Molpeceres, it serves as an Unreal Engine wrapper around Auburn's open-source FastNoise library [69]. Auburn's library generates noise described in section 3.5.1. The plugin builds upon the work of Auburns, whose FastNoise library provides the

[16] Link to official Fab website: https://www.fab.com/
[17] Fast Noise Generator link: https://www.fab.com/listings/c1d444fc-54cc-4f11-8a4a-c0c41112a321
[18] Link to Fab publisher details: https://www.fab.com/listings/c1d444fc-54cc-4f11-8a4a-c0c41112a321

core noise generation functionality and is available for versions UE4.22 and beyond. It provides a wide array of noise algorithms suitable for procedural content generation.

In this project, the FNG plugin generated height values that serve as a replacement for voxelization to generate detailed semi-random voxel models. It supports complete configuration through C++ and Blueprints, making it highly accessible for various workflows. A reference to the Fast Noise Wrapper must be created before using the FNG. This uses the `CreateDefaultSubobject` function in C++, or the `Construct Object from Class` node in Blueprints. Once instantiated, the noise generator can be configured using the Setup Fast Noise function, which allows specification of key parameters such as:

- **Noise Type**: Defines the algorithm used to generate noise.
- **Seed**: Controls the deterministic randomness of the noise.
- **Frequency**: Affects the coarseness of the generated pattern.
- **Interpolation**: Determines the smoothing method between noise values (e.g., Linear, Hermite, or Quintic).
- **Fractal Settings**: Includes options such as fractal type, octave count, lacunarity (frequency multiplier), and gain (intensity multiplier).
- **Cellular Noise Options**: Includes settings like jitter (displacement of cell points), distance function (e.g., Euclidean, Manhattan), and return type.

Once configured, noise values can be retrieved using the `GetNoise2D` or `GetNoise3D` functions. These values may used to generate heightmaps or other procedural data layers. For C++ integration, the `FastNoiseGenerator` and `FastNoise` modules must be added to the public dependencies in the project's `Build.cs` file. Additionally, `FastNoiseWrapper.h` must be included in any source files where the noise generator is utilized. All relevant functions and properties are exposed to Blueprints under the `Fast Noise` category, making the plugin fully accessible without writing C++ code.

5.19.1 RealTimeMesh Component

The Realtime Mesh Component (RMC) is a plugin for Unreal Engine that enables the rendering of dynamically generated or modified mesh data at runtime. It is available in a free version[19] and a paid version[20]. Due to the open source nature of this thesis, only the free version is used.

Core Features include:

- Efficient rendering with lower memory usage (50–90%) and reduced CPU time (30–90%) compared to ProceduralMeshComponent.
- Support for up to 8 UV channels with both normal and high precision.
- Full collision support for both static and dynamic objects.
- Level of Detail (LOD) support, including up to 8 levels and dithering.

[19]Link to RMC free version: https://www.fab.com/listings/bb2e4fbb-617c-41d3-aac6-e181eddf8b3b
[20]Link to RMC paid version: https://www.fab.com/listings/c04e6977-a3bd-4e24-b0a2-0b696bee90da

- Full NavMesh integration and asynchronous collision updates.
- Distance field support and static/dynamic draw paths for performance tuning.

Paid Version features:

- OBJ mesh importing
- Lumen integration (cards and distance fields).
- Mesh optimization tools.
- Conversion between DynamicMesh and RMC.
- Upcoming support for Nanite and spatial streaming (octree/quadtree).
- Mesh replication across networked clients

This plugin was chosen for upcoming Nanite support. As described in section 5.15, Nanite could optimize meshes by dynamically creating LODs during runtime on a GPU. Thus, making additional LOD calculations unnecessary to perform on a CPU. If voxel models are loaded and not procedurally calculated, for example, sent over the network or loaded from disk, this could unload additional meshing performance of LODs from CPU to GPU. For this reason, the support of LODs is out of the scope of this bachelor's thesis.

According to documentation [82], it is optimized for performance and memory usage, offering significant advantages over standard Unreal components such as the ProceduralMeshComponent and DynamicMeshComponent. RMC supports essential features including static and dynamic collision handling, asynchronous collision updates, up to eight UV channels, high-precision vertex attributes, and full integration with Unreal Engine's Level of Detail (LOD) and NavMesh systems. It also provides tools for converting between various mesh types, including StaticMesh, ProceduralMesh, and DynamicMesh, making it highly adaptable to different rendering scenarios. Internally, RMC represents mesh data using an indexed triangle list consisting of two core components: the vertex buffer and the index buffer. The vertex buffer holds all unique vertices with position, normal, tangent, UV coordinates, and vertex color attributes. The index buffer defines the triangle connectivity by referencing groups of three vertices. To ensure correct rendering under Unreal's backface culling system, the vertex indices must be ordered counter-clockwise. To support the grouping of triangles within a mesh, RMC introduces the concept of Polygroups.

A Polygroup is an identifier assigned to each triangle, enabling logical grouping for separate rendering or processing. Polygroups are stored in a dedicated stream, with one entry per triangle, and can be used to partition a mesh into contiguous triangle clusters. This allows for selective rendering of sections, optimizing performance and material application. Polygroups are expected to be sorted contiguously for optimal rendering. The data structure within RMC is built around the concept of streams. Each stream stores a single attribute type, such as position or color, and streams are grouped into sets called Stream Sets. These sets can be linked via Stream Linkages to ensure that changes to one stream (e.g., position) are consistently reflected across related streams (e.g., normals or UVs). Developers can manipulate these data structures using tools like `StreamBuilders` and the `MeshBuilder`, which simplify common mesh construction workflows. The component system of RMC includes `URealtimeMeshComponent` for rendering by attaching the component to an Actor and `URealtimeMesh` for managing shared mesh data and collision structures.

Depending on the use case, different mesh implementations are available. RMC supports a hierarchical mesh layout for rendering optimization using Level of Detail (LOD) groups, section groups, and individual sections. Each has corresponding render-thread proxies to ensure high-performance rendering without blocking the game thread. Sections can be configured as static or dynamic, depending on how frequently their data is updated. Support for LOD is out of the scope of this bachelor's thesis and is not part of the assignment.

However, in this library, there are multiple **limitations and bugs**, as discussed in the previous section, that arise from the use of RMC polygroups. Specifically, this library requires that vertices associated with the same material type be inserted consecutively within the vertex sequence. Consequently, quads must be added to the data streams continuously based on its material. As can be seen on Fab, Realtime Mesh – Core only supports Unreal Engine version 5.4. Version 5.5 is not supported by this plugin as of writing this thesis. For this reason, this project also only supports version 5.5. Only the paid version of this plugin supports the current UE version. There is a bug where the editor may trigger a debugger breakpoint when stopping Play-In-Editor, if collision generation is still in progress. The root cause appears to be that the collider is created on an asynchronous thread. Unfortunately, no method is currently available to wait for this thread's completion or to safely cancel the collision generation. As a result, the thread may become invalid, leading to incorrect memory access and triggering the debugger. This issue does not halt execution; the process can be safely continued after the breakpoint is triggered. The event will only result in a console log entry. Because of the collision generation, a workaround was made to remove collision when regenerating the mesh. The issue is described on a Unreal Engine forum[21]. I reported and discussed the bugs on an official RMC Discord community[22].

[21]Link to the collision issue on Unreal Engine Forum: `https://forums.unrealengine.com/t/collision-data-isnt-removed-after-clearing-a-section-if-buseasynccooking-true/268178`

[22]RMC discord link: `https://discord.gg/KGvBBTv`

Chapter 6

Competitive Algorithms in Unreal Engine

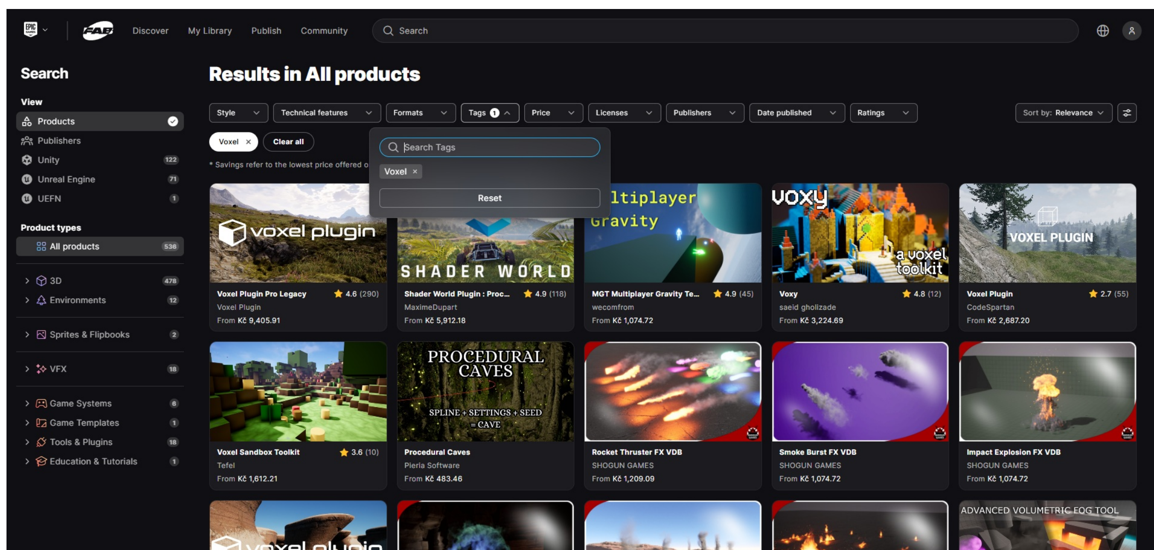


Figure 6.1: Fab screenshot with a voxel filter.

In the context of this thesis, a competitive algorithm refers to a community-developed, freely available plugin for Unreal Engine distributed via the Fab platform. The thesis assignment includes a request for finding competitive algorithms.

Due to proprietary licensing restrictions, only freely accessible algorithms were considered. Within the Unreal Engine ecosystem, the majority of meshing solutions are commercial plugins or tutorials that are not distributed through the Fab platform. Such cases were therefore excluded from this thesis.

For a plugin to be classified as a *competitive algorithm* in this thesis, an implementation must meet the following requirements:

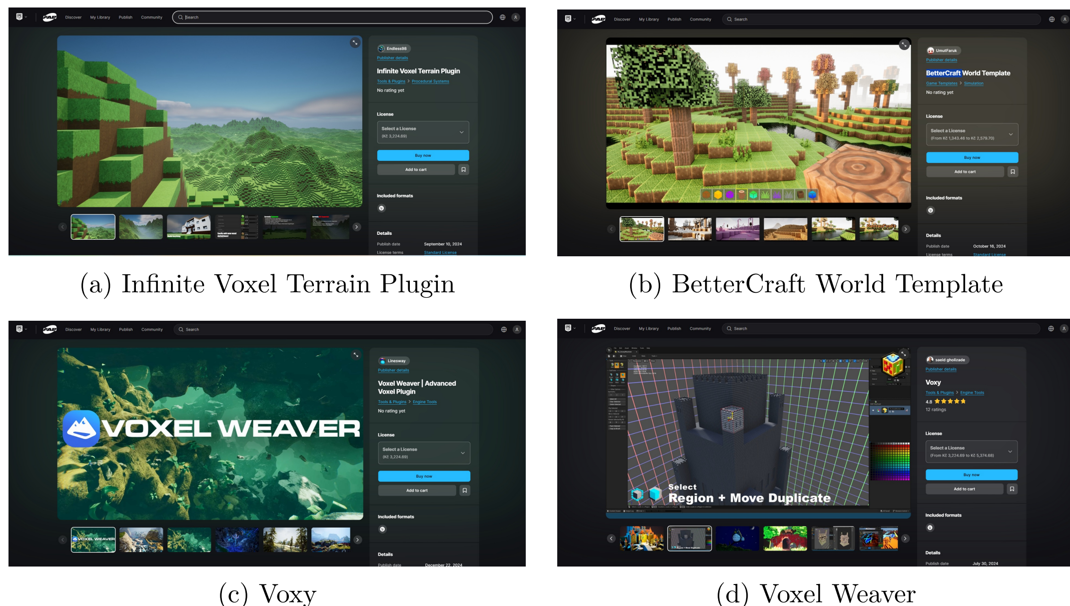
1. **Open-source availability**: The source code must be freely accessible and publicly available.

2. **Fab platform distribution**: The plugin must be distributed through the Fab platform.
3. **Support for cubic mesh generation**: The algorithm must support cubic mesh generation as stated in the visual goals defined in this thesis.
4. **Runtime voxel interaction**: The implementation must enable voxel meshing and support interaction with the stored voxel model to allow its modification during runtime.
5. **Compatibility with custom voxel models**: The algorithm must allow integration of voxel models defined in this thesis.

Each requirement plays a specific role in establishing the algorithm's relevance and applicability within the scope of this project. The availability of source code is essential for conducting performance profiling, as further described in chapter 8, and also ensures favourable licensing terms. Distribution through the Fab platform guarantees visibility and ensures that the implementation is usable within the current Unreal Engine ecosystem, which serves as the target environment. Support for cubic mesh generation aligns directly with the visual goals of the thesis. Enabling voxel meshing along with runtime interaction with stored data ensures the same implementation scope of the algorithm. Finally, compatibility with custom voxel models allows consistent evaluation using the same input data and voxel models.

Figure 6.1 illustrates the specific filter[1] of the Fab platform that was examined to identify algorithms satisfying these requirements. Algorithms on the internet not distributed on Fab fail 2. requirement.

6.1 Paid



(a) Infinite Voxel Terrain Plugin

(b) BetterCraft World Template

(c) Voxy

(d) Voxel Weaver

Figure 6.2: Fab screenshots of paid algorithms

[1] Link to FAB filtered search: https://www.fab.com/search?tags=voxel

The plugins discussed below represent examples of what is currently available on the Fab platform. None of the paid plugins reviewed meet the first requirement 1, which is essential for classification as competitive algorithms. This includes tools such as Infinite Voxel Terrain Plugin[2], Voxel Forge[3], and BetterCraft World Template[4]. While these plugins support runtime voxel interaction, are distributed through Fab, and in some cases allow custom voxel model integration, they still fall short of fulfilling all required criteria.

Other tools, like Voxel Weaver[5] and Shader World Plugin[6], seem to use smooth terrain for rendering instead of cubic mesh generation, which does not meet requirement 3. Similarly, Corgi Voxels[7], Voxel Toolkit: MagicaVoxel[8] and Voxy[9] mainly function as static sculpting tools, and do not seem to satisfy requirement 4.

Some of these plugins together with their pricing can be seen in figure 6.2.

Some plugins offer demos but are not distributed through the Fab platform, which makes the demos incompatible with requirement 2.

In summary, these plugins may showcase what kind of algorithms are currently available on Fab, but I cannot use them in a technical detail due to the aforementioned requirements. None of them meets all five requirements. Therefore, they cannot be considered competitive algorithms within the context of this thesis, and I do not have access to evaluate them.

6.2 Porism DIMs World Generator

PorismDIMsWorldGenerator is a procedural voxel world generation plugin for Unreal Engine, available in both a paid[10] and a Lite version[11]. It is based on the FastNoise2 implemented by this plugin, it is framework for noise-based terrain generation and is accessible through Blueprints, requiring no C++ knowledge. The plugin supports terrain shaping, biome configuration, transparent materials, instanced static meshes, and automatic material painting. It includes configurable chunk sizes, viewing ranges, and a multithreaded architecture.

The paid version includes multiple mesh shape calculation approaches, terraforming, transparency handling with different collision configurations, and multi-actor tracing for surface calculation. The Lite version provides core generation features with limited mesh and collision functionality and does not support networking. The plugin includes a publicly available GitHub repository[12] and example projects.

During the writing of this thesis, the Lite version was freely available. However, at the time of revision, the Unreal Engine Fab platform lists it with a price. After forking the repository into a personal GitHub repository[13] for evaluation, I found that integration with custom voxel models is not possible. The free version is primarily designed for terrain gen-

[2]Link to the Fab page: https://www.fab.com/listings/3ebd8b53-e314-4aff-b92f-ecee4f062c8c
[3]Link to the Fab page: https://www.fab.com/listings/c8484da4-4908-4dad-9d39-f8b29927442c
[4]Link to the Fab page: https://www.fab.com/listings/7c36c2f2-6456-4dcf-b64d-ceb90296e313
[5]Link to the Fab page: https://www.fab.com/listings/9230afd2-c327-4a78-bed6-c68a3b965076
[6]Link to the Fab page: https://www.fab.com/listings/ee4bcbaf-36ea-4dd9-aade-068e1a8a4794
[7]Link to the Fab page: https://www.fab.com/listings/a68a2e2b-b3bd-4752-bb4c-c7f9558da2fa
[8]Link to the Fab page: https://www.fab.com/listings/5b8a7f93-3dd4-4b41-be5e-bc934dca6d76
[9]Link to the Fab page: https://www.fab.com/listings/4ce8bafb-e0b0-4ef8-a81b-1d4e8d5e97ca
[10]Link to the Fab page: https://www.fab.com/listings/664e9fcc-3639-40bf-a3bd-ae9d4b5aa8ba
[11]Link to the Fab page: https://www.fab.com/listings/3160898c-aef2-4f63-95e2-6a029b9c88a9
[12]Link to the free version GitHub repository: https://github.com/EnginFirestorm/PorismDIMsWorldGenDoc
[13]Link to the personal fork: https://github.com/Pawlost/PorismDIMsWorldGenDoc/

eration rather than general voxel modeling. Additionally, the system relies on precompiled binary files located at `Plugins/PorismDIMsWorldGenerator/Binaries/Win64`, which prevents access to and modification of core functionality. As a result, support for custom voxel models cannot be implemented, requirement 5 is not satisfied, and this plugin cannot be stated as a competitive solution.

6.3 Voxel Plugin

Voxel Plugin is a plugin for Unreal Engine that enables the creation of fully volumetric, infinite voxel-based environments. It is available in both a paid[14] and a free version[15], with the free version offering core functionality suitable for custom development and benchmarking. The plugin supports terrain editing both in-editor and at runtime and includes systems for procedural generation using voxel graphs, mesh manipulation, and foliage placement. It is integrated with Unreal Engine, exposing functionality through both Blueprints and C++, and is designed to work with the engine's rendering and world systems.

It provides multithreading support and dynamic LOD features for performance management. The plugin's architecture and documentation [88] allow developers to extend or modify its components. Source code access is available via GitHub[16], which permits low-level integration with custom implementations. In this thesis, the term Voxel Plugin will refer to this free version of the plugin.

As described in previous sections, this plugin was the most frequently mentioned in the community forums under voxel meshing support for Unreal Engine, often recommended as the best voxel meshing solution available for Unreal Engine. This is for example proven by community tutorials[17]. Among available solutions, Voxel Plugin is the only plugin that satisfies all requirements for integration with Unreal Engine, real-time editing, procedural generation, and low-level extensibility. As such, it can be considered a competitive algorithm for evaluating alternative voxel meshing techniques.

I made and published a fork of the free version of the Voxel Plugin on my GitHub[18] to enable direct performance comparisons. Even if the Voxel Plugin documentation [88] mentions support for performance profiling of the plugin, I need to implement custom profiling scopes to allow precise benchmarking of meshing operations and ensure that performance data is collected under consistent conditions and comparable to other solutions.

To enable direct performance comparisons, a fork of the free version of Voxel Plugin was created as part of this work. Custom profiling scopes were implemented within this fork, allowing precise benchmarking of meshing operations and ensuring that performance data could be collected under consistent conditions.

[14] Voxel Plugin Pro Legacy Fab Page link: https://www.fab.com/listings/5c85f2f0-cf03-4860-b22e-e4f470af4133

[15] Voxel Plugin Free Legacy Fab Page link: https://www.fab.com/listings/4f9c8104-1f2b-42c3-b167-85ea589a8b3b

[16] Link to Voxel Plugin Free Legacy GitHub repository: https://github.com/VoxelPlugin/VoxelPluginFreeLegacy

[17] Link to a community tutorial: https://dev.epicgames.com/community/learning/tutorials/4ODO/how-to-create-an-interactive-voxel-world-in-unreal-engine-5

[18] Link to my GitHub fork of Voxel Plugin Free Legacy: https://github.com/Pawlost/VoxelPluginFreeLegacy

6.3.1 Voxel meshing in Voxel Plugin

A showcase of terrain generation using the Voxel Plugin is available in Unreal Engine level located at `Content/Map_VoxelPlugin_Showcase.umap`.

Figure 6.3: Voxel-based environment generated by Voxel Plugin in a showcase level.

Figure 6.4: Voxel-based environment generated by Voxel Plugin in a showcase level with highlighted LOD transitions.

To generate a voxel-based surface within an Unreal Engine level using the Voxel Plugin, a C++ class named `VoxelWorld` must be added to the level as an actor. This actor serves as the primary interface for voxel terrain rendering and generation. The showcase environment created with the Voxel Plugin is illustrated in figure 6.3. In this example, the plugin's LOD system is visible. The LOD transitions between different levels are highlighted in figure 6.4.

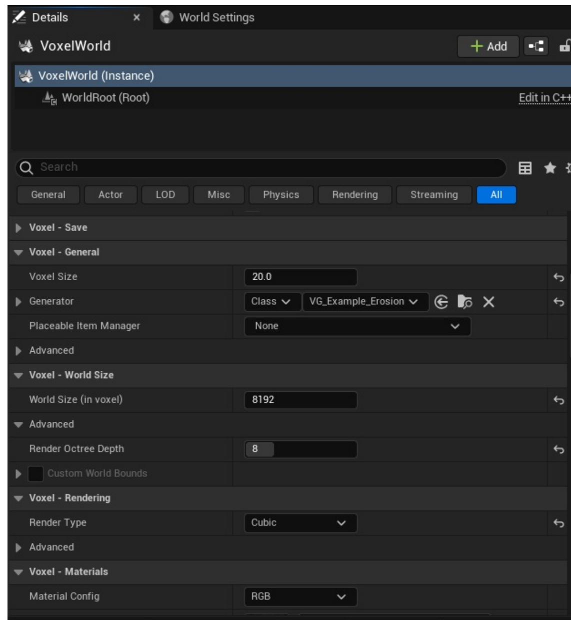
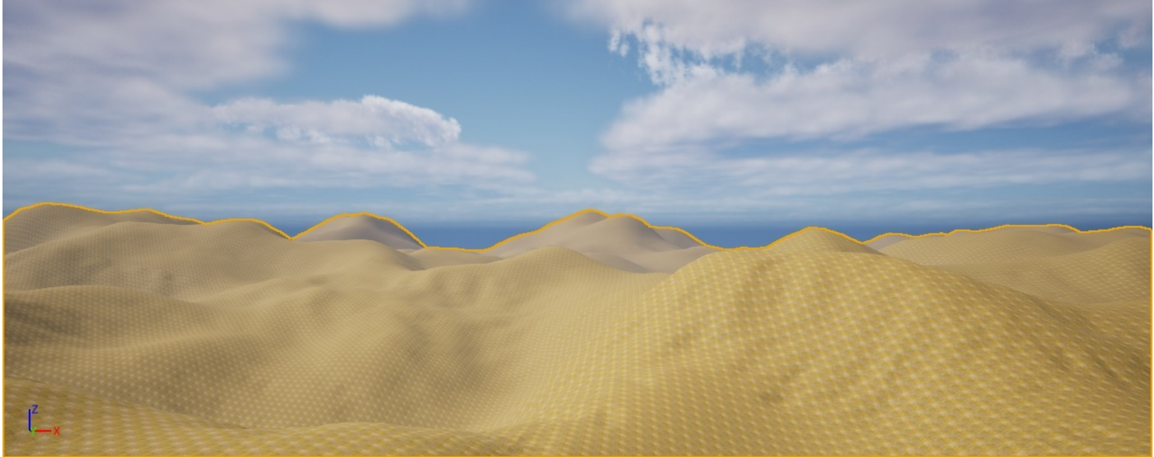
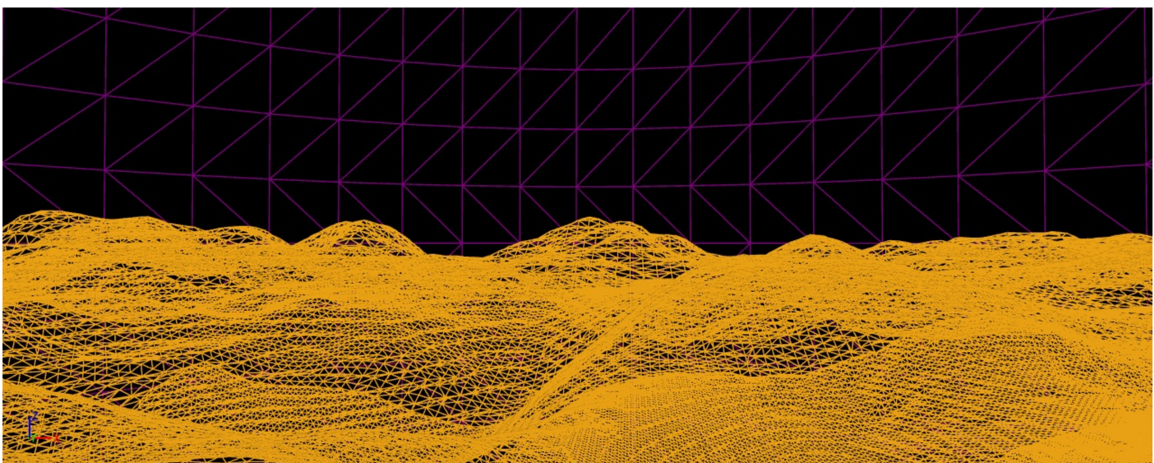


Figure 6.5: Configuration of an VoxelWorld actor.

After placing the `VoxelWorld` actor, a `Generator` must be selected under section `Voxel - General` of the actor configuration. This can be seen in figure 6.5. For the purpose of this demonstration, the `VG_Example_Erosion` generator was selected from the provided default generators.The overall size of the generated voxel world can be controlled by adjusting the octree depth parameter.



Showcase

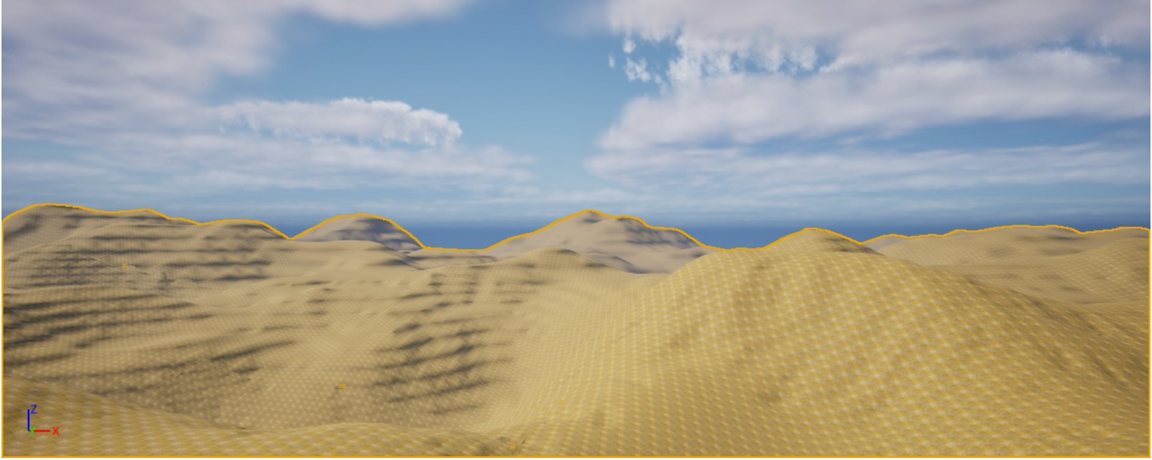


Wireframe

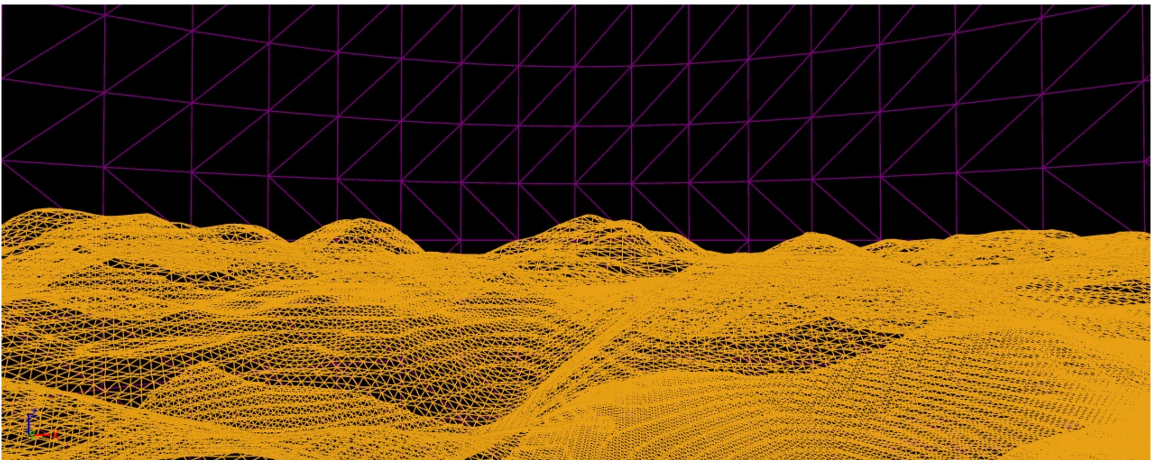
Figure 6.6: Screenshots of Marching cubes surface generated using Voxel Plugin.

Another configurable parameter is the `Render Type`, which determines the voxel meshing algorithm that is applied. For the purposes of this thesis, the cubic voxel mesher was selected, as it generates a cubic surface which is aligned with the stated visual goals. However, the Voxel Plugin also supports additional meshing algorithms designed to produce smoother surfaces. These voxel mesher were discussed in section 4.1.

It is specifically Marching Cubes and Surface Nets. Examples of terrain generated using these algorithms, along with their corresponding wireframe representations, are shown in figures 6.6 and 6.7. Because these algorithms differ from the stated visual goals, they are excluded from performance profiling in this thesis.



Showcase



Wireframe

Figure 6.7: Screenshots of Surface nets generated using Voxel Plugin.

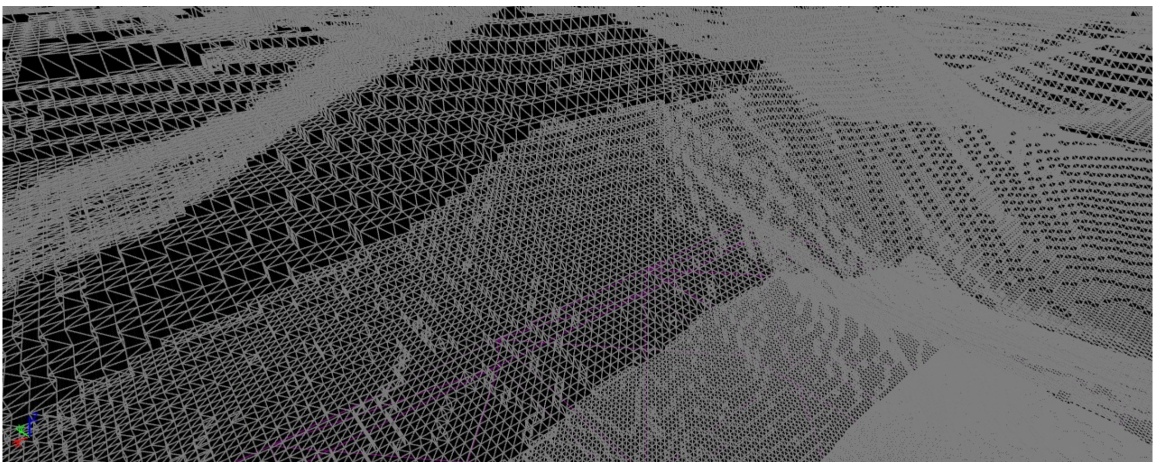


Figure 6.8: Cubic surface wireframe generated by Voxel Plugin.

The wireframe that can be seen in figure 6.8 confirms that the cubic voxel mesher is an implementation of the culled voxel meshing algorithm described in section 4.2.4. It also shows again the LOD transitions that keep this type of voxel meshing efficient. This approach primarily performs culling of inner voxel faces and nothing more. Each visible voxel face is rendered as a separate quad, without utilizing optimization techniques such as for example greedy meshing.

All voxel meshing algorithms are located within the plugin's source directory in a module called `Voxel` under `Source/Voxel/Private/VoxelRender/Meshers` folder, where the base *VoxelMesher* classes are defined. The implementation found in `VoxelCubicMesher.cpp` shows that a chunk is generated immediately before the meshing process. The data is sampled from the internal octree structure using the `QueryZone` function, which operates within predefined chunk bounds. The sampled data is then processed using the mentioned simple cubic voxel meshing algorithm to generate geometry.

From this, it is evident that the plugin leverages an octree-based system for voxel data management. Chunks are dynamically extracted from the octree, which seems to store only positional data and uses procedural noise functions to generate voxel values on demand. Each chunk has a fixed resolution of $34\times34\times34$ voxels, including neighbors. This setup allows for dynamic, on-demand sampling of voxel data and supports lazy evaluation, where the noise function is computed only when necessary, thereby reducing the computational overhead associated with noise generation. Furthermore, LODs are determined based on the player's distance from the voxel. While this improves performance, it also leads to frequent remeshing when a player moves and visible popping artifacts. A personal note is that an improvement to this could be implemented by introducing technologies such as Nanite, which performs GPU-based dynamic LOD optimization on meshes.

Currently, a single material is applied uniformly across all voxels. This simplification is sufficient for performance profiling, because each quad is generated anyway, the material of the quad does not matter. This allows for accurate assessment.

To assess the performance of the Voxel Plugin, custom Unreal Insight-scope traces were inserted into some VoxelMesher classes as well as custom logging. This profiles the important data needed for performance evaluation. As detailed in chapter 8, preliminary code analysis and profiling indicate that the plugin performs highly efficient parallelization. When debugging with breakpoints, it can be found that generated vertices are eventually passed into Unreal Engine's static mesh buffers, which it extends using a custom wrapper.

Profiling scopes were primarily implemented within the various VoxelMesher class implementations.

6.3.2 Voxel Plugin Wrapper

In the context of voxel model generation in the Voxel Plugin. Because an octree structure is used within the Voxel Plugin, we can assume that a typical use case is terrain generation, and the plugin is primarily designed for it. However, when modified, it can also support more general-purpose voxel models. However, these voxel models must be manually constructed and passed into the plugin through a custom implementation.

To facilitate this integration, I developed a C++ wrapper class that bridges my custom voxel models with the Voxel Plugin's octree-based system. Generation of these custom voxel models is described in the next chapter 7. This wrapper effectively implements a custom voxel generator by interfacing with my chunk spawner implementation. It receives

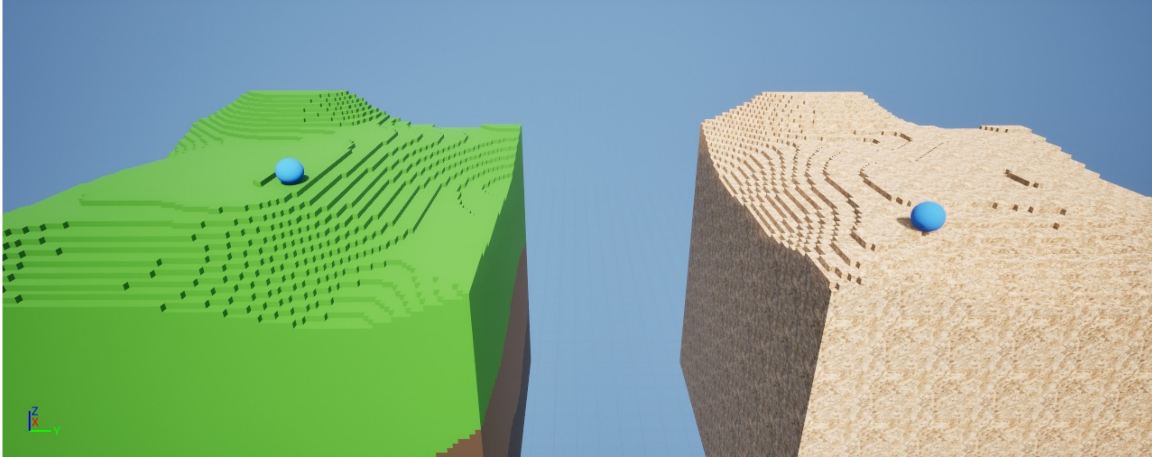


Figure 6.9: Figure showing my voxel model meshed using Voxel Plugin. Left is a voxel model generated using my implementation of voxel meshing. Right is a voxel model generated using Voxel Plugin.

requests for voxel data sampling and redirects them to a flat array containing voxel model data stored in the chunk spawner's memory.

The wrapper was derived from an example generator class provided by the Voxel Plugin and represents additional development work undertaken to satisfy requirement 5: enabling the use of custom voxel models. It equalizes the data structure between my and Voxel Plugin implementation so a consistent results may be produced.

The implemented wrapper is located in the main Unreal Engine project source files and is called `VoxelGeneratorWrapper.h` and the example it was implemented from is located in the source file of the Voxel Plugin in a module called `VoxelEmpales`, The class is called `VoxelGeneratorExample.h` and the path to it is `Source/VoxelExamples/Public`.

To function correctly, the wrapper must be added as a child actor of the VoxelWorld actor within Unreal Engine. The wrapper searches child actors for a chunk spawner. Through this approach, the Voxel Plugin can render the same voxel models as my implementation.

The outcome of this integration is illustrated in Figure 6.9. It allows shows that both voxel meshing algorithms generate colliders that are able to create a physics-based interactions in a combined scene. We can see a test sphere colliding and interacting with both voxel models.

6.4 Conclusion

In conclusion, the only competitive and freely accessible algorithm suitable for performance profiling in this bachelor's thesis is the free version of the Voxel Plugin. Therefore, it will be implemented into the performance profiling scenarios in the chapter 8 and compared to my implementation.

The submitted files to this bachelor's thesis include a modified version of the Voxel Plugin, extended with custom profiling scopes and logging. A voxel generator wrapper and additional integration logic required for this thesis were also added. My implementation is located in a separate plugin next to the `Plugins/VoxelPluginFreeLegacy` directory. This directory contains the copied Voxel Plugin, which serves as an external library. With the

exception of the added profiling traces and logging logic, all other components of this plugin *remain the intellectual property with full authorship rights of the original authors.*

Chapter 7

Implementation of Run Directional Voxel Meshing

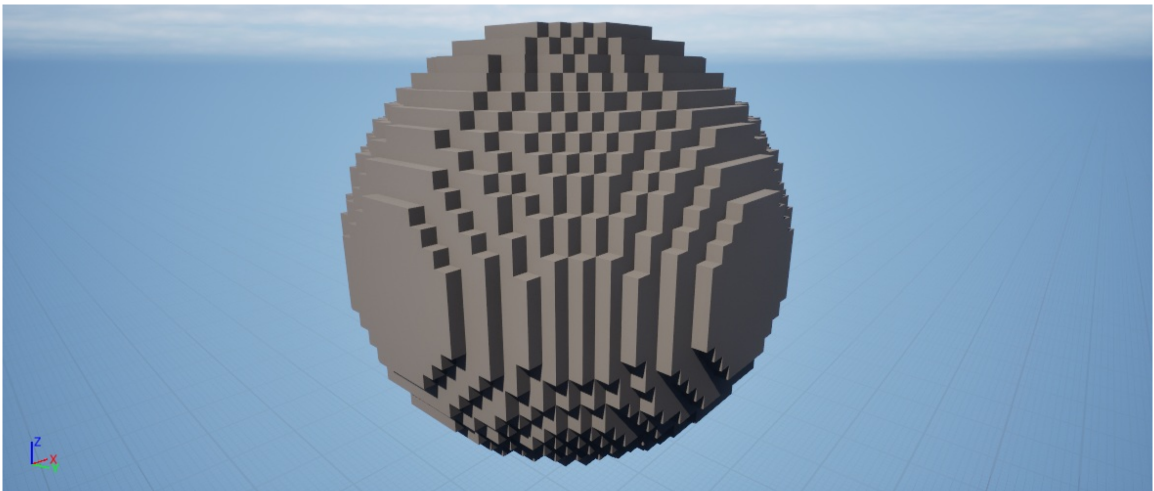


Figure 7.1: Sphere mesh generated using RDM.

Run Directional Voxel Meshing (RDM) is the new voxel meshing algorithm introduced in this thesis. It leverages the way voxel data are stored in a flat array, described in section 3.3.3. The key idea behind RDM is that it generates all necessary quads by traversing the voxel grid only once, in a single, unidirectional pass. This approach minimizes iteration overhead by avoiding redundant passes and directional changes across chunk dimensions. As a result, RDM only needs to sample each voxel once for face generation, maintaining a time complexity of $O(n)$, where n is the number of voxels.

The algorithm incrementally creates, merges, and organizes quads as it processes voxel indices in a flat array. It starts at index 0 and linearly traverses an axis up to its chunk dimension. During traversal, it generates faces in a way that sorts generated quads into a helper array; quads in this array can be eventually merged, ultimately reducing the number of vertices and simplifying the final mesh. Additional implementation details are provided in subsequent sections.

RDM implementation has two variants. Each variant can use different types of voxel data: a standard flat-array voxel grid and an RLE compressed grid. While the primary

objective of this thesis is to demonstrate the algorithm's effectiveness with compressed data, the uncompressed version was implemented as a prototype to develop and validate the core concepts for the compressed variant. The compressed variant builds on the uncompressed one.

Both versions are implemented as a C++ plugin for Unreal Engine.

A notable advantage of RDM is its theoretical independence from chunk size or shape, provided the chunk is a cuboid. This flexibility may be affected if binary-level optimizations are introduced. Working with different chunk shapes may be a motivation for future work.

Figure 7.1 shows a mesh that can be generated from both variants. Each variant includes in its results a figure showing a wireframe of this rendered mesh.

7.1 Voxel Faces

This thesis introduces the concept of voxel faces as a foundational element for mesh generation in all implemented algorithms.

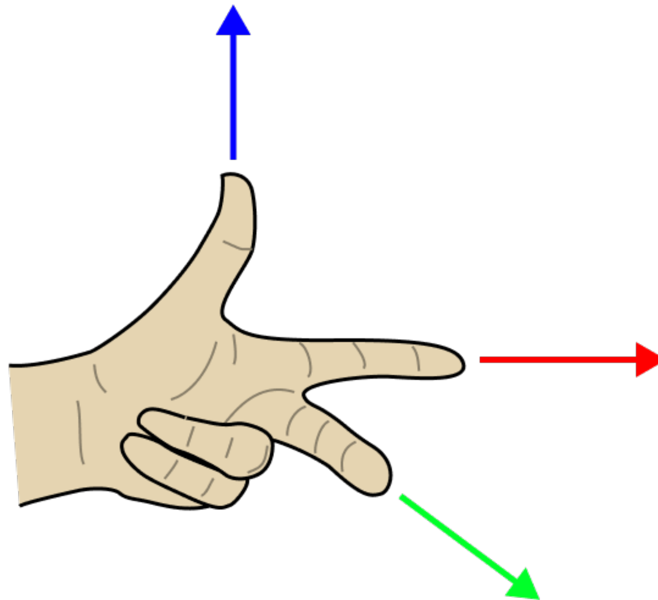
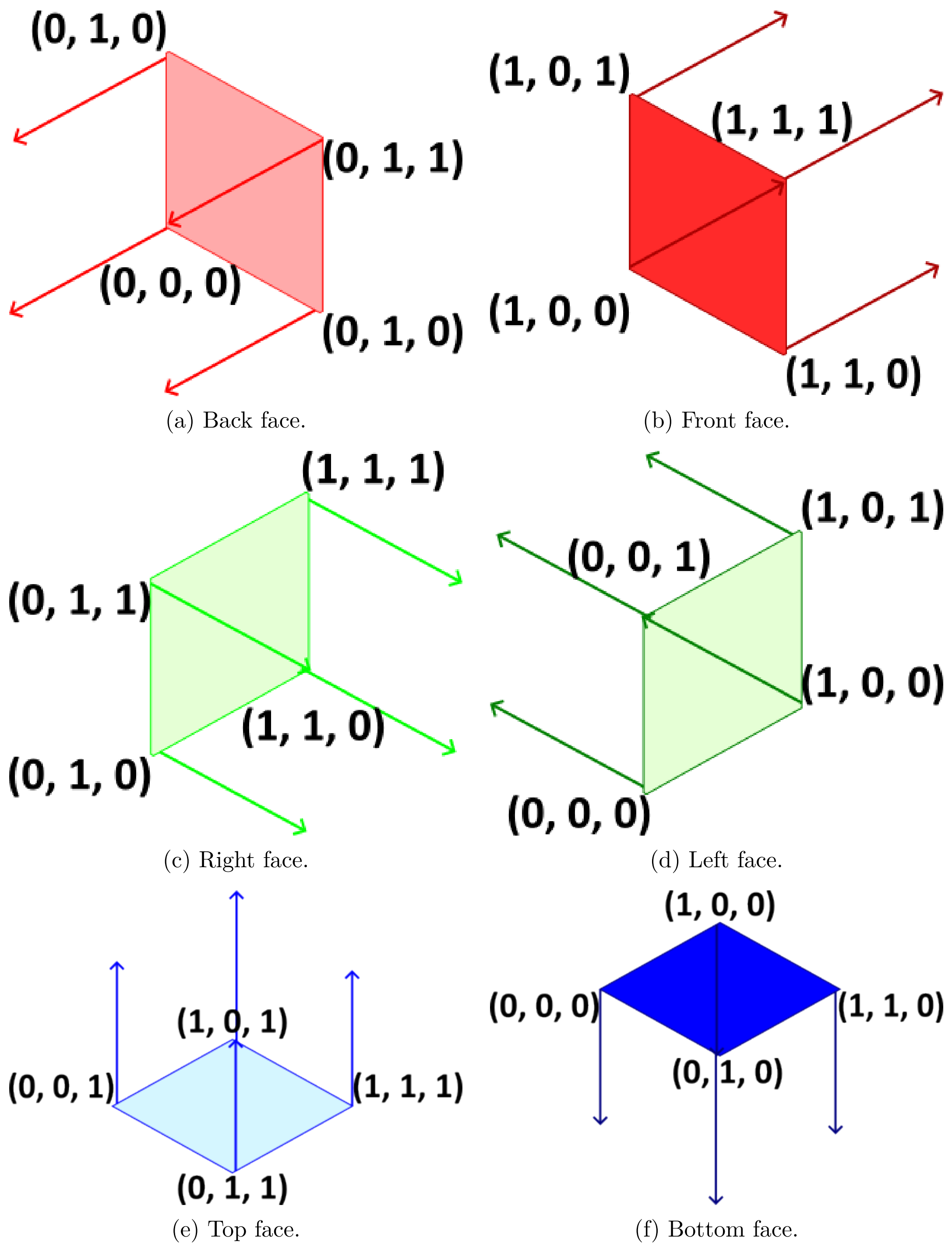


Figure 7.2: Unreal Engine left-handed coordinates image taken from Unreal Engine documentation [21, Coordinate System and Spaces].

Voxel faces in this contextualized around Unreal Engine's left-handed coordinate system, as can be seen in figure 7.2, where:

- The **X–axis** is represented by the red arrow,
- The **Y–axis** by the green arrow,
- The **Z–axis** by the blue arrow.

A voxel face is defined, in context of this thesis, as an oriented quad that represents one of the sides of a voxel in a specific world space coordinate. These faces serve as a boundary representation of a voxel's geometry.



(a) Back face.

(b) Front face.

(c) Right face.

(d) Left face.

(e) Top face.

(f) Bottom face.

Figure 7.3: Voxel faces.

This is illustrated in figure 7.3. The numbers in the illustration are vertex coordinates, which are set when a voxel face is created. These coordinates are then adjusted based on the voxel's position within a voxel grid. The six possible voxel face orientations, corresponding to the sides of a cube, are:

- Front – positive X (+X)

- Back – negative X (-X)
- Right – positive Y (+Y)
- Left – negative Y (-Y)
- Top – positive Z (+Z)
- Bottom – negative Z (-Z)

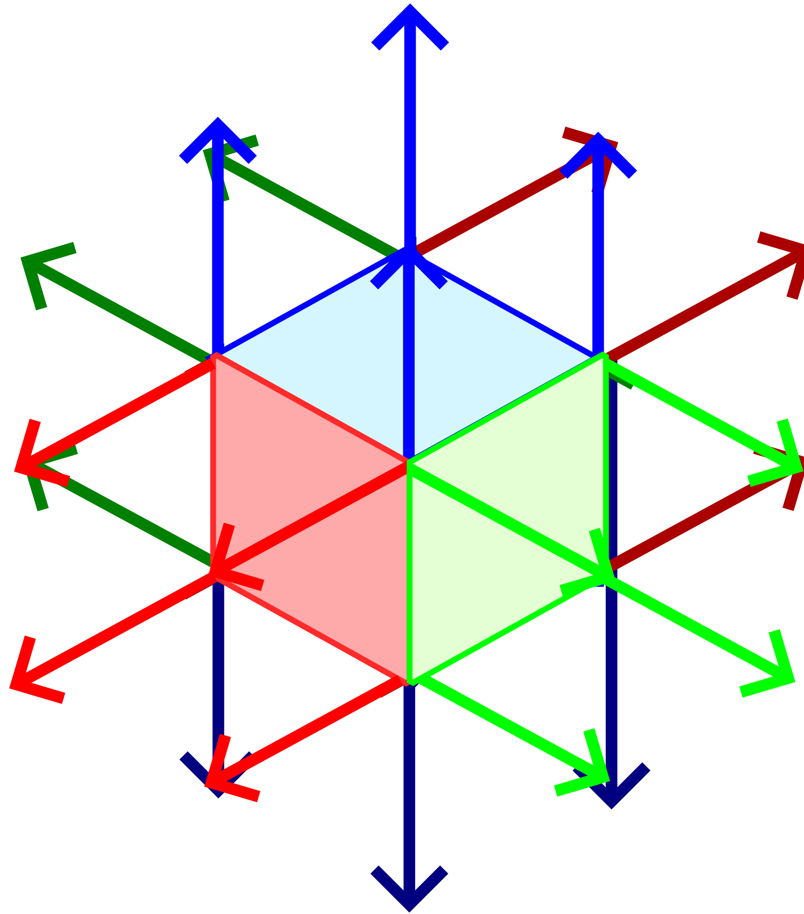


Figure 7.4: Voxel boundary representation using faces.

Together, these faces form a complete boundary representation, as depicted in figure 7.4. When the quads are converted into a mesh by connecting the vertices, they create a mesh boundary where a voxel would be. This way, a voxel is represented by mesh geometry.

All meshing algorithms presented by this thesis utilize a temporary array to store voxel faces. This array serves as an intermediate quad storage from which the final mesh is constructed.

A final part of any meshing pipeline implemented in this thesis is to extract relevant data from voxel faces, such as vertices, indices, and normals, and insert them into Runtime Mesh Component buffers, which are described in their relevant section. This final step is implemented in the `MesherBase` class, specifically within the `GenerateMeshFromFace` function. `FVoxelFace` is a C++ structure that represents the implementation of these voxel faces in this thesis.

7.1.1 Merging of voxel faces

Another concept presented in this thesis is the merging of voxel faces and therefore producing optimized meshes. For clarity, each vertex in a quad is assigned a name: Start Vertex Up (SU), End Vertex Up (EU), Start Vertex Down (SD), and End Vertex Down (ED). This naming convention is illustrated in figure 7.5 and simplifies the process of comparing and merging faces.

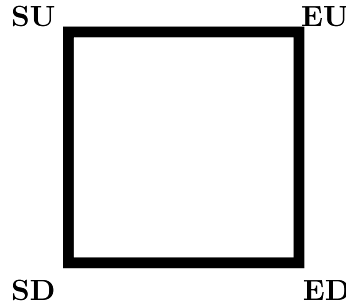
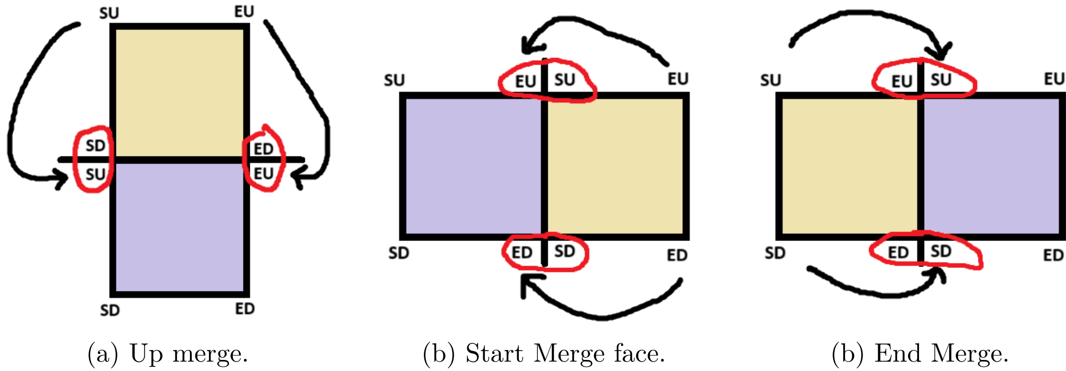


Figure 7.5: Named vertices in a voxel face.

During traversal, voxel faces are merged by comparing the vertices of a newly generated face with those of the previous face. If the corresponding vertices match, the previous face is extended to merge the coordinates of the new face, and the new face is then discarded. Possible vertex–comparing the merging scenarios is illustrated in figure 7.6.



(a) Up merge. (b) Start Merge face. (b) End Merge.

Figure 7.6: Examples of how vertices may be compared and merged together.

To enable this type of merging, voxel faces must first be sorted to maintain continuity between relevant vertices. This sorting ensures that adjacent and mergeable faces are positioned consecutively in a processing order.

This method is applicable to any generated quads and reduces the number of vertices, and therefore optimizes the generated geometry, such as run strips or greedy meshes.

7.2 Voxel Grid variant

An important step toward developing the assigned RLE–based Run Directional Meshing algorithm, is the implementation of a grid-based variant. This variant will be referred to

throughout the thesis as `Grid-based RDM`. It uses a voxel grid and serves as a simplified testbed for exploring the core principles of the algorithm. These principles include directional traversal, voxel face generation and quad merging, which are essential to the later development of a compressed variant.

The grid–based approach allows constant-time access to any voxel within the voxel grid, which significantly simplifies both implementation and debugging. This is why it was implemented first. The immediate accessibility made it easier to validate the behavior and concepts of directional meshing and to develop a baseline set of optimizations, by implementing and evaluating the core ideas first on an uncompressed voxel grid, it became possible to better understand the behavior and identify which optimizations could later be adapted or extended to the compressed domain.

Although many of the optimizations introduced in the grid-based version could, in theory, be transferred to the RLE-based implementation, implementing these optimization strategies is beyond the scope of this bachelor's thesis and is recommended for future work. Nonetheless, the insights gained from the Grid-based RDM directly inspired the development of the RLE-based algorithm, which introduces a new approach by operating on the compressed voxel data without decompression.

In summary, the Grid-based RDM functions primarily as a proof of concept for RUn Directional Meshing. Confirming these concepts is out of scope for this thesis and is considered additional work.

7.2.1 Voxel Meshing

The process of Run Directional Voxel Meshing is illustrated in figure 7.7. In this figure, the colored voxels surrounding the currently sampled voxel highlight the logic behind face generation and culling. Specifically, two directions are evaluated for each voxel: the backward direction is used to determine whether a new face can be merged with an existing one, while the forward direction checks if the adjacent voxel is opaque. If a solid voxel is found in front of the current one, the corresponding face is considered hidden and is culled, as it would not be visible in the final mesh.

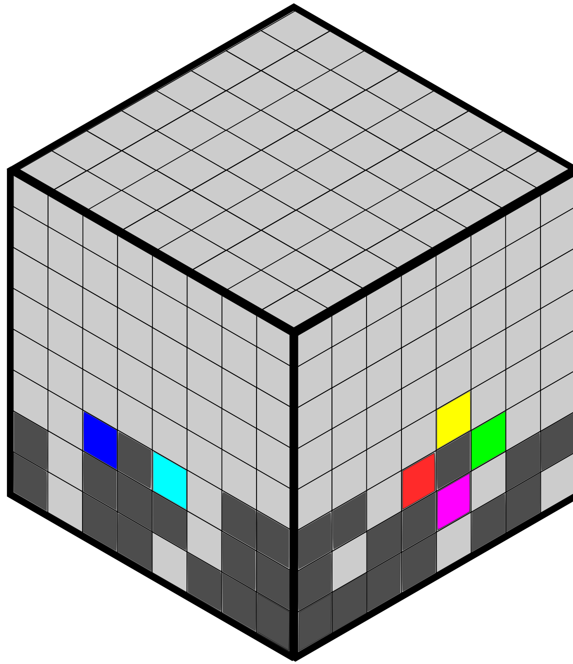


Figure 7.7: Grid–based RDM traversal and merging.

A distinctive aspect of this method lies in the directional traversal of the voxel grid. Each axis can be traversed in a unique run direction, meaning that the order in which the grid is accessed can differ per axis. For example, traversal along the X-axis may follow a different iteration pattern than that of the Y- or Z-axis. This asymmetry allows more efficient grouping and merging of adjacent faces during traversal. Crucially, since voxel data is stored in a flat array with constant-time ($O(1)$) access, the iteration direction can be altered without impacting performance. This flexibility is key to enabling optimized face sorting and merging without compromising iteration complexity.

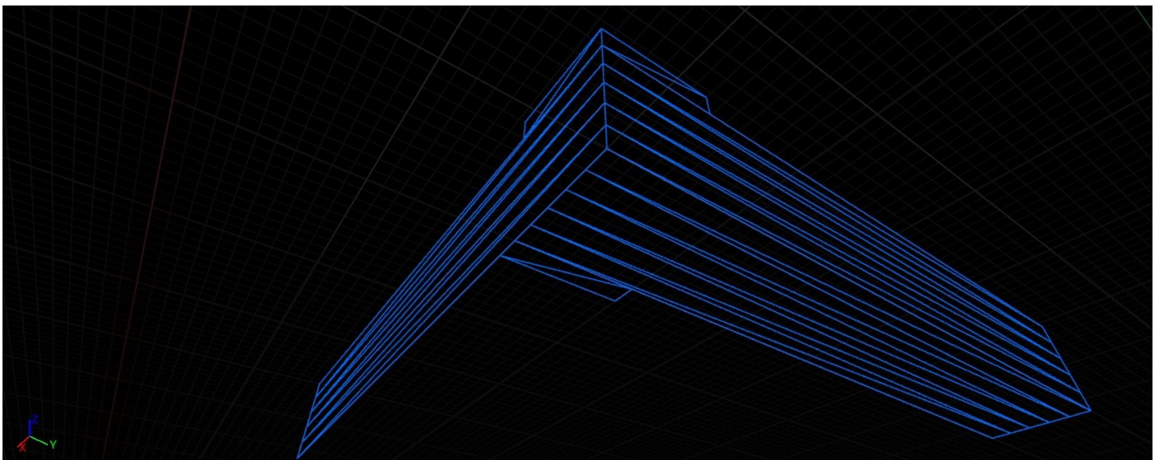


Figure 7.8: Screenshot taken during the initial process of face generation and merging phase in Grid–based variant, visualized via a modified version of RDM.

The practical result is demonstrated in figure 7.9. When two voxels are adjacent, their shared internal face is eliminated (culled), and if their visible faces are identical, they can be

merged into a single, larger quad. All generated faces are temporarily stored in a dedicated face container, which facilitates both sorting and merging during traversal. If two adjacent faces in the container share at least two vertices along the merging direction, the algorithm does not add a new quad but instead extends the previous one. This enables the creation and merging of quads to occur simultaneously in a single pass through the voxel grid.

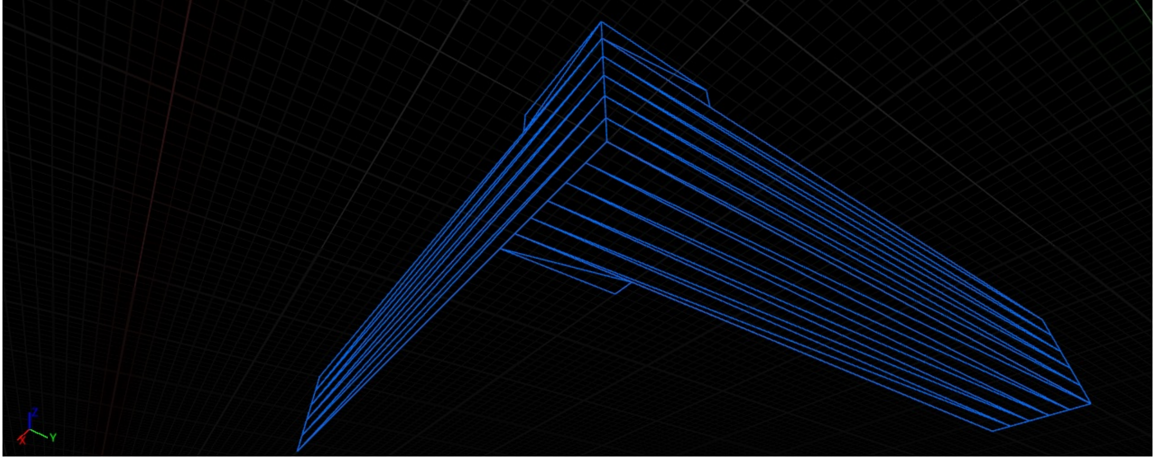


Figure 7.9: Screenshot taken during the initial process of face generation and merging phase in Grid–based variant, visualized via a modified version of RDM.

The sorting strategy takes full advantage of the flat array structure. Because index computations are efficient, the algorithm can traverse the voxel grid in any desired run direction and consistently use the same coordinates if they are used in a different order for each axis. As a result, quad strips are produced in an organized and consistent manner along the traversal path. These directional passes accumulate improvements throughout the process, forming longer and more optimized mesh segments.

The final mesh run strip quads aligned with the traversal directions and produced from this voxel meshing are shown in figure 7.9. The core result of the Run Directional Voxel Meshing algorithm is the ability to produce sorted, merged quad runs in a single, efficient pass through the voxel data.

This voxel meshing, together with some optimizations, is implemented in class `RunDirectionalVoxelM`

7.2.2 Optimization

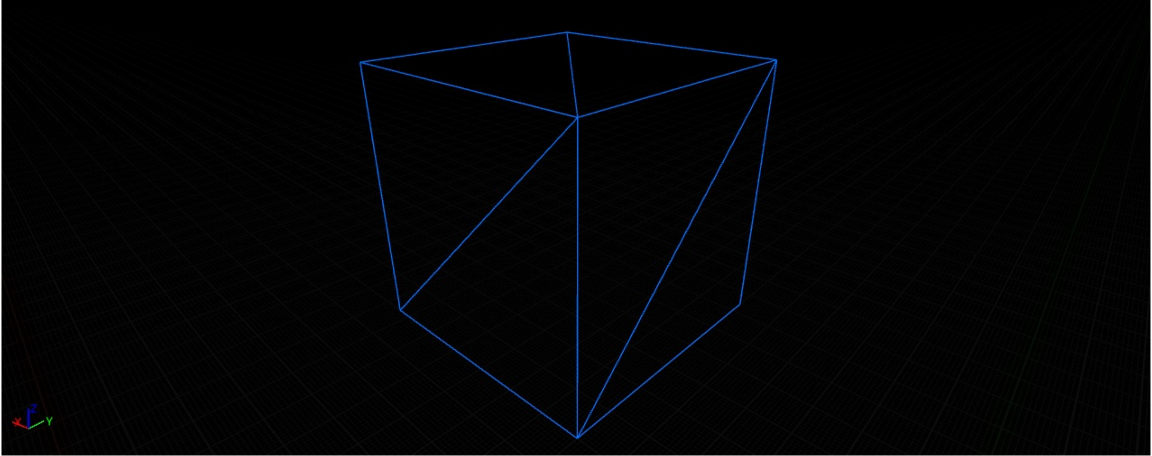


Figure 7.10: Final mesh after greedy mesh optimization.

The goal of mesh optimization in this context is to produce a greedy mesh, which is ideal for performance, as it reduces the total number of generated verticies by merging adjacent faces into larger quads.

After voxel faces are generated and initially merged into a run strips, a second merging phase takes place. In this phase, the container for each voxel face is traversed in a single direction, and because the faces are pre–sorted, the algorithm attempts to merge each face with the previous one by traversing up to second row and trying to merge each face with the current one.

This backward–looking merge of up to two rows is a key technique that can also be leveraged in RLE compressed Run Directional Meshing. The principle of returning to check the previous sequence would allow for face culling. However, when faces are disjointed or sparsely distributed, this merging approach may introduce non-trivial overhead due to redundant checks and failed merge attempts.

To further optimize the process, this merging logic could potentially be integrated directly into the face creation phase. This would require the use of a buffer zone between merge operations, allowing for optimization without significantly increasing time complexity. In contrast, the current implementation increases complexity based on the number of run strips generated in the previous phase.

Despite this limitation, the current approach is sufficient to demonstrate the viability of the concept, and further optimizations are considered outside the scope of this thesis.

The greedy–meshed result can be seen in figure 7.10.

7.2.3 Results

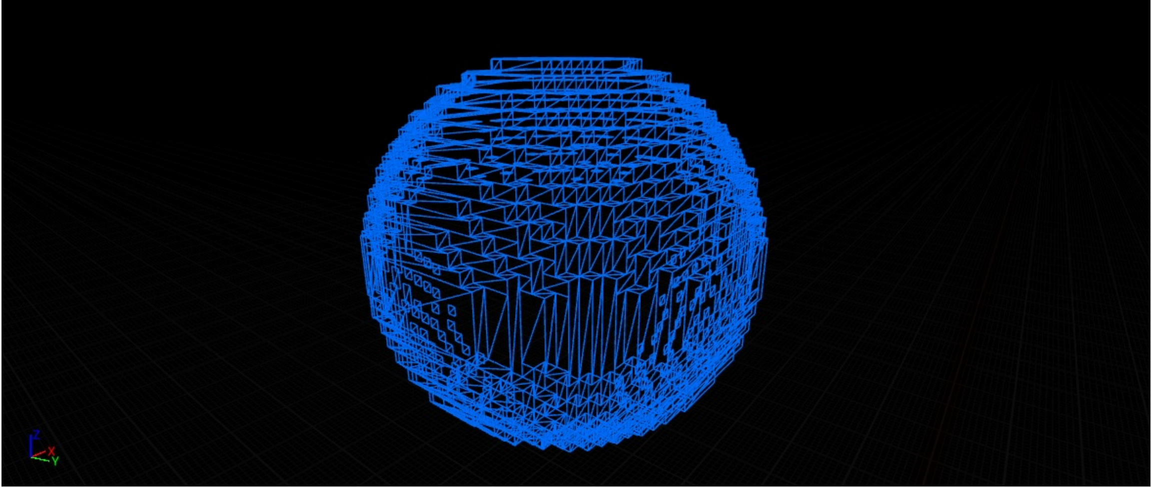


Figure 7.11: Sphere wireframe generated using Grid–based RDM.

The Grid-based Run Directional Meshing algorithm successfully demonstrated the feasibility of performing efficient voxel meshing using a single-pass traversal strategy. Despite operating on an uncompressed voxel grid, the implementation confirmed the core principles of the proposed meshing technique, namely directional face generation, quad strip formation, and face merging.

The Grid-based variant also permits interactive modification of the voxel model. Real-time actions such as voxel placement or removal are immediately reflected in the mesh output, confirming the implementation's responsiveness. These interactions are realized through performing the meshing process again with applied modifications.

Visual inspection of the generated meshes revealed correctly formed and contiguous surfaces, with no visible topological defects or rendering artifacts. All quads were successfully aligned with voxel boundaries, and face merging operations yielded expected reductions in triangle count without compromising surface fidelity. An example of the final result is shown in the figure 7.11, which depicts a wireframe of sphere meshed using the Grid-based RDM variant. The wireframe clearly shows the reduction in geometry.

Although this prototype implementation prioritized clarity and correctness over performance, it nevertheless produced meshes of acceptable quality for real-time rendering. The uncompressed nature of the grid variant served as a valuable debugging aid and enabled a straightforward evaluation of traversal order, face orientation, and merge logic.

In conclusion, the Grid-based RDM implementation validated the fundamental design decisions behind the Run Directional Meshing algorithm. It established a functional reference point for subsequent work on the compressed (RLE–based) variant and laid the groundwork for further optimization, particularly in reducing memory usage and traversal overhead. For this reason, this algorithm will also be used for the performance evaluation in the next chapter.

7.3 RLE-compressed variant

This section introduces the core problem assigned to this thesis: the generation of polygonal meshes directly from run-length encoded (RLE) voxel models without the need for decompression. The proposed method is an extension of the previously developed Grid-based variant. From this section on, the new variant is referred to as RLE-based RDM.

The main objective is to develop an efficient algorithm that converts compressed voxel data directly into mesh geometry. Traditional meshing techniques may require a decompression of the voxel data before mesh generation, which introduces unnecessary overhead. The RLE–based RDM algorithm eliminates this step by operating directly on RLE–compressed voxel data.

The motivation stems from a key insight: RLE–compressed voxel data can be traversed in a manner that mimics the original voxel grid structure. Although the data is stored as compressed intervals (runs), the logical structure of a 3D grid is preserved. Each voxel in the grid still corresponds to a specific 3D coordinate, and each run represents a continuous sequence of identical voxels along the Y-axis. This is illustrated in figure 7.12.

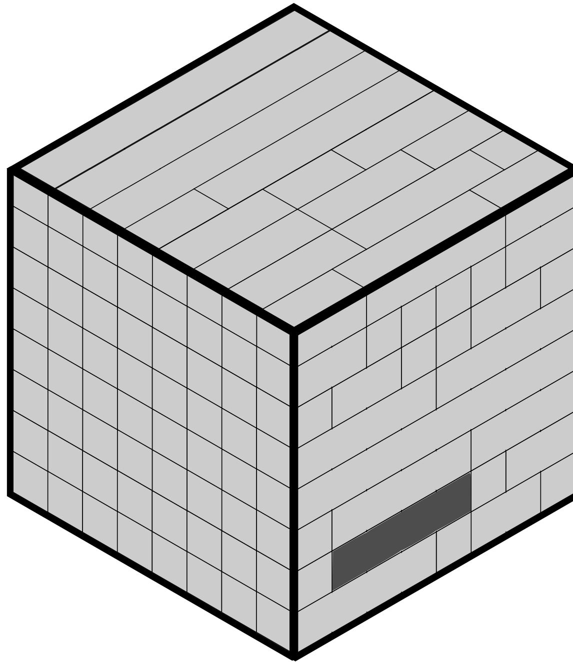


Figure 7.12: Illustration of RLE sequences in RLE–based variant.

One of the primary advantages of this approach is that it allows the mesh to be constructed in a single traversal pass over the compressed data, avoiding full decompression. This is made possible by adapting the RDM algorithm to operate directly on RLE runs. Each RLE sequence is traversed in a fixed axis order—first Y, then Z, and finally X. This is illustrated in figure 7.13, mirroring the original traversal logic of RDM. During traversal, mesh quads are generated from entire runs, taking advantage of the natural face culling produced by the runs in the Y direction.

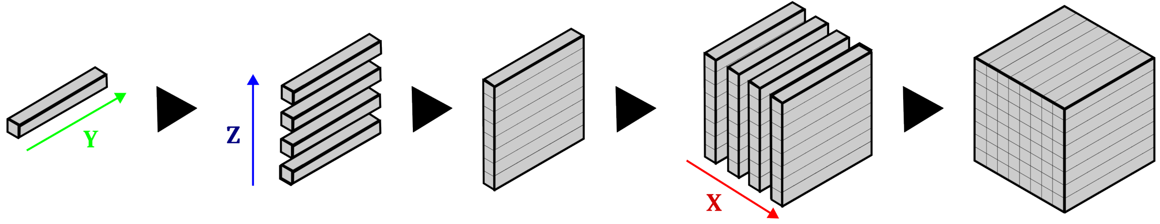


Figure 7.13: Illustration of RLE sequence traversal. It is similar to the voxel grid illustration, but the voxels have become interval-like sequences.

Another important aspect of this method is its support for in-place read-write access to RLE sequences. When modifying voxel data, only the affected part, which I call *edit area* of the sequence needs to be located and rewritten. The rest of the sequence can be traversed without needing to decompress or reconstruct the full grid. This allows for interactive editing and meshing operations to occur simultaneously.

Despite these benefits, working with RLE introduces some challenges. Locating a specific voxel requires linear traversal through the run sequence, which can be costly for large grids. Additionally, meshing must account for run ends, which can happen due to a change in voxel value or when the spatial coordinate changes along a different axis.

7.3.1 Voxel Meshing

During quad generation, the same voxel faces are used as described previously in section 7.1. However, in the RLE-based approach, all non-zero Y coordinates are replaced with the length of the current run, as the Y-axis represents the primary traversal direction in this algorithm.

The nature of RLE voxel meshing is that it produces long strips of quads. Nevertheless, RLE compression inherently merges and culls thanks to naturally occurring run culling along the Y-axis, thanks to the cuboid structure it generates when generating all faces in a run. This automatic culling is advantageous but comes at the cost of significantly less efficient merging and culling along the X and Z axes, which are not compressed during the traversal.

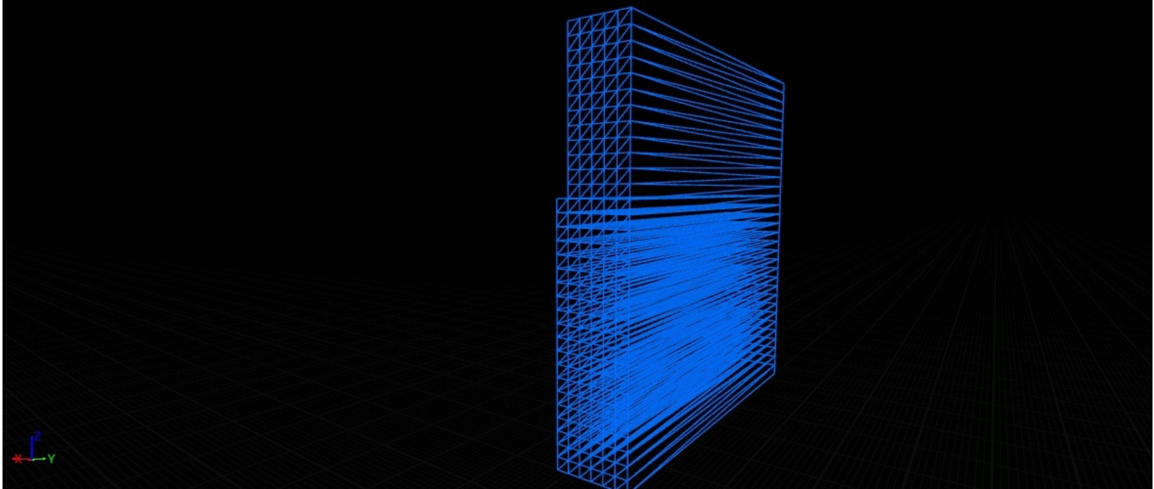


Figure 7.14: Screenshot taken during the initial process of face generation in RLE–based variant, visualized via a modified version of RDM.

Since RLE forms contiguous voxel intervals, the algorithm can directly generate quad strips without requiring an explicit merging step, unlike the grid-based RDM method. An example of this intermediate stage in quad generation is illustrated in figure 7.14.

Although further optimization, such as merging these run strips in a manner similar to grid-based RDM or using the same principle for face culling, is possible, it lies beyond the scope of this thesis. Therefore, this implementation only generates these simple meshes.

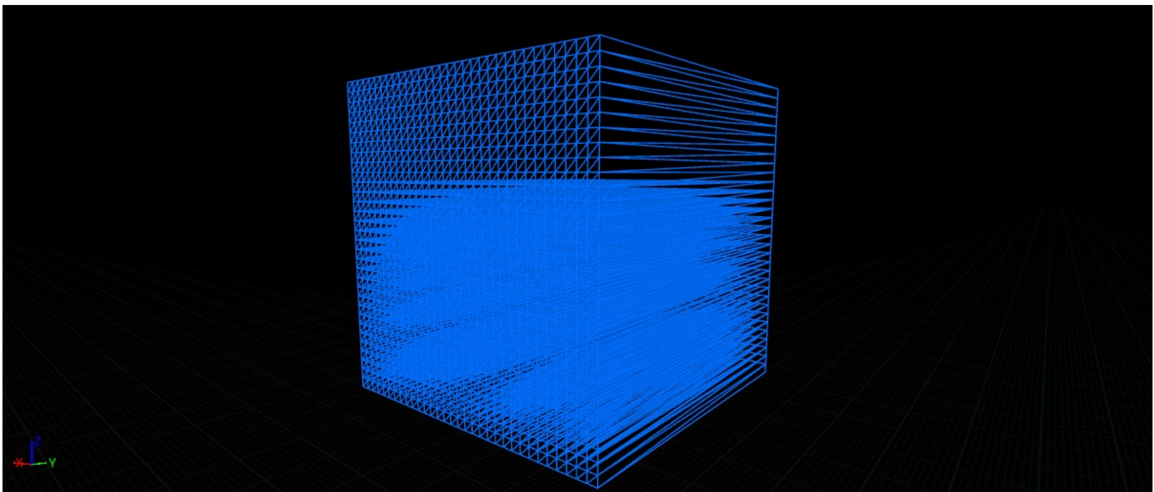


Figure 7.15: Final mesh generated from RLE–based variant.

The resulting mesh, as shown in Figure 7.15 and is primarily composed of unculled run strips. This voxel meshing is implemented in `URLERunDirectionalVoxelMesher` class.

7.3.2 Optimization

The only optimization implemented in the presented algorithm, I call RLE tail culling. This technique leverages the nature of RLE, where sequences may span across multiple axis

runs, to eliminate the generation of certain quads. Specifically, it is a culling optimization integrated directly into the voxel face creation process.

The core idea behind this face culling method is straightforward: when an RLE sequence extends over multiple axis runs, its length can be used to compute the corresponding spatial coordinates. If the voxel material at the preceding coordinate is identical to the current one, the corresponding face can be safely culled, as it will be occluded by a voxel of the same material and thus not visible in the final mesh.

While this represents a basic form of face culling, it is the only such optimization employed in this thesis due to time constraints and because performance optimization lies outside the scope of this work.

It should be noted that implementing more advanced culling techniques in this context is challenging, particularly due to the interval-based structure introduced by the compression.

Nonetheless, the principles of quad merging, as introduced in the grid-based RDM variant, form a foundational basis for potential future optimizations within the RLE-based variant.

7.3.3 Interaction

In voxel systems, read operations are essential. Voxel grids offer simplicity and speed. However, in interactive applications, where user actions primarily involve writing or modifying the model, a different approach is more appropriate. The implementation of RLE–based RDM described in this thesis is specifically improved for such use cases, prioritizing rapid surface–level modifications over frequent voxel–type queries.

A core observation is that interactivity in voxel environments requires immediate visual feedback. Users typically interact with the surface of the model, and any modification to this surface must be reflected in the mesh representation without delay. Consequently, the system must support real-time meshing updates in response to voxel edits. This is a cornerstone of interactive algorithms.

Another critical insight is that RLE sequences support simultaneous reading and writing up to the point of modification. During meshing, the sequence is processed incrementally, and when an edit is detected, the process can apply the change locally and resume meshing from the same point. This allows for partial recomputation rather than full restart, achieving performance comparable to implementations from voxel grids, where the true time complexity for interactivity is not just the constant-time array update ($O(1)$), but also the $O(n)$ cost of regenerating the mesh to update the change.

Due to the compressed nature of RLE, direct access to individual voxels is non–trivial. To address this, interactions are defined as change descriptors, either structured objects or lambda functions, which are applied conditionally during RLE traversal. When a match occurs, the modification is integrated directly into the sequence, minimizing disruption and allowing traversal to continue seamlessly. If no match is found, the system can defer the update and continue background meshing without noticeable interruption.

The primary challenge in this implementation was efficiently locating and modifying the correct segment within the RLE sequence while maintaining compression integrity. This was achieved through the introduction of a new concept: Edit Indices. These indices are computed during meshing and represent precise locations within the compressed sequence where modifications must be applied. This innovation enables real-time interactivity directly on the compressed data, avoiding the need for full decompression.

Modifications are applied within a bounded Edit Area, defined as the region where existing runs intersect with the modification interval. Voxel coordinates are mapped relative to the virtual grid to ensure consistency. The RLE data is then locally unpacked, altered, re-encoded, and meshing continues from the update point.

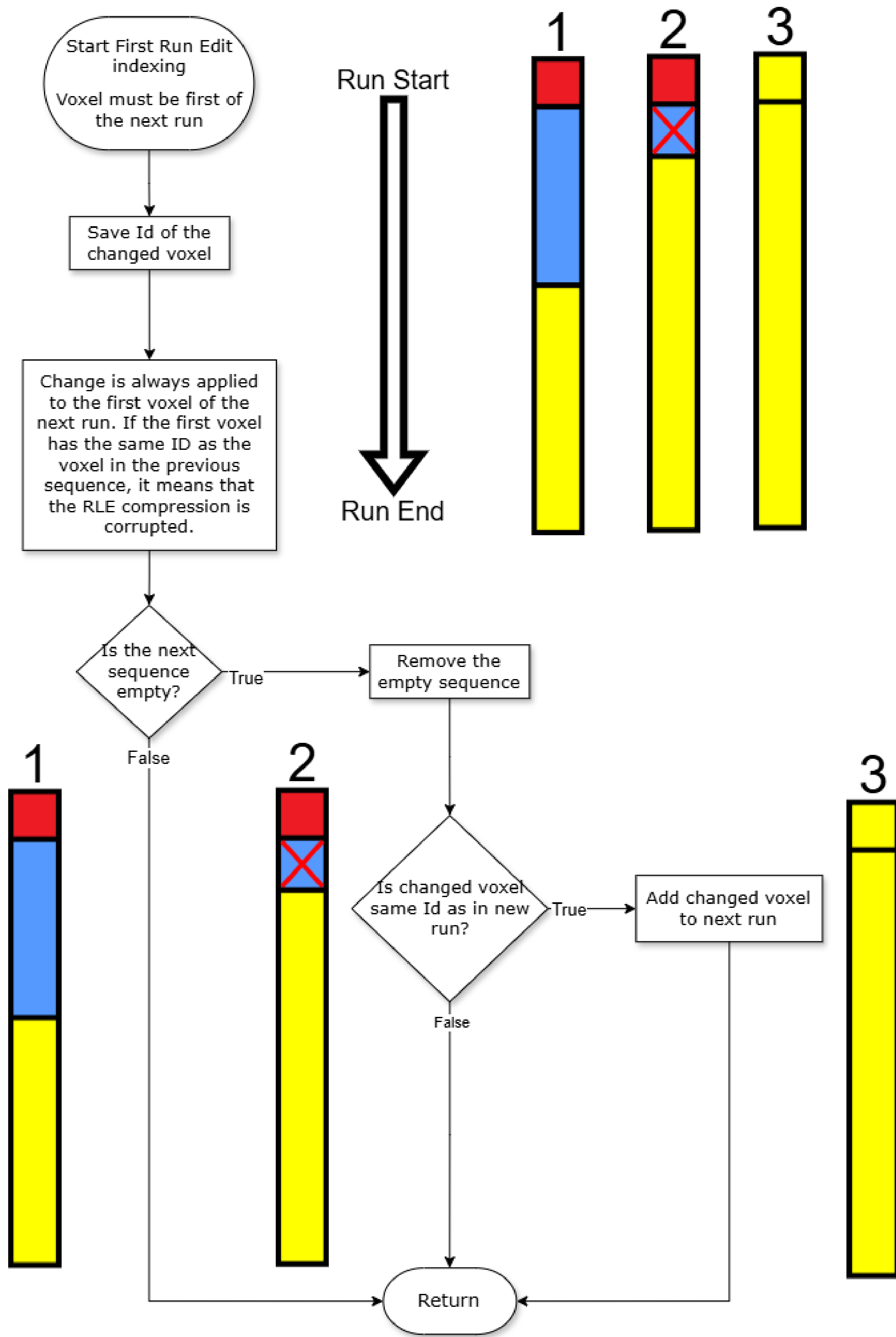


Figure 7.16: Diagram that shows how voxel change is applied to the first run.

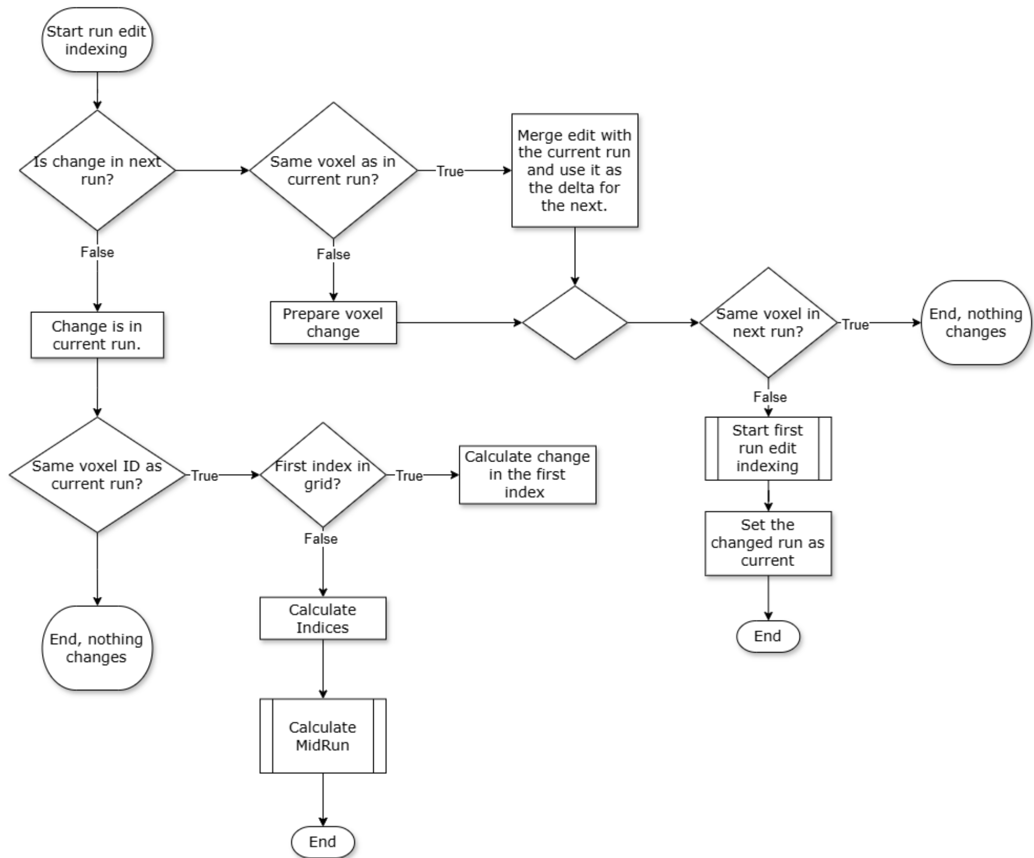


Figure 7.17: Diagram that shows how voxel change is detected at the start of a new run.

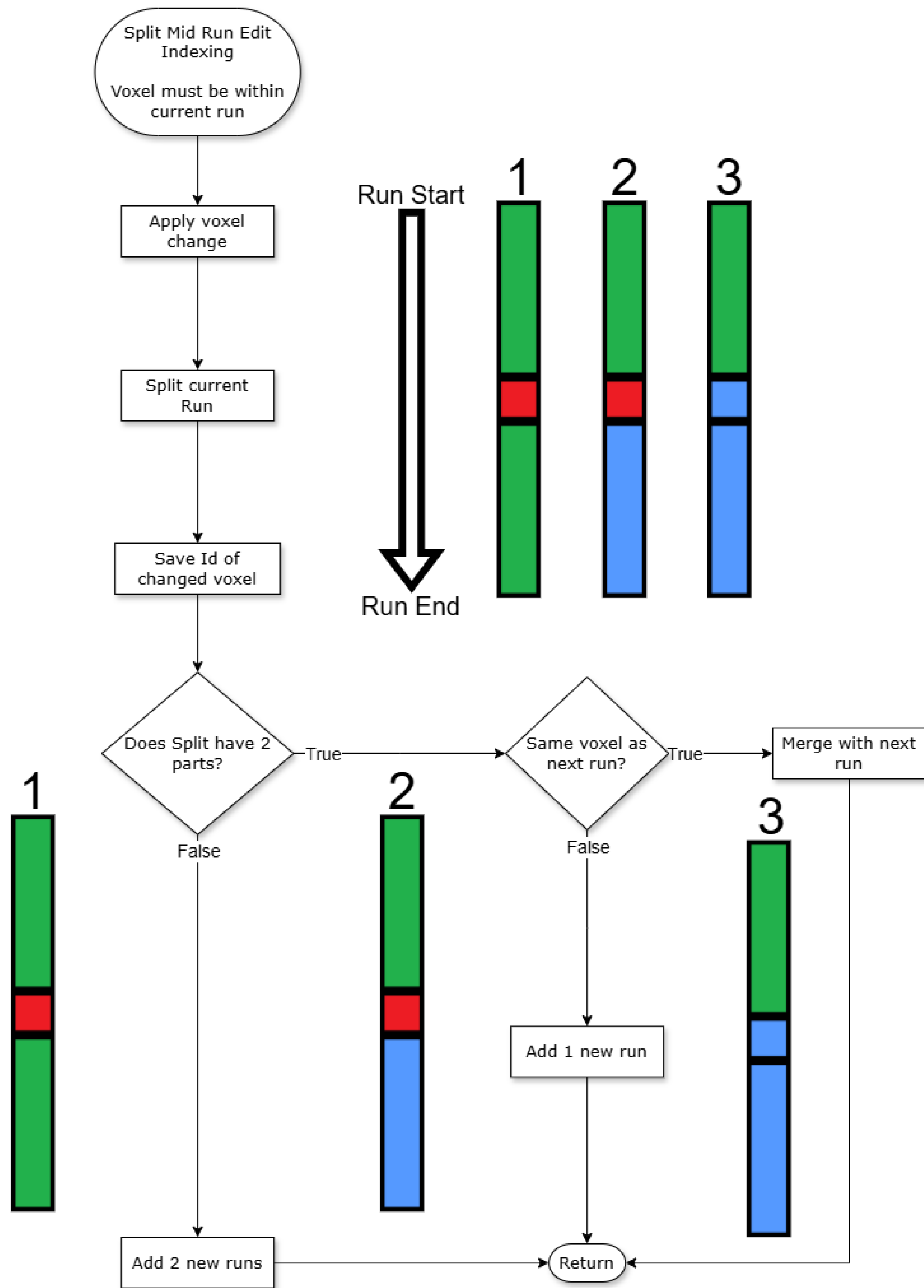


Figure 7.18: Diagram that shows how voxel change is applied when detected mid run.

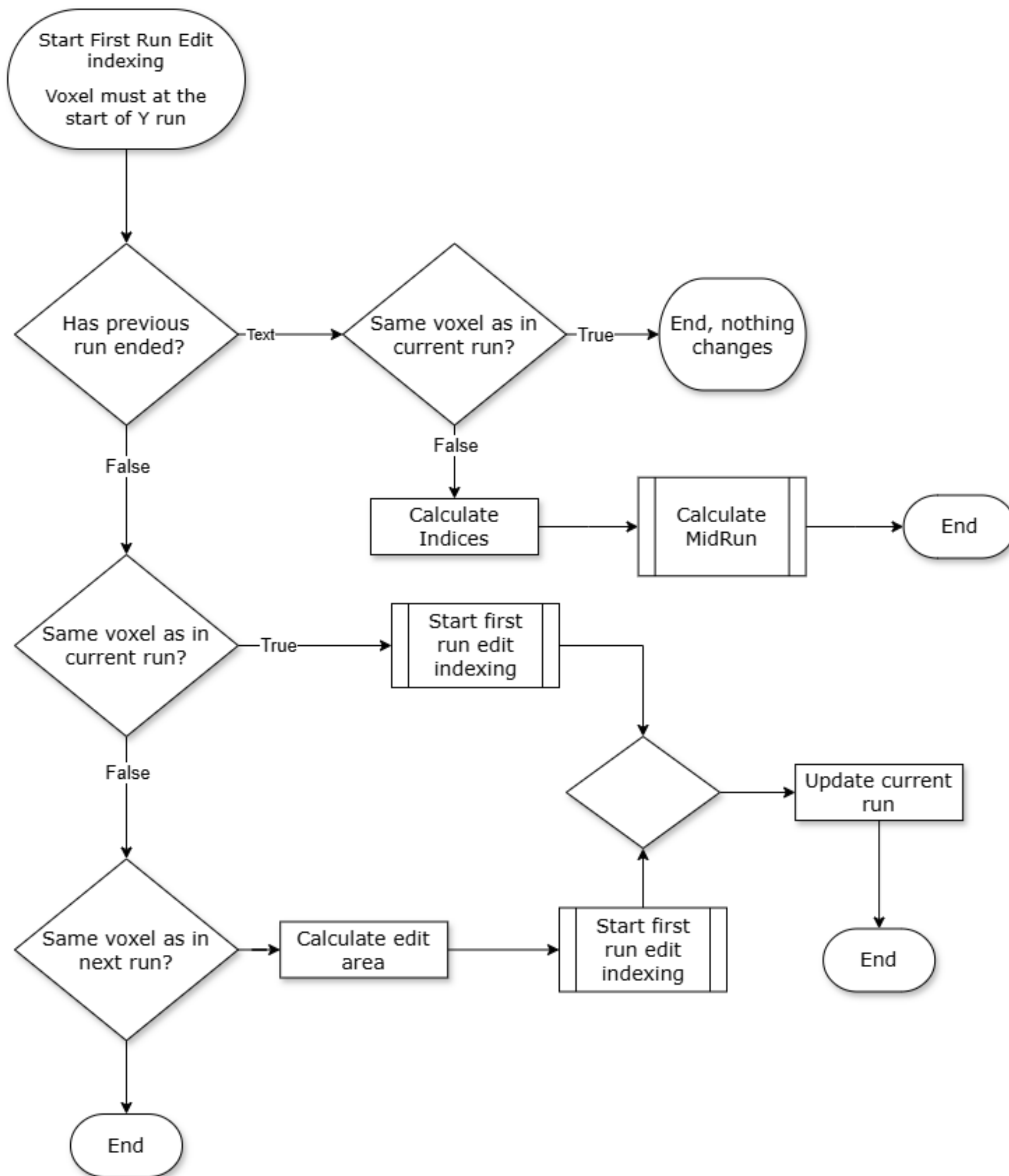


Figure 7.19: Diagram that shows how voxel change is detected when the change is at the start of a new axis.

Two categories of diagrams were developed to illustrate this process:

- Change Application Diagrams: Show how the RLE sequence is modified to preserve valid compression after an edit (7.16, 7.18).
- Run Location Diagrams: Demonstrate how the relevant segment of the sequence is identified based on the voxel coordinate of the modification (7.19, 7.17).

Each RLE sequence in the possible scenario is represented by the color strips. The arrow indicates run direction and by color-coded voxel IDs, enabling intuitive visualization of the compression behavior before and after applying an interaction change. For example, figure 7.18 shows a modification applied to a voxel located within the middle of a run and figure 7.16 shows how a change is applied during the start of a run.

This system therefore supports two fundamental interactions:

- Placing: Replacing an empty voxel with a material.
- Picking: Removing a material voxel, making it empty.

A major contribution of this thesis is the ability to perform interactive meshing concurrently with these operations, without decompression. As RLE is a lossless format, any changes made are faithfully retained when converting back to a standard voxel grid.

This capability fulfills the performance and responsiveness criteria of interactive modeling systems and represents a significant advancement in real-time voxel editing using compressed data structures.

7.3.4 Results

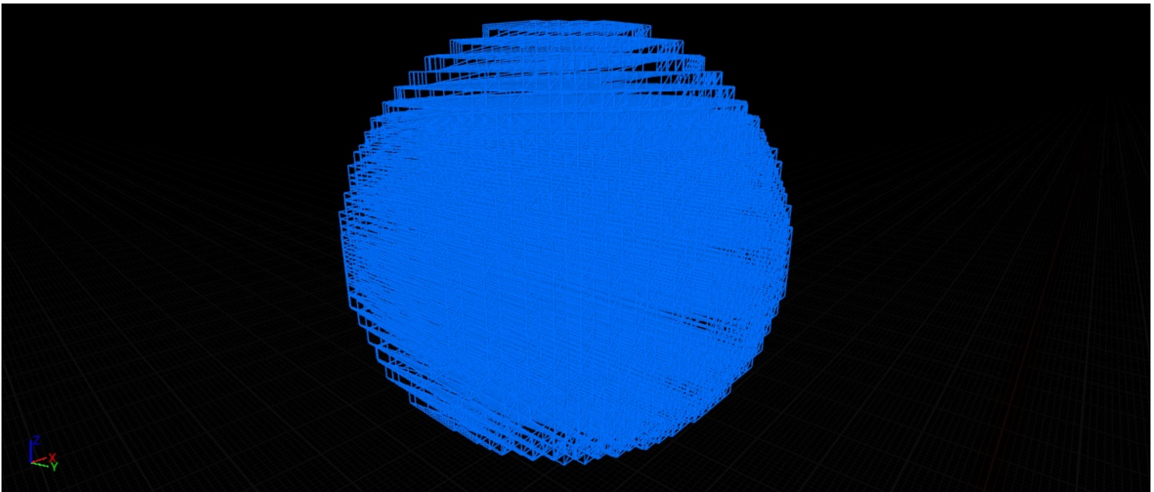


Figure 7.20: Sphere wireframe generated using RLE–based RDM.

The RLE-based Run Directional Meshing algorithm introduces a structured approach to voxel meshing that leverages the advantages of both compression and direct meshing. By combining run-length encoding with directional traversal, this method generates mesh representations that preserve the original voxel structure without requiring decompression.

Although the current implementation is unoptimized, particularly in terms of vertex generation, it effectively demonstrates that direct meshing from RLE-compressed data is viable. This prototype serves as solid proof of concept, establishing the fundamental feasibility of the method. Optimization remains outside the scope of this thesis.

This work should therefore be viewed as a functional prototype that validates the viability of RLE-based meshing. The resulting meshes share structural characteristics and challenges of meshes with unculled faces. In this context, an optimized RLE-based RDM implementation would create a similar difference to what greedy meshing is to the simple mesher.

An example of the final result is shown in the figure 7.20, which depicts a wireframe of sphere meshed using the RLE-based RDM variant. In the wireframe we can see that the unculled strips obstruct visibility of any complex geometry.‘

7.4 Implementation of Voxel Storage

Figure 7.21: Screenshot of Unreal Engine Data table with types of voxels

This implementation is based on storing voxels in a flat array. It uses Data tables introduced by Unreal Engine to store all voxel properties. Voxels are stored in a custom struct `FVoxel` within a `TArray`, Unreal Engine's dynamic array type. For RLE compressed storage, a second struct `FRLEVoxel` is used, which includes the `FVoxel` and a run–length variable indicating how many times the voxel repeats.

Voxel data is defined in a custom data table, where each row corresponds to a unique voxel ID. This ID is used during meshing to retrieve properties such as material type, transparency, and spawning conditions. These materials are passed to Unreal Engine's rendering system to define the appearance of each visible quad. An example of this table is shown in Figure 7.21.

To ensure consistency, the voxel table remains static at runtime—each row's index must match its voxel ID (e.g., row 0 defines ID 0). This simplifies material lookup and avoids runtime data mismatches. This means that a voxel model is made of rows from this table.

7.5 Voxel Generators

This thesis does not focus on content generation. Instead, its primary objective is to evaluate the performance of the meshing algorithm under varying voxel distributions. To support this, procedural noise and terrain generation are used solely to create diverse test cases. Voxel models may also be loaded from persistent storage or received over a network. Regardless of how they are obtained, it is the internal distribution of voxels that has the most significant impact on voxel meshing performance.

The voxel meshing process is executed asynchronously to avoid blocking the main thread. The voxel data used for meshing is produced by components called `VoxelGenerator`.

Voxel Generators are custom classes that procedurally fill a flat array with voxel IDs. By varying the generation logic, it is possible to create a wide range of voxel patterns. These patterns serve as input for performance profiling, allowing the meshing algorithm to be tested under different conditions. Although voxel generation is not a core focus of

the thesis, it is included as an auxiliary system developed specifically to support controlled performance profiling. As long as the resulting voxel grids are correctly filled, the speed of generation is not taken into account.

Each Voxel Generator initializes a reference to a voxel table, which defines available block types, along with a reference to a `VoxelMesher` instance that performs the meshing. These components are configured at runtime. Compression of the generated voxel data is handled by the mesher itself. This approach allows each mesher implementation to convert the raw voxel data into a representation that is most efficient for its specific processing needs.

Future improvements could include direct voxelization into the RLE (Run-Length Encoding) format, further streamlining the workflow, and reducing the number of processing steps.

7.6 Unreal Engine integration

Chunks are voxel models placed in a 3D grid, each defined by its coordinates and size. In this implementation, a chunk consists of two parts: a struct containing a flat array of voxels and an Unreal Engine Actor reference with RMC component. The struct holds a reference to its Actor, linking data to its visual representation. Flexible Chunk sizes are compatible with Run Directional Meshing.

Chunks are created and managed by ChunkSpawners, implemented using Unreal Engine's `TMap` for efficient storage by grid position. Two variants are included: one for managing a single chunk, and another for spawning a grid of chunks. Both inherit from the base class `ChunkSpawnerBase`.

At runtime, ChunkSpawners generate voxel data, build mesh buffers, and spawn Runtime Mesh Component (RMC) Actors to render the mesh in the scene. They expose a Blueprint API that supports event–driven voxel modification. Player interactions, such as raycasts, trigger events handled by the spawners, which is represented in figure 7.22, enabling real-time updates to the voxel model. The full integration of these systems is illustrated in Figure 7.23.

Figure 7.22: Screenshot from Blueprint on how interaction event is passed from Unreal Engine into a Chunk Spawner

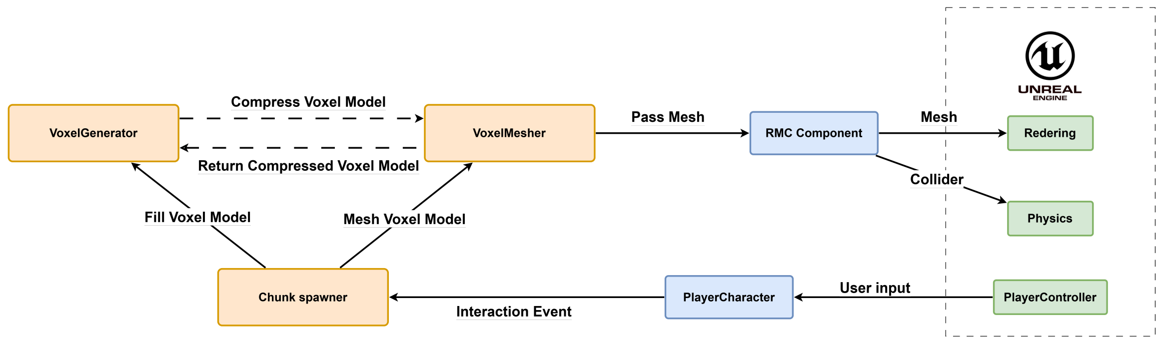


Figure 7.23: Diagram of implementation with Unreal Engine.

7.7 Conclusion

The Unreal Engine project serves as a profiling prototype that includes predefined scenarios and datasets specifically designed to test the proposed voxel meshing algorithm.

As this work is intended as a prototype, the plugin is not officially released but is available as open-source on my GitHub repository[1] and within the submitted files to this thesis. The plugin can be found under `Plugins/RunDirectionalMeshingDemo` and integrated directly into any Unreal Engine project. A README file is provided to facilitate integration and understanding of the implementation.

[1] GitHub: https://github.com/Pawlost/bachelors-thesis-compressed-meshers

Chapter 8

Performance profiling

Performance was conducted within the attached Unreal Engine project submitted in this thesis and combines UE modules and plugins introduced in this thesis to generate performance profiling results. All files are included in the thesis submission.

8.1 Scenario

Each profiling scenario is implemented as a separate Unreal Engine level designed to evaluate the performance of voxel models. In each scenario, a single voxel model type is used and instantiated multiple times within a chunk grid to generate sufficient data for performance testing. The only exception is the voxel model generated using noise, where each chunk contains a unique voxel model. All scenarios are stored under the directory `Content/ProfilingScenarios/`, organized into numbered subfolders. Each subfolder contains a custom voxel generator used in the scenario.

Additionally, the directory includes:

- `DT_Profiling_VoxelTable`: A data table defining all voxel types used across scenarios.
- `BP_Preloaded_Profiling_Spawner`: A blueprint that spawns voxel models into a chunk grid..
- `Map_Scenario_Template`: A level template used as a base for creating new scenarios.

The `Map_Scenario_Template` (see figure 8.1) provides a standardized testing environment and includes the following assets:

- Dynamic lighting (`DirectionalLight`)
- Sky box (`SkyAtmosphere`)
- Chunk spawner (`BP_Preloaded_Profiling_Spawner`)
- Player character spawn point (`PlayerStart`)
- Ground surface mesh (`Floor`)
- Test objects (`SM_TestCube`, `SM_TestSphere`)

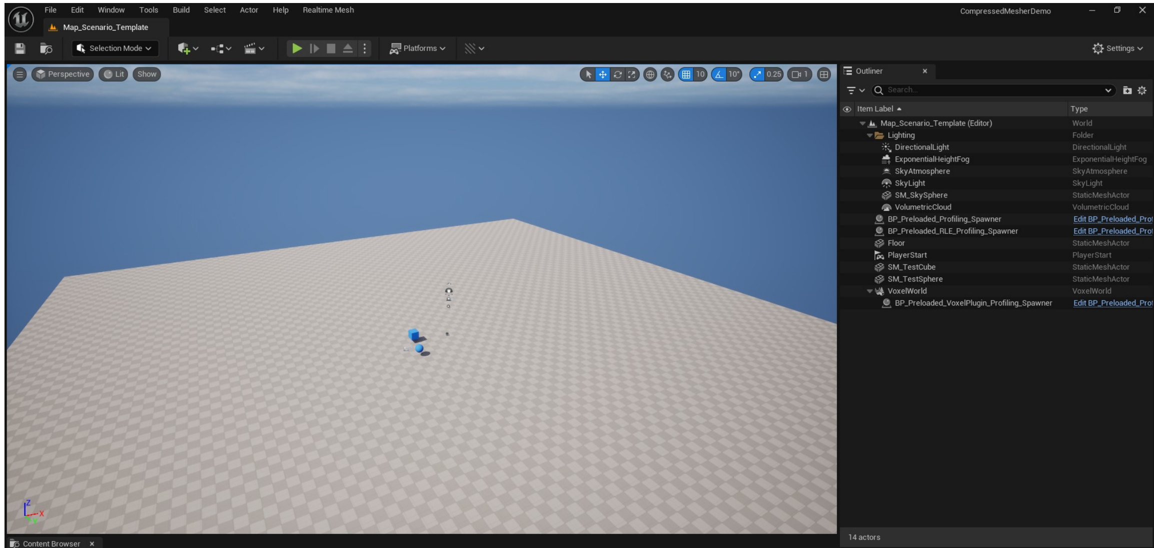


Figure 8.1: Screenshot of a scenario template.

- Voxel model generator components (`UVoxelGeneratorBase`)

The lighting and sky elements improve scene visibility. Each level includes a dedicated `Lighting` folder containing additional visual assets. The physics and collision is enabled on test objects. It serves to test collision behavior with voxel models. A player character, spawned from `PlayerStart` as `BP_VoxelInteractionObserverCharacter`, is included to test user interaction and volume changes. The ground surface ensures that the player character does not fall into empty space, providing a stable surface for testing.

Voxel generators are blueprint components derived from `UVoxelGeneratorBase` and are customized for each scenario. These components are typically attached to chunk spawners, which instantiate them in the scene. In configurations that use the Voxel Plugin, the generator may instead be a child of a `VoxelWorld` actor. In such cases, a wrapper component within `VoxelWorld` provides access to the voxel model generated by the associated generator. All voxel meshers sample a voxel model from the same spawner, even if they have different requirements for accessing a voxel model. Spawner works as an interface between voxel models and voxel meshers.

8.2 Dataset

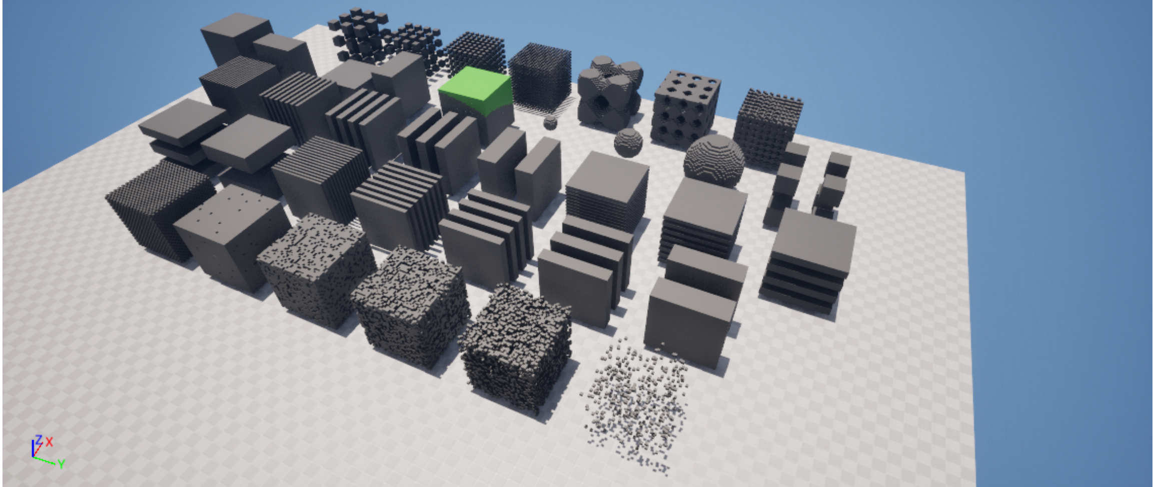


Figure 8.2: Screenshot of dataset placed into a scenario scene.

The dataset consists of two main components:

- Unreal Engine levels located in `Content/ProfilingScenarios/`
- A C++ module implementing custom voxel generators

The voxel generators are blueprints that procedurally generate voxel models and inherit from the `UVoxelGeneratorBase` class. They are used by chunk spawners to populate each scenario level with a distinct voxel model. Some generators rely on procedural noise, while others use coordinate-based logic. Additionally, Scenarios 1, 7, and 17 include variations in chunk size, with dimensions of 8, 16, 32, 64, and 128. Default chunk dimension for every scenario is 32.

There are **37 scenarios** in total, each stored in a separate subfolder (e.g., `Scenario[N]`), containing three level maps and corresponding voxel generator blueprints:

- Grid–based Run Directional Meshing (RDM)
- RLE–based RDM
- Voxel Plugin integration

Scenarios are organized based on voxel type and memory usage. A showcase level, `Map_Scenario_Showcase`, displays a side-by-side visual comparison of all voxel models (see figure 8.2).

The dataset is further divided into four distinct groups based on scenario characteristics:

- **Group 1** — Scenarios 1–17: General-purpose scenarios.
- **Group 2** — Scenarios 18–22: Scenarios with varying random voxel distributions.
- **Group 3** — Scenarios 23–37: Scenarios with voxel strips that run along or against the traversal direction.

Scenario 17 is an extreme outlier due to its checkerboard voxel pattern and will be presented in separate charts for clarity in some cases. The pattern is shown in figure 8.3.

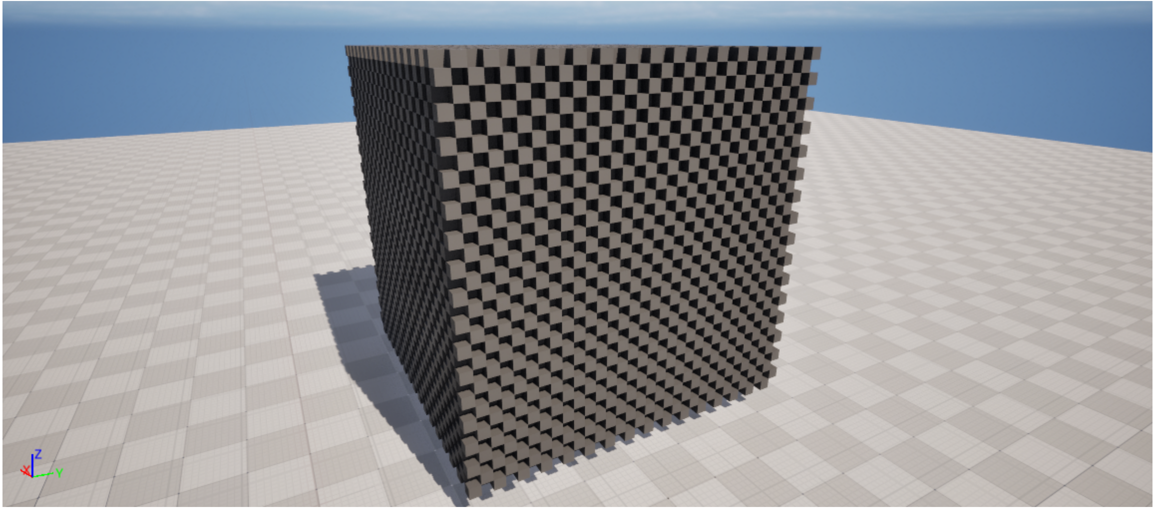


Figure 8.3: Screenshot of Voxel Model in Scenario 17.

8.3 Profiling methodology

The goal of profiling was to measure various performance characteristics of the voxel meshing algorithm, including memory usage of voxel models, frame rate (FPS) in a level, average number of generated vertices, voxel meshing speed, distribution of empty and opaque voxels, and to identify common phases during meshing. Measuring triangles on dynamically created meshes is more challenging than measuring vertices in Unreal Engine. Since these cubic meshes consist of quads that may not share vertices, the vertex count serves as a more consistent and representative metric, as it accounts for cases where vertices are duplicated rather than shared, as mentioned in the T-Junction problem.

8.3.1 Tools and Procedure

Performance profiling was carried out using Unreal Engine's built-in tools. Meshing speed was measured using Unreal Insights, with `TRACE_CPUPROFILER_EVENT_SCOPE` macros inserted into key sections of the code. These macros generate CPU tracks that define the start and end of measured code regions. Before running each test, Unreal Insights was enabled to ensure consistent data collection. The resulting tracks were exported as CSV files[1] and saved in the `PerformanceProfiling/Data/PerScenario` directory.

Frame rate (FPS) was recorded using Unreal Engine commands `StartFPSChart` and `StopFPSChart`. Once stopped, the generated log file was manually moved to the relevant scenario folder for further analysis.

To log triangle count, memory usage and opaque voxel count, a custom Unreal class `FVoxelMeshingProfilingLogger` and a dedicated log category `LogVoxelMeshingProfiling` were created. These logs were automatically generated upon Unreal Editor launch and capture data during PIE (Play In Editor). The resulting files were stored in the `PerformanceProfilingDatal` directory.

[1]CSV definition: https://en.wikipedia.org/wiki/Comma-separated_values

8.3.2 Data Processing

All exported CSV files were processed using custom Python[2] scripts located in the `ProfilingResultsPars` directory. These scripts utilized the `matplotlib`[3], `numpy`[4], and `pandas`[5] libraries to analyze the data and generate visualizations in SVG format[6]. Final charts were saved in the `PerformanceProfiling/Output` folder.

The generated graphs are presented in the following sections.

8.3.3 Scope of Measurements

It is important to note that the speed of voxel model generation (e.g., block data creation) was not measured, as it is outside the scope of this thesis. Only the performance of the conversion process from voxel data to mesh was evaluated.

8.3.4 System Specifications

Performance profiling was performed on a PC with the following specifications:

- **OS** – Microsoft Windows 11 Pro 64-bit
- **CPU** – Processor Intel(R) Xeon(R) E3-1240 v5 @ 3.50GHz, 3504 Mhz, 4 Cores, 8 Logical Processors
- **GPU** – NVIDIA GeForce RTX 3050
- **RAM** – 32GB

Unreal Engine was built in `Development Editor` configuration to measure performance.

8.4 Memory

In the implementation, a flat array of a voxel model is made of these structs:

The `FVoxel` structure requires 8 bytes of persistent memory, consisting of:

- 4 bytes for the voxel ID
- 1 byte for the transparency flag
- 3 bytes of Unreal Engine overhead

The `RLEVoxel` structure requires 12 bytes of persistent memory; it is made of:

- 4 bytes for the voxel ID
- 1 byte for the transparency flag
- 3 bytes of Unreal Engine overhead

[2] Link to official python website: https://www.python.org/
[3] Link to Matplotlib: https://matplotlib.org/
[4] Link to numpy: https://numpy.org/
[5] Link to pandas: https://pandas.pydata.org/
[6] SVG format: https://en.wikipedia.org/wiki/SVG

- 4 bytes for the run length

Only the voxel ID and run length are essential for compression. The exact memory footprint may vary by the implementation, but the relative memory differences between models will remain the same.

Transparent voxels are supported, but the algorithm is not optimized. Unreal Engine is more sensitive to the number of vertices when handling transparent objects. The generated meshes are not usable because of the impractical result.

Only persistent voxel storage is measured; temporary allocations during meshing are excluded. Dynamic memory usage during meshing can peak, approximately, at worst 7 times the voxel grid's size, but may be reduced through pooling (reuse). This number accounts for the 6 containers for faces and one for a copy of the voxel model, but the real number will be smaller. Meshing is feasible as long as the system can hold up to seven times the voxel model in a temporary memory, though actual usage is usually lower and can be reduced.

Memory statistics were logged using Unreal Engine's `TArray::GetAllocatedSize` directly from the flat array, and saved to the output log. Python scripts included with the project allow further analysis as previously described.

The following charts illustrate the variation in memory performance across different profiling scenarios, highlighting the impact of voxel sparsity on storage and compression.

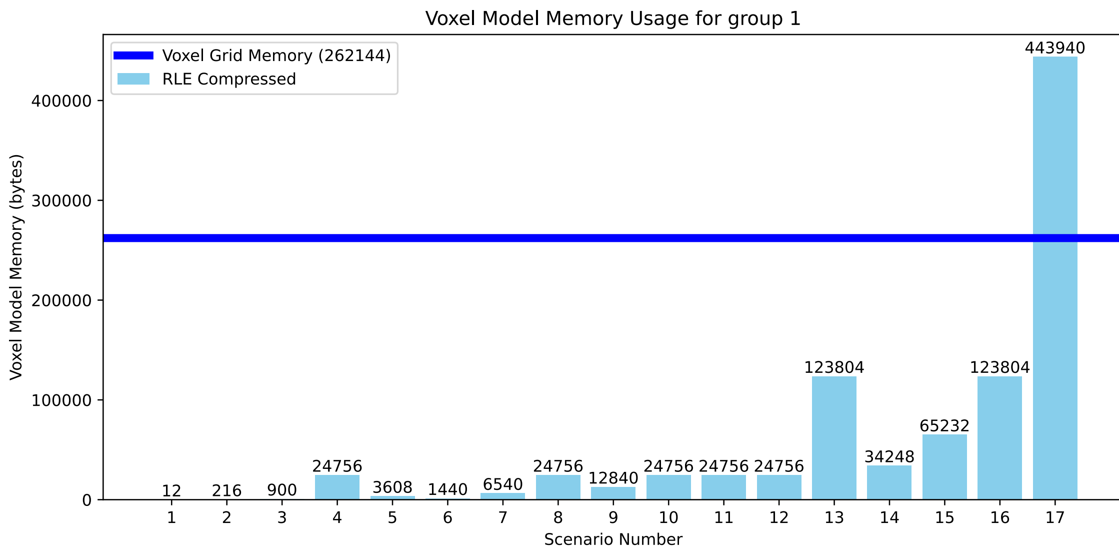


Figure 8.4: Persistent memory of voxel models in group 1.

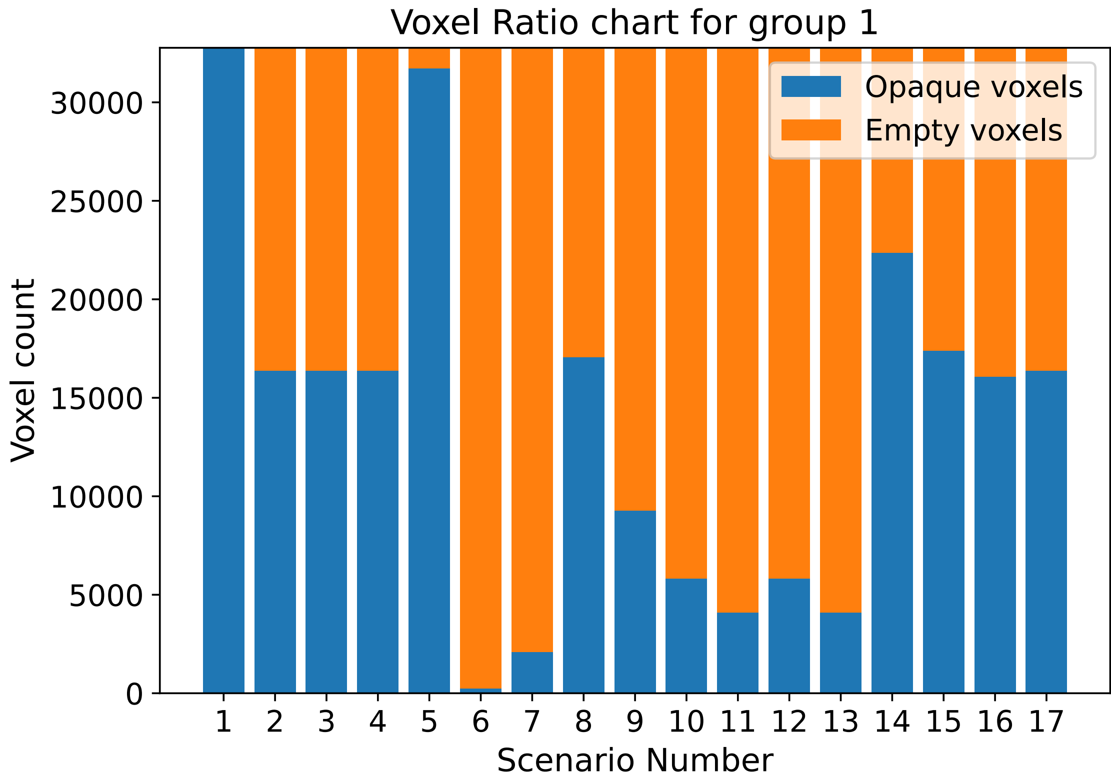


Figure 8.5: Ratio of opaque and transparent voxels in Voxel Models in group 1.

The blue line in chart 8.4 and in the following figures represents the persistent memory allocated for each voxel grid. Given that the grid dimensions remain constant across all scenarios, this memory usage does not vary unless explicitly specified. The chart legend includes the fixed memory footprint of the voxel grid, which is 262 144 bytes.

Although not immediately visible in the charts 8.4 and 8.5, memory performance deteriorates as the voxel ratio between different voxel types approaches an equal ratio. This is due to two overlapping factors that affect the efficiency of Run-Length Encoding (RLE):

1. The overall ratio between different voxel types.

2. The alignment of voxel runs with the traversal direction.

For instance, Scenarios 2 and 17 have the same ratio but differ greatly in memory usage due to differing run alignments. These factors are clarified in the following charts.

As shown in chart 8.4, some scenarios, such as Scenario 17, demonstrate increased memory usage under RLE, revealing a key limitation of the compression method. This occurs when run lengths are minimal and each voxel forms an individual run. In the case of Scenario 17, which features a chessboard pattern (see figure 8.3), no adjacent voxels share the same value, preventing the compression. As a result, RLE stores both the voxel ID and a redundant run length of 1 for every voxel, leading to higher memory usage than the uncompressed grid, where each voxel's length is implicitly 1. While other patterns could similarly increase overhead if they contain a high proportion of single-voxel runs relative to merged runs, such cases are beyond the scope of this thesis. As showcased in this thesis, a more common scenario for voxel models is a memory reduction.

In contrast, Scenario 1 demonstrates the ideal case, where a uniform voxel grid is compressed into a single run, minimizing memory usage. Despite the outliers, RLE yields an average memory reduction of **74%** in all included scenarios. The effectiveness of compression thus depends strongly on voxel sparsity and directional alignment.

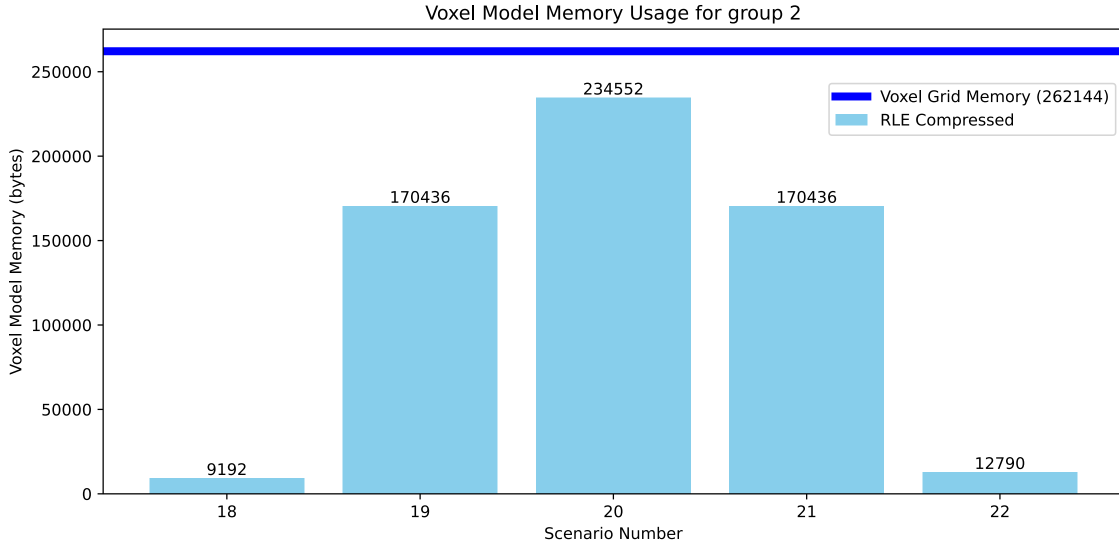


Figure 8.6: Persistent memory of voxel models in group 2.

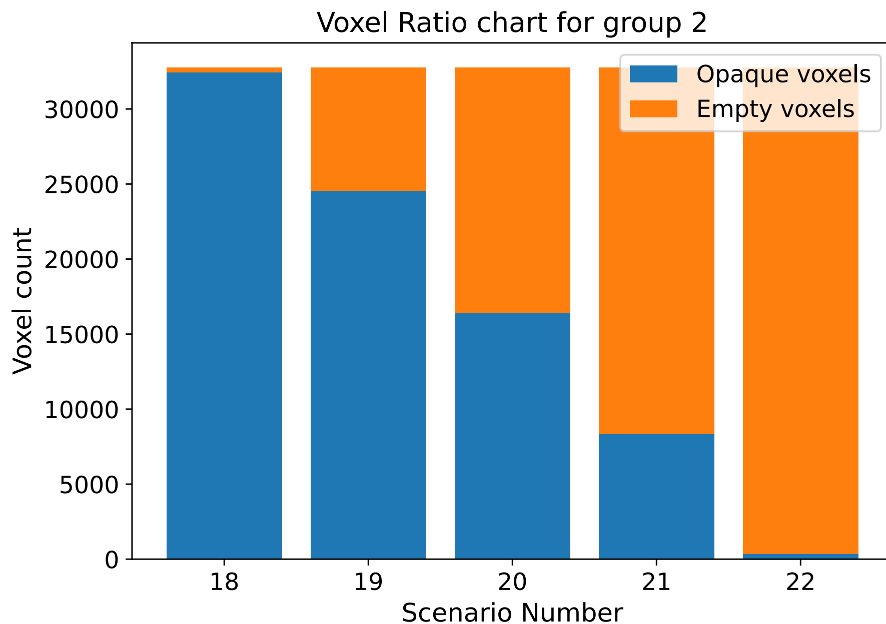


Figure 8.7: Ratio of opaque and empty voxels in Voxel Models in group 2.

In chart 8.7, we can see why this group was made. The ratio of empty to opaque voxels changes across the scenarios from voxel models nearly full of opaque voxels to nearly empty. The voxel models have a random distribution of voxels. This figure clearly illustrates

the first factor that the more the overall ratio equilizes, the less effective the compression becomes. We can see that scenario 18 is closer to maximum and scenario 22 is closer to minimum, but both have a very low memory usage as shown in chart 8.6. Scenario 20 has an equal ratio but also uses the most memory. Both charts together illustrate this trend.

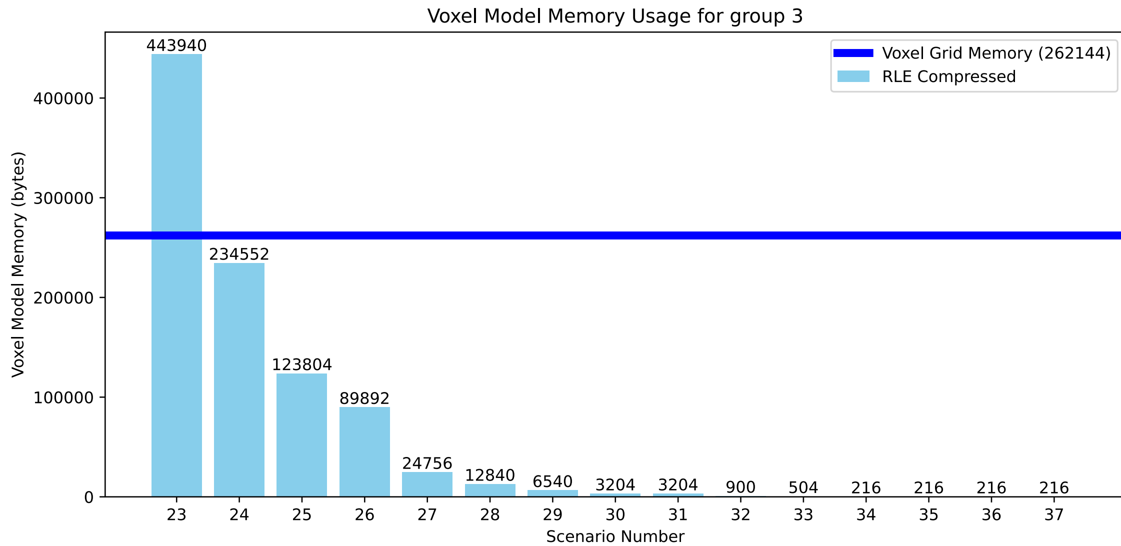


Figure 8.8: Persistent memory of voxel models in group 3.

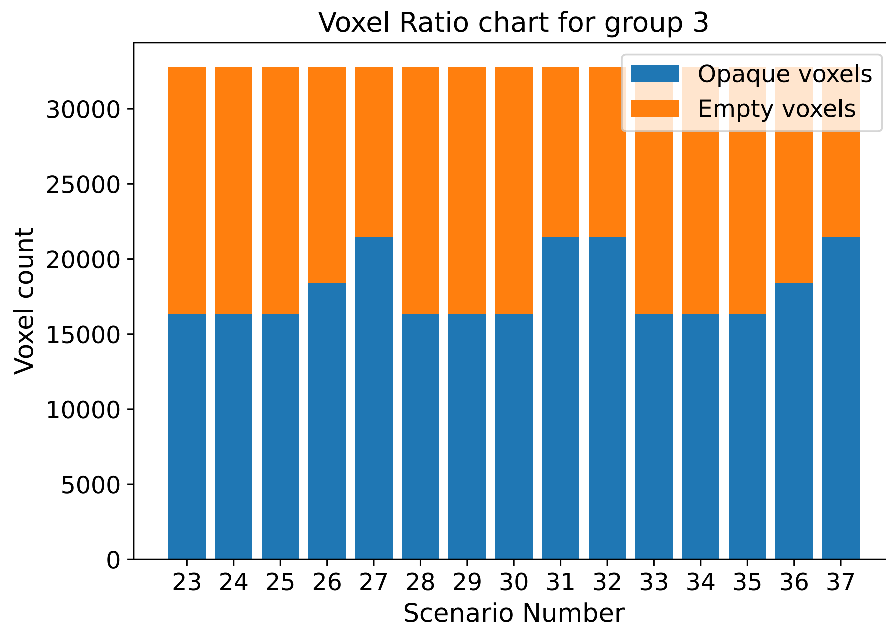


Figure 8.9: Ratio of opaque and transparent voxels in Voxel Models in group 3.

Charts 8.8 and 8.9 once again reveal a slight correlation between the compression ratio and effectiveness, supporting the claim about the impact of the first factor, even though this is not the primary focus of Group 3.

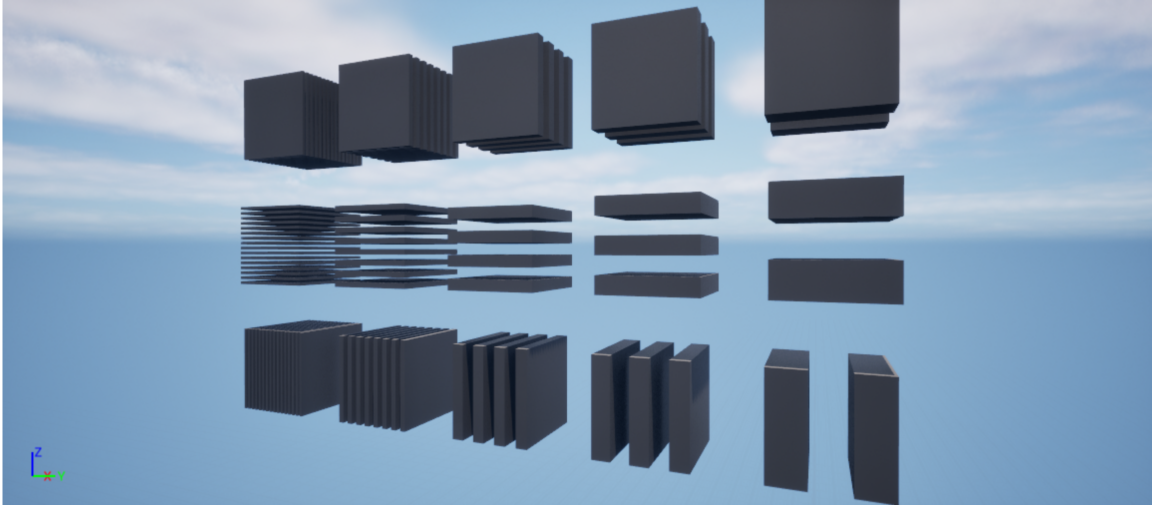


Figure 8.10: Strip voxel models in group 3.

In this group, every fifth voxel model is intentionally designed to have the same number of strips, as shown in figure 8.10. Each strip has a different size or orientation. This is to isolate and demonstrate the influence of the second factor.

Chart 8.8, which depicts persistent memory usage for Group 3, illustrates the major influence of voxel strip orientation on RLE compression efficiency. In this group, the leftmost scenario exhibits the weakest compression, while the rightmost achieves the most effective. This variation arises from how well the strips align with the primary traversal axis, which is the y-axis. Strips aligned with the y-axis, as seen in the rightmost case, produce optimal compression results. In contrast, when voxels deviate from this alignment and are instead oriented along the z or x axes, they introduce orthogonal transitions that break data continuity and substantially reduce compression efficiency, as only y-axis aligned transitions can be effectively compressed. Of these, x-axis alignment is the least efficient, as it requires two orthogonal changes relative to the primary traversal direction. Being the last and thus least prioritized dimension in the traversal order, the x-axis alignment results in the poorest compression performance.

These observations reinforce the earlier conclusion that voxel model orientation significantly affects RLE compression efficiency. Proper rotation during compression can enhance it by increasing the length of voxel runs. For example, if the model in scenario 23 were rotated to match the alignment seen in scenarios 28 or 33, as seen in figure 8.10, a more favorable compression outcome could be achieved. On the other hand, poor orientation can also degrade compression performance and increase memory usage.

However, certain cases, such as the previously discussed scenario 17, have poor compression regardless of the orientation. This model represents a worst-case scenario, where the internal structure inherently prevents efficient compression under any traversal direction. In contrast, scenario 1 demonstrates consistently good compression across all orientations due to its uniform voxel alignment, making it an ideal case for RLE compression.

The potential solution could be rotation-based optimization. This is further described in the next chapter 8, but its implementation lies outside the scope of this thesis.

Regardless of the eventual adoption of Run Directional Meshing, the inefficiencies of RLE highlighted in this work remain consistent and significant. These limitations must be addressed in any future work about RLE-based compression usage for voxel models.

8.5 FPS and Vertices

This section presents charts illustrating the impact of vertex count on average FPS in an Unreal Engine level. Vertex counts are measured per chunk, taken from Unreal Engine's vertex buffer structures before rendering, and averaged.

The charts compare triangle counts across different meshing algorithms. Frame per second (FPS) is measured only in selected high-vertex scenarios, as lower-vertex cases inherently perform better. Because all meshing is done asynchronously, only the number of vertices and triangles directly affects the FPS.

Additionally, the Grid algorithm demonstrates cross-chunk culling when operating on multiple voxel models arranged in a chunk grid.

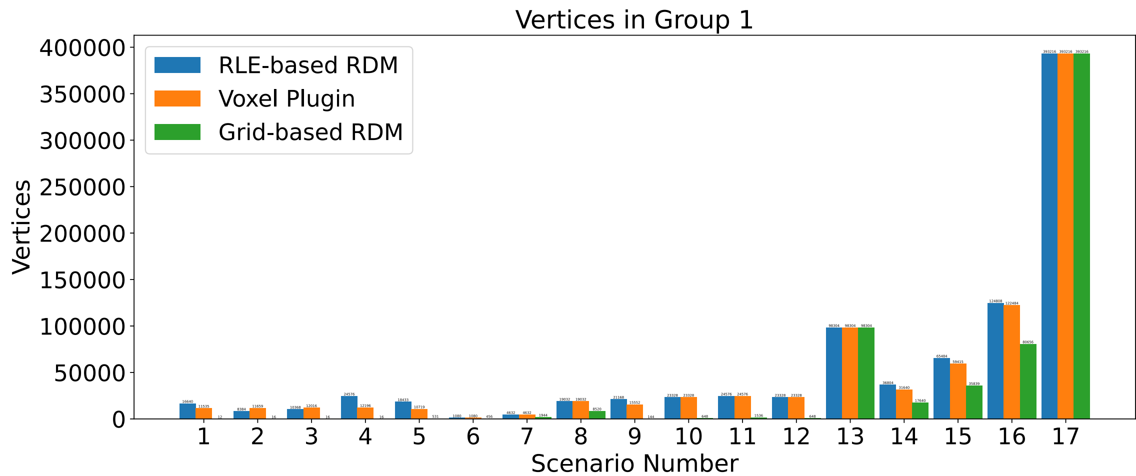


Figure 8.11: Vertices in Voxel Grid Scenarios.

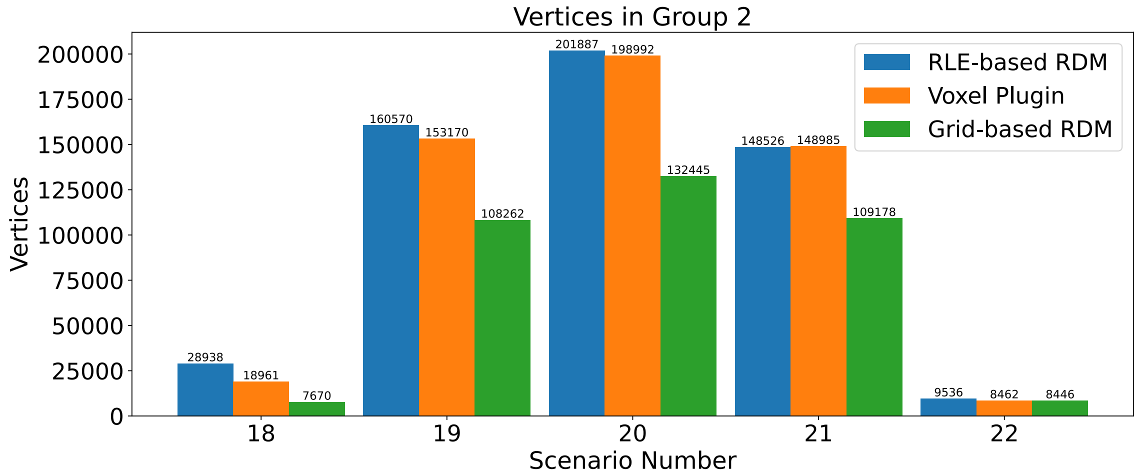


Figure 8.12: Vertices in Voxel Plugin Scenarios.

A key trend observable in figures 8.11 and 8.12 is the consistent vertex reduction achieved by the Grid-based RDM algorithm. In all scenarios, Grid-based RDM outperforms the other algorithms in terms of vertex efficiency, even in the most challenging cases such as scenarios 13 and 17, as can be seen in figure 8.11. In these cases, it produces the same number of vertices as the other methods, which is 98304 in scenario 13 and 393216 in scenario 17, without exceeding them. Among these, scenario 17 is the most computationally demanding overall. Conversely, the lowest vertex count generated by Grid-based RDM occurs in scenario 1, where only 12 vertices are produced. For the Voxel Plugin and the RLE-based RDM, the minimal vertex count is observed in scenario 6, with 1080 vertices. This reduction is attributed to the high proportion of empty voxels that do not contribute to the visible surface as previously discussed in Section 8.4.

A similar trend is visible in scenario 22 (figure 8.12), which is characterized by a largely empty volume. This results in a substantial reduction in the number of generated quads. However, Grid-based RDM generates even fewer vertices in scenario 18 than in scenario 22, due to more effective quad merging. In scenario 22, the structure of the voxel model limits such merging.

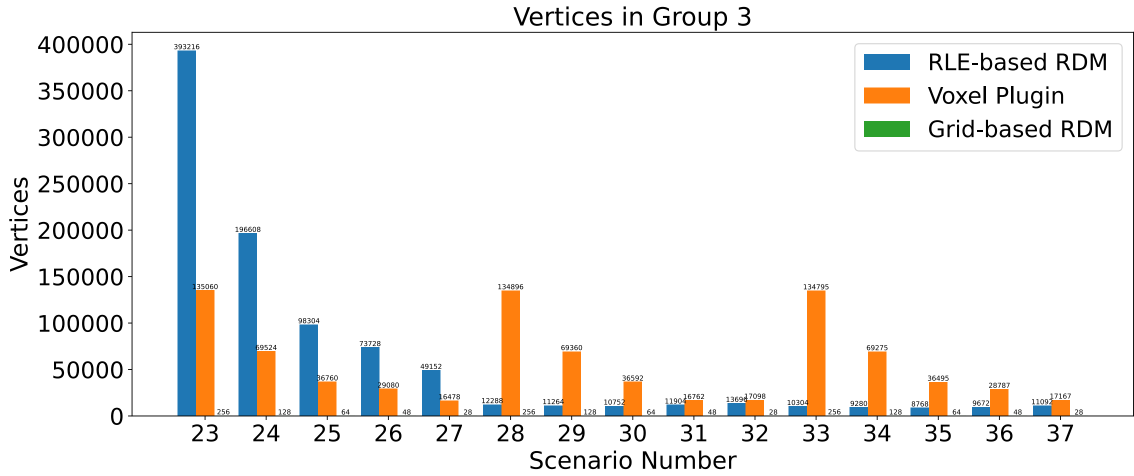


Figure 8.13: Vertices in Voxel Grid Scenarios.

Figure 8.13 demonstrates how the orientation of a voxel model significantly impacts both vertex count and mesh structure when using the RLE-based RDM algorithm. This effect arises primarily from the algorithm's limited face culling capabilities in the current implementation. Voxel models aligned along the x-axis exhibit the least effective culling, resulting in either higher vertex counts or reduced performance, depending on whether face culling optimization is fully integrated. When the optimization is fully integrated, it improves vertex reduction at the cost of conversion speed along the x-axis and z-axis, whereas without optimization, the vertex count is significantly increased. In contrast, y-axis aligned voxel models benefit from more natural run culling and greater compression efficiency, consistently delivering better performance. This is especially noticeable from scenario 28 onward. These orientation-dependent factors thus influence both memory usage and voxel meshing performance. Notably, sequences aligned along the x-axis significantly degrade the performance of the RLE-based RDM approach.

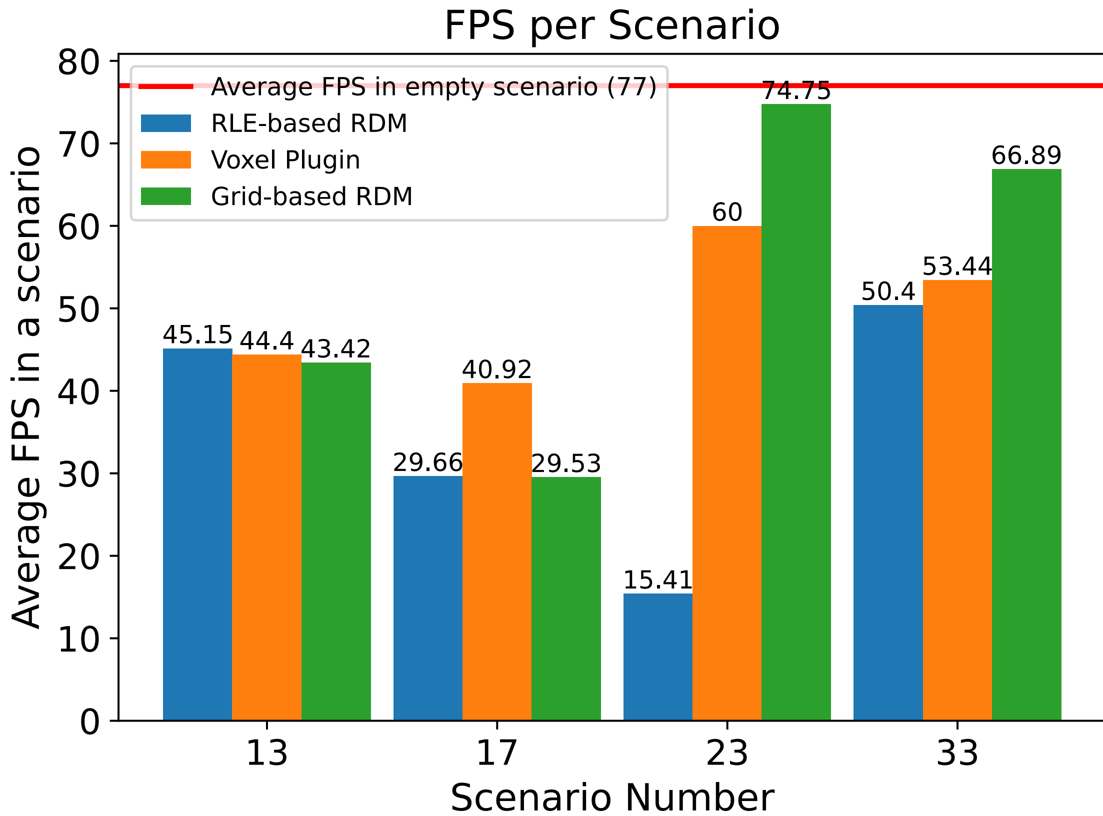


Figure 8.14: FPS performance in selected scenarios.

Figure 8.14 presents selected scenarios used for FPS performance profiling. The red line indicates the baseline FPS, measured from an empty scene template made from a scenario located template in `ContentProfilingScenarios`, with all generators removed. This baseline, representing the maximum achievable FPS on the test machine, which is **77**, and the corresponding log is stored at `PerformanceProfilingDatafps_empty_template_scenario.log`.

Scenarios were chosen based on distinct performance characteristics:

- **Scenario 13:** Balanced vertex count across algorithms (98304), representing a mid-range test case.
- **Scenario 17**: Worst-case FPS across all algorithms.
- **Scenario 23**: High vertex count due to inefficient culling in RLE meshing.
- **Scenario 33**: High vertex output from the Voxel Plugin.

Scenario 13 demonstrates that 98304 vertices result in an average of 45 FPS, suggesting that configurations producing fewer vertices will yield higher FPS, while higher counts, such as 393216 in Scenario 17, reduce performance to 30 FPS. These values help approximate a linear relationship between vertex count and FPS.

Interestingly, Voxel Plugin benefits from LOD optimizations which can boost performance above 40 FPS even at higher vertex counts, though this introduces visual artifacts

(figure 8.15). Without LOD, the performance would likely drop by about 30 FPS as the rest but would correctly represent the voxel models.

Scenario 23, is characterized by minimal culling in the RLE-based RDM algorithm. It demonstrates a drop to 15 FPS, I declare, that this is below a threshold for interactive applications. Despite having similar vertex counts, the Voxel Plugin outperforms RLE-based RDM due to its ability to cull internal geometry and generate only visible surfaces. In contrast, RLE-based meshing suffers from additional overdraw caused by long thin triangles and geometry culling that needs to be performed on a GPU.

Grid-based RDM achieves lower or comparable vertex counts across all scenarios. We can see direct result of this in scenario 23 and 33. Particularly in scenario 23 the framerates level is almost the same as in an empty scene. While the RLE-based approach has room for optimization, Grid RDM currently offers superior balance of mesh compactness and runtime efficiency. The Voxel Plugin sometimes produces small trinangles that may lead to overdraw or visual artifacts in some LOD cases. However, because of this aggressive LOD generation it achieves stable framerate across scenarios unlike RDM algorithms.

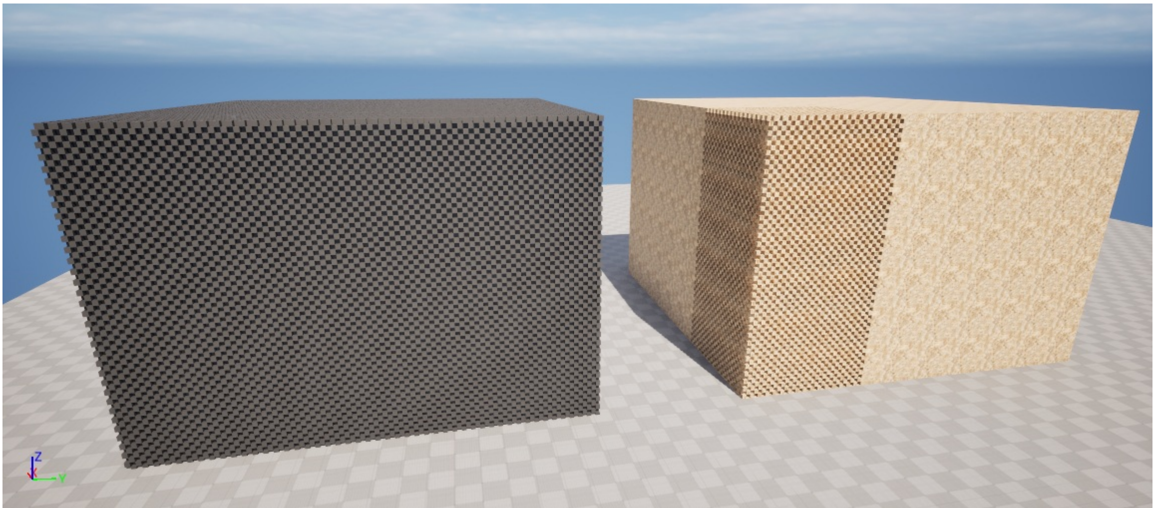


Figure 8.15: Voxel Plugin LOD artifact in voxel model taken from scenario 17. Left side is a voxel model generated using Grid-based RDM on the right side is a voxel model generated using Voxel Plugin

8.6 Voxel Meshing Performance Profiling

The performance data presented in the following charts was collected using Unreal Insights, Unreal Engine's built-in profiling tool. To enable measurement of the voxel meshing process, custom macros were integrated into the codebase to mark the entry and exit points of relevant code scopes. Charts were generated by processing the exported CSV data. For each scenario, the duration of each scope was computed by subtracting the start time from the end time, and the results were then averaged or otherwise aggregated, depending on the specific analysis requirements. This, therefore, represents the speed at which vertices are made from voxel representation. The process of data extraction into charts can be seen in the attached Python scripts used for aggregating the data.

The resulting profiling data are presented in three distinct types of charts:

- **Phase Breakdown Charts** — Illustrate the time spent in each common stage of the voxel meshing algorithms.
- **Timing Statistics Charts** — Display the median, average, minimum, and maximum speed of the voxel meshing process in various scenarios.
- **Scalability Charts** — Demonstrate how the voxel meshing performance scales with increasing chunk sizes.

8.6.1 Phase Breakdown

I identified three core phases that are common across all evaluated meshing algorithms. Potential optimization strategies are discussed for each phase in the next chapter.

1. **Voxel Meshing** — Phase of iterating through voxel data and generating quads as a surface geometry.
2. **Insertion into UE Buffers** — Transferring generated mesh data into vertex, normal, and index buffers of Unreal Engine.
3. **Other timings** — Time spent on auxiliary tasks such as memory allocation, data optimization, or other overhead.

In addition to these general phases, the Voxel Plugin introduces a unique phase due to its reliance on an octree data structure. Since it stores voxel data within an octree and samples from it during mesh generation, the time spent traversing and querying the octree constitutes a significant and distinctive component of this algorithm's performance profile. In the following charts, *VP* stands for voxel meshing in Voxel Plugin, *RLE* stands for RLE-based RDM, and *Grid* stands for Grid-based RDM.

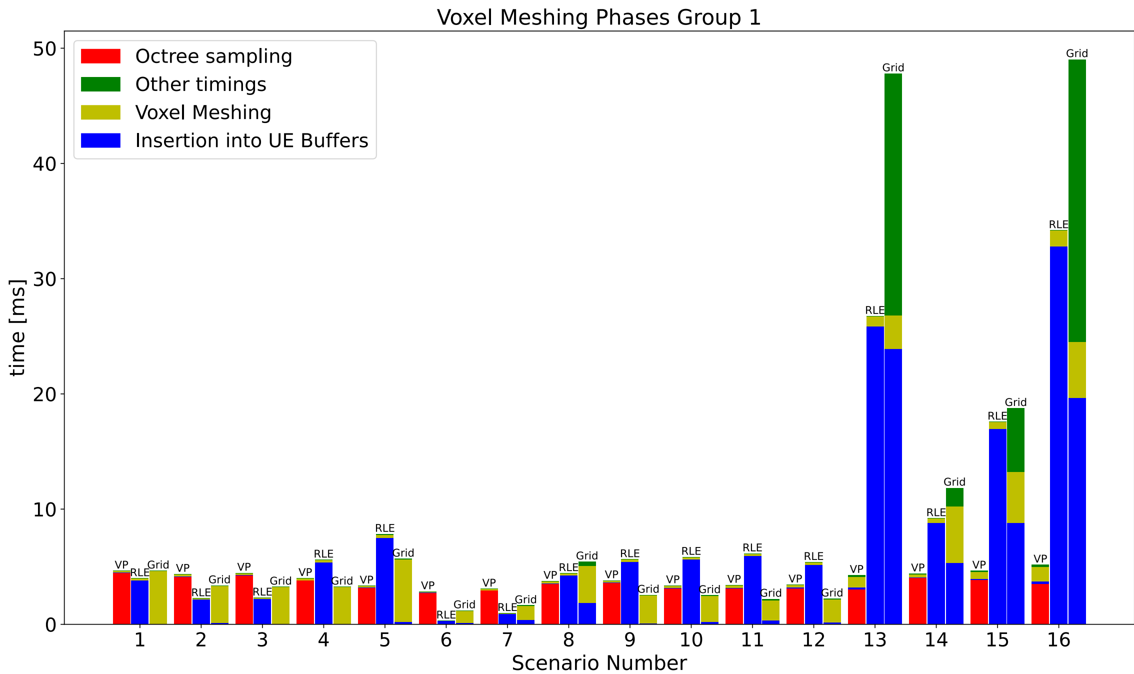


Figure 8.16: Phases Voxel Meshing algorithms in Group 1

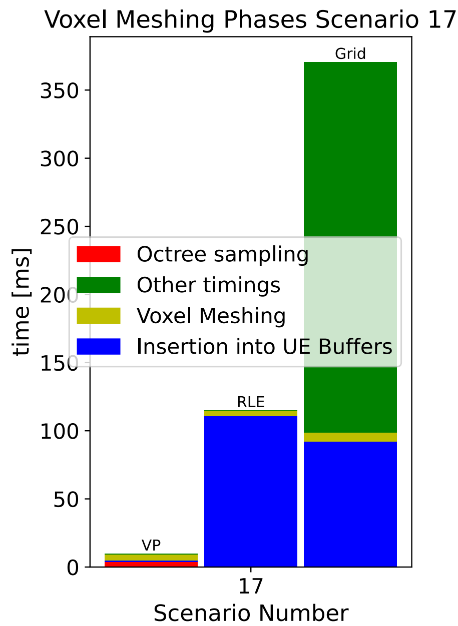


Figure 8.17: Phases in scenario 17.

To improve clarity, scenario 17, an extreme performance outlier, has been moved to a separate figure (8.17). Alongside figure 8.16, these figures illustrate the average speed of different phases in voxel meshing algorithms across general-purpose scenarios.

In most scenarios, the voxel meshing in Voxel Plugin performs comparably to Run Directional Meshing. While in scenarios like, for example, 2, 3, and 6, it even performs better. However, while the plugin maintains stable performance across all scenarios, RDM's efficiency varies a lot depending on the voxel model structure in a scenario.

Octree sampling time and the other times for voxel meshing in Voxel Plugin remain nearly constant. Although it is the least performant phase overall in this algorithm, it does not significantly degrade total performance. Starting with scenario 13, RDM begins to struggle, particularly its Grid-based variant is significantly worse. In these cases, processing times can increase from approximately 8 ms to 50 ms, which is a degradation of roughly 625%. Scenario 17 shows the most severe performance drop, spiking up to 350 ms, while the Voxel Plugin remains unaffected. This corresponds to a performance degradation of up to 4375%, indicating that RDM can experience substantial slowdowns in extreme scenarios.

RLE-based RDM is particularly efficient in sparse voxel models, such as in scenarios 6 and 7. Its use of run-length compression successfully reduces iterations through the voxel model, resulting in faster voxel meshing. However, it also produces a large number of redundant vertices. The overhead of adding these vertices to Unreal Engine's buffers becomes the primary bottleneck. This suggests that reducing the number of generated vertices could significantly improve its performance, potentially making RLE-based RDM the fastest method overall if these vertices are removed.

Both RDM variants are highly sensitive to the number of generated vertices. In scenarios like 13, 15, and 17, increased vertex counts correspond to overall slower performance. Since memory allocation remains stable across scenarios, it can also be concluded that the effectiveness of vertex reduction optimizations (indicated by the green bar) in Grid-based RDM becomes another limiting factor.

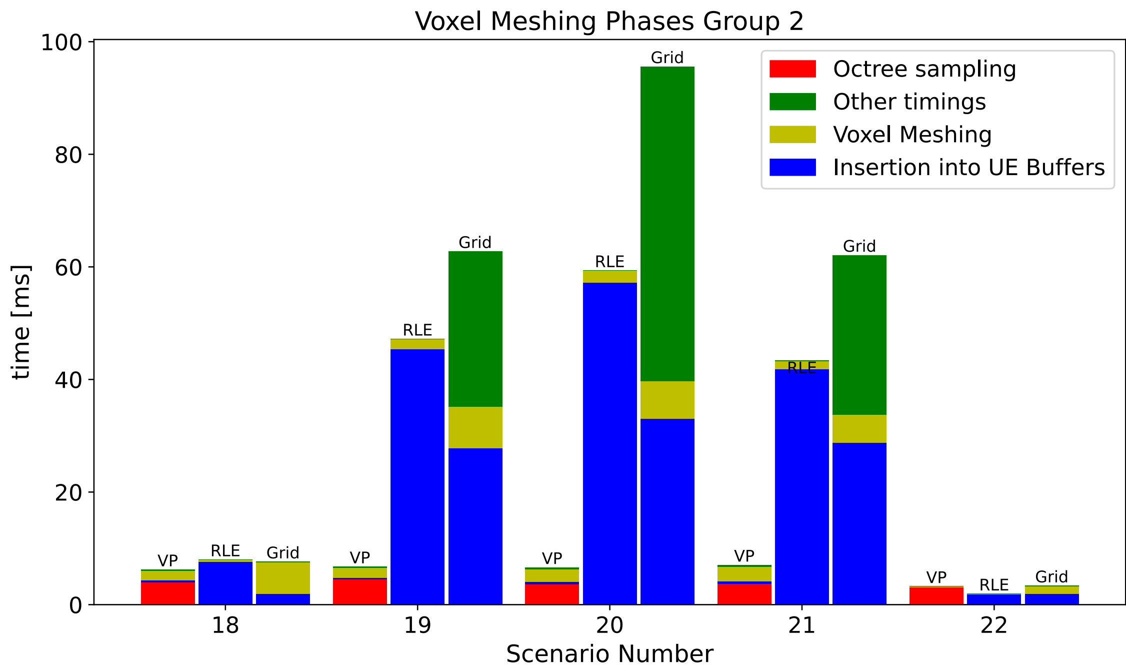


Figure 8.18: Phases Voxel Meshing algorithms in Group 2.

Figure 8.18 provides further evidence that RDM performance deteriorates as the ratio of opaque to empty voxels approaches equal values. This effect, previously stated as factor 1 in section 8.4, is now clearly visualized through the corresponding performance decline in the chart.

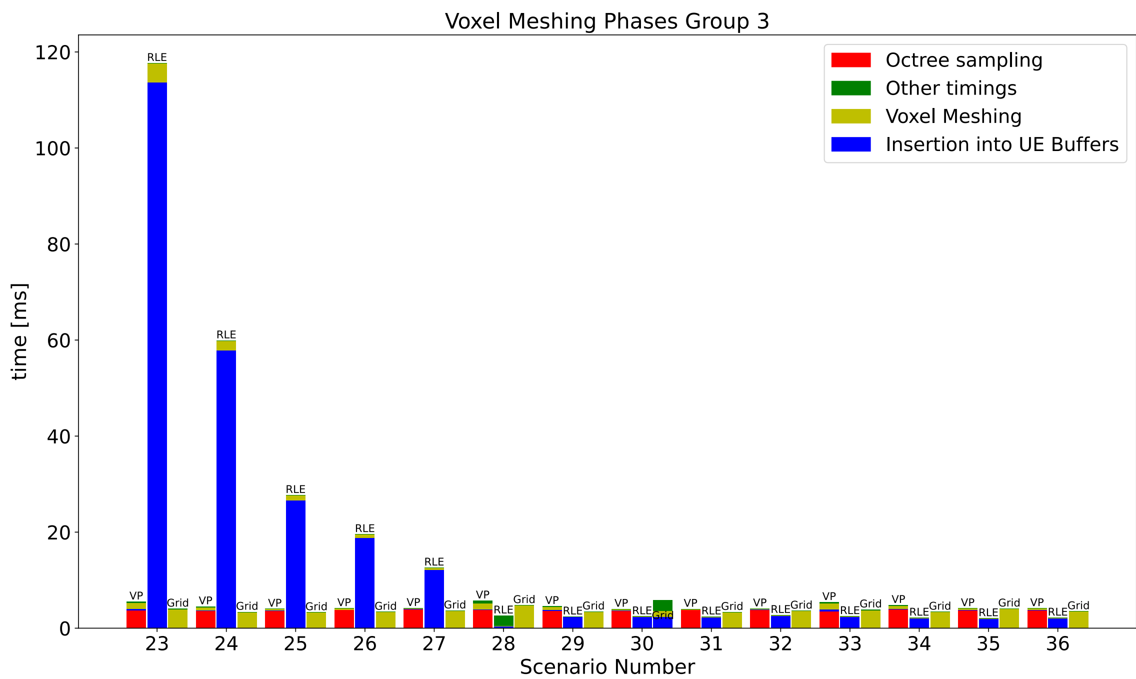


Figure 8.19: Phases Voxel Meshing algorithms in Group 3.

Figure 8.19 illustrates the performance impact of factor 2 discussed earlier. The chart clearly shows how compression misalignment affects the RLE-based RDM variant. In cases of misaligned runs, processing time increases dramatically from 5 ms to 120 ms. This results in a 2400% speed degradation due to inefficient run-length encoding.

The stable performance of the Voxel Plugin is largely due to its highly efficient, continuous parallel sampling and built-in LOD system. Unlike RDM, which processes the voxel model once, the plugin resamples voxel models continuously as the player moves. Based on CSV entries, for every 20 samples performed by RDM, the plugin executes approximately 400 lightweight, parallel voxel sampling and voxel meshing processes. This results in consistent performance across all scenarios and a near-constant meshing state. Thanks to its efficient parallelization, this frequent resampling does not negatively impact interactivity with the voxel model.

8.6.2 Timing events

This subsection highlights a trend of the two aforementioned factors on RLE-based RDM. Since Groups 2 and 3 isolate these factors within specific scenarios, only their results are considered here to better observe the effects on voxel meshing. The charts are exclusively on the RLE-based RDM algorithm, as it represents the core contribution of this thesis.

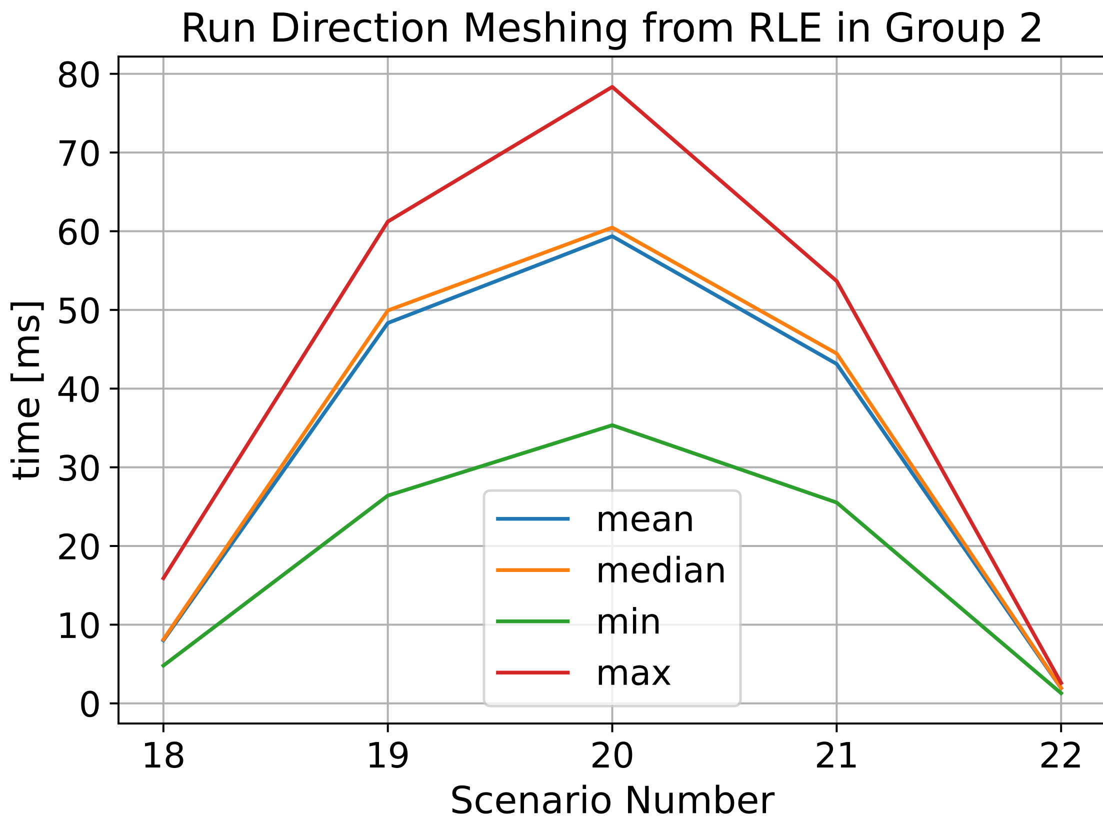


Figure 8.20: Voxel Meshing Speed of Run Directional Meshing from RLE in Group 2.

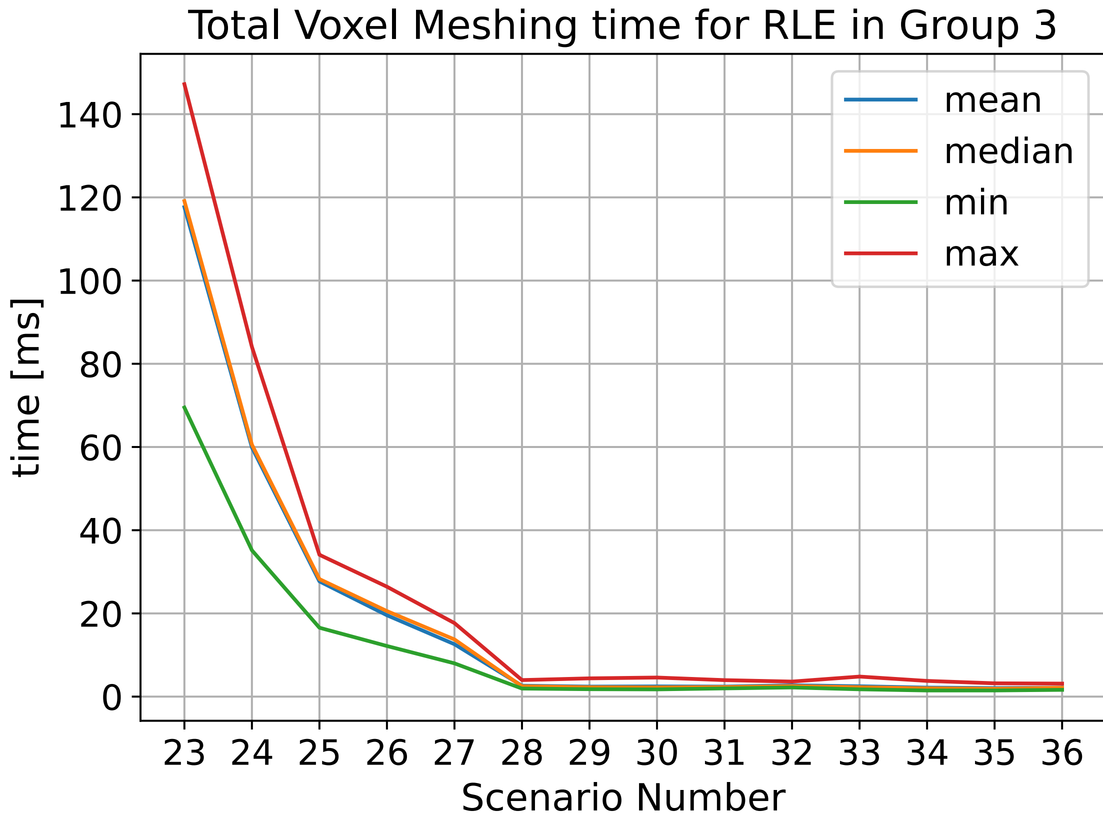


Figure 8.21: Voxel Meshing Speed of Run Directional Meshing from RLE in Group 3.

In figures 8.20 and 8.21, the mean, minimum, maximum, and median execution times all follow the same consistent trend, indicating a systematic degradation in performance. Median time follows almost exactly the mean time in figure 8.21. This pattern becomes clear when comparing figure 8.20 with figure 8.7, and figure 8.21 with figure 8.8. The decline is part of the recurring factors correlated with varying levels of voxel sparsity and compression across different scenarios. We can also observe that the more extreme a scenario is, the higher the difference between the best and the worst time. This means that the visualization of changes after interaction with a complex voxel model may have more unpredictable timing than interacting with a simpler voxel model.

8.6.3 Chunk Scalability

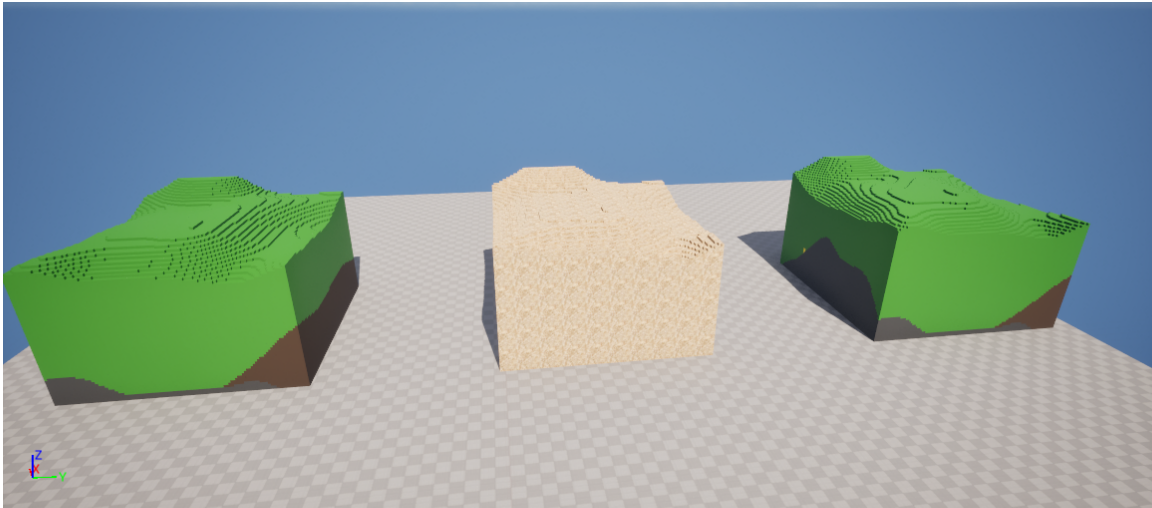
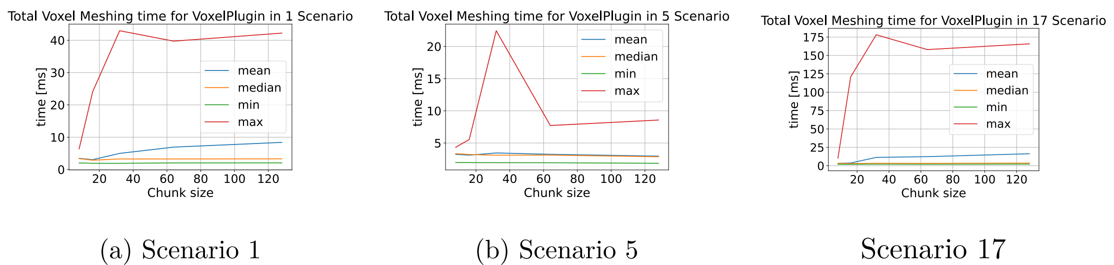


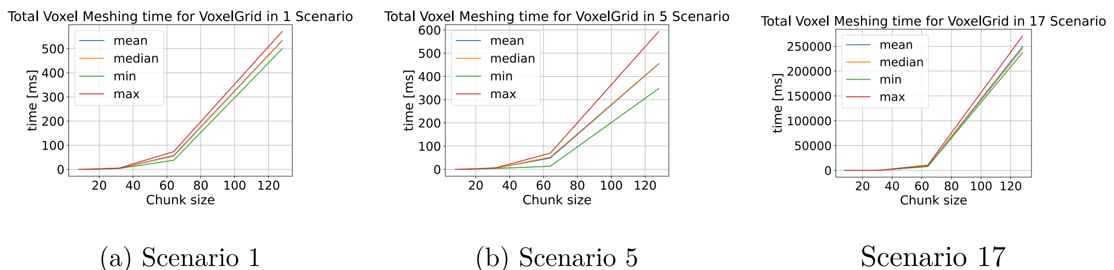
Figure 8.22: Screenshot of all voxel meshing algorithms in Scenario 5

This subsection examines the impact of increasing chunk size on voxel meshing performance. A selection of representative scenarios was chosen to illustrate this effect. Scenario 1 reflects a general best case. Scenario 5 is based on noise-generated voxel models and offers the most accurate representation of an average voxelized model, as demonstrated in the figure 8.22. Scenario 17 represents the worst-case scenario for voxel meshing. Used chunk sizes are 8, 16, 32, 64, 128.



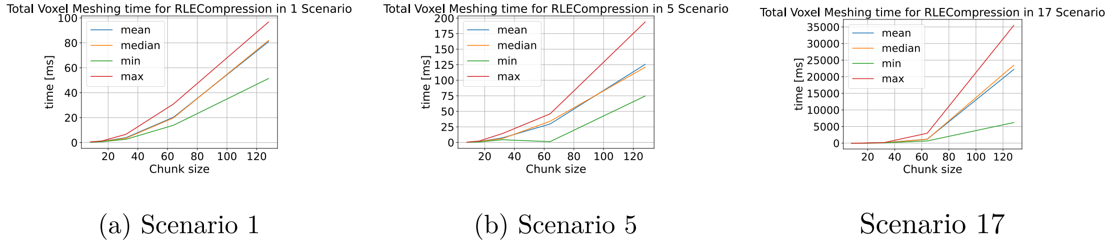
(a) Scenario 1 (b) Scenario 5 Scenario 17

Figure 8.23: Influence of different chunk sizes on Voxel Plugin across scenarios.



(a) Scenario 1 (b) Scenario 5 Scenario 17

Figure 8.24: Influence of different chunk sizes on Run Directional Meshing from Grid across scenarios.



(a) Scenario 1 (b) Scenario 5 Scenario 17

Figure 8.25: Influence of different chunk sizes on Run Directional Meshing from RLE across scenarios.

Figures 8.23, 8.24, and 8.25 present the total meshing times across different algorithms and chunk sizes. In figure 8.23, the Voxel Plugin implementation demonstrates consistent meshing performance regardless of chunk size, with the exception of the maximum meshing time, which varies depending on the scenario and chunk size.

In contrast, the Grid-based RDM results shown in figure 8.24 display a more exponential trend, with values clustered more tightly than those seen in the RLE-based variant in figure 8.25. The RLE-based algorithm exhibits greater variability, particularly in its minimum and maximum values.

At a chunk size of 128, performance disparities become especially evident: the Voxel Plugin completes meshing in approximately 40 ms, whereas the grid-based RDM takes around 500 ms, and the RLE-based RDM achieves approximately 100 ms in the best-case scenario (Scenario 1). These results approach the upper limits of what is acceptable for real-time interaction. Notably, in the worst-case scenario for chunks of this size, meshing times extend into the range of several seconds, rendering real-time use entirely impractical. Based on these findings, chunk sizes of 128 are considered unsuitable for real-time applications using RDM. This emphasizes that RDM algorithms can become inefficient under certain voxel model scenarios and that their performance scalability diminishes more rapidly than that of the Voxel Plugin as chunk size increases.

Moreover, Scenario 5 consistently exhibits increased performance divergence across all figures. This effect is attributed to the use of pseudo-random voxel models, which introduce greater structural complexity and irregularity. These characteristics lead to heightened variability in meshing performance, as reflected in the broader dispersion of data trajectories within the charts. But this dispersion reflects regular use in practical scenarios.

8.7 Performance Profiling results

In figure 8.26, we can see that the unoptimized RLE-based RDM method achieves an average voxel meshing time of approximately 20 milliseconds, including all worst-case scenarios. Both code bases of RLE-based and Grid-based RDM variants remain unoptimized in the current implementation. Among the tested methods, the Voxel Plugin delivers the fastest results, averaging around 5 milliseconds per voxel meshing process. While Grid-based RDM is the slowest at approximately 25 milliseconds, it excels in minimizing vertex count.

Voxel meshing efficiency is defined by how quickly a polygonal surface can be represented by a mesh using the least number of polygons. The memory requirements of persistent storage are not important for voxel meshing efficiency as it is a conversion process. The ideal point of any method is in the bottom-left corner of the figure 8.26. This means balancing low vertex count and voxel meshing time to maintain playable framerates without excessive

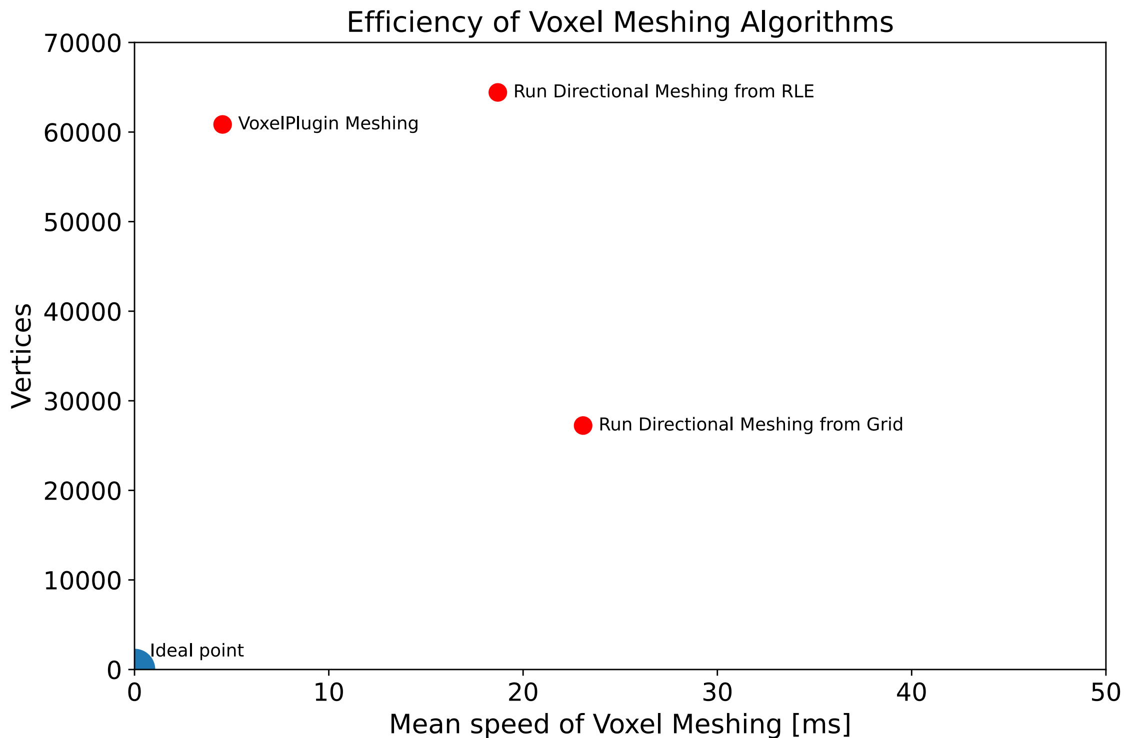


Figure 8.26: Efficiency of Voxel Meshing Algorithms.

geometry. This figure visualizes the trade-off between speed and output complexity across all tested approaches.

The Voxel Plugin mesher has a consistent speed and performance, thanks to efficient voxel sampling and simple geometry supported by LODs. Though it does not minimize vertex count as effectively as Grid-based RDM, its fast runtime makes it highly practical. Grid-based RDM offers the most optimized meshes but suffers from scalability issues and inconsistent performance.

RLE-based RDM significantly reduces iteration steps through compression, enhancing voxel meshing speed. However, this advantage comes at the cost of difficult vertex count reduction. Right now, the implementation lacks essential optimizations such as face culling and quad merging. The resulting excessive geometry places a heavy burden on Unreal Engine's buffer creation pipeline, leading to slower conversion. This is because fully optimizing the algorithm is not an important assignment for this thesis. The compression improves iteration and storage memory, but makes it harder to access voxels in an area. In contrast, Grid-based RDM includes built-in optimizations that generate highly efficient meshes, although these same optimizations can impair performance in worst-case scenarios. Despite its current drawbacks, RLE-based RDM is already more memory-efficient than the usual voxel grids. If the vertex count were reduced through the integration of geometry optimizations and buffer handling improved, RLE-based RDM could not only match the mesh quality of Grid-based RDM but also outperform most algorithms in overall meshing speed, thanks to the reduced iterations. This shows immense potential for future work and improvements. Even if it can be considered only a prototype yet, it already proves this approach to voxel meshing from compressed representation is viable.

However, according to the assignment of this thesis, based on current performance measurements, I must select the **Voxel Plugin as the fastest currently available voxel meshing algorithm in Unreal Engine**.

Chapter 9

Conclusion

This thesis presented a new voxel meshing technique called **Run Directional Meshing**, designed to convert voxel models into polygonal meshes within the Unreal Engine framework. The algorithm was implemented in two forms: one utilizing a conventional flat voxel grid and the other operating directly on Run-Length Encoded (RLE) compressed data. The goal was to explore whether compressed voxel representations can be meshed efficiently and interactively, eliminating the need for full decompression and redundant traversal operations.

RDM introduces a unidirectional pass traversal pattern to reduce iteration overhead, enabling efficient face generation and quad merging. In the Grid-based variant, RDM achieved excellent vertex reduction and demonstrated strong runtime performance. The RLE-based variant, although currently a prototype, demonstrated the feasibility of direct voxel meshing from compressed input, offering significant memory advantages.

Despite its unoptimized state, the RLE-based implementation already shows potential to rival or exceed existing voxel meshing methods in terms of speed, provided further geometry optimization and integration with Unreal Engine's vertex buffers are implemented. This fulfills the second assignment objective: to propose a new algorithm capable of operating on compressed voxel data while preserving interactivity.

Performance profiling indicated that, in its current state, the Voxel Plugin remains the fastest available voxel meshing solution in Unreal Engine. However, RDM, particularly in its grid-based form, offers highly competitive mesh quality. As such, RDM may serve as a foundation for further work into interactive voxel meshing pipelines, especially in scenarios where memory efficiency is critical.

9.1 Future optimization

Although optimization was beyond the scope of this thesis, several potential improvements could be explored in future work.

- **One pass insertion into Unreal Engine buffers**: During voxel meshing this can be done by eliminating the current two-step process and instead performing all operations during a single voxel meshing pass. This can be achieved by leveraging the cumulative properties of merged quads.

- **Double-pass RLE Traversal**: Performing a forward and reverse pass at the same time during a single iteration through RLE compression could resolve ambiguity during culling for certain voxel faces that grow in a negative direction.
- **Run Return Culling**: This is a necessary step for eliminating internal faces and is essential for advancing the algorithm from a prototype to a minimum viable product. Similar to the mesh merging approach used in grid-based RDM, a run should return to check whether a face should be culled. This process could be implemented more elegantly if combined with the double-pass optimization technique.
- **Single Layer Decompression**: Test if decompressing only on a layer of a chunk could be more efficient than potentially returning.
- **Bitwise Optimizations**: Enhancing the current implementation through binary-efficient and bitwise operations could improve overall performance.
- **Cross-Chunk Culling**: A potential optimization involving a helper structure, such as a quadtree or interval tree, to improve handling of outer faces in cross-chunk culling scenarios.

9.2 Future Work

Several branches exist for extending the work initiated in this thesis:

- **Optimization of RLE-based RDM**: Many techniques successfully applied in Grid-based RDM, such as face culling and quad merging, can be adapted to the compressed version.
- **GPU-side implementation**: Offloading the RLE-based approach to the GPU could be an interesting case study, particularly for profiling its impact on GPU memory usage reduction.
- **Dynamic Compression Rotation**: Compressing chunks along multiple axes and selecting the optimal one for meshing could improve persistent storage. How to efficiently implement this remains a topic for future work and discussion.
- **Stack-Based Edit Indexing**: Voxel modifications can be optimized by organizing them in a stack that prioritizes changes based on their execution order. This approach would enable multiple voxel edits to be processed efficiently in a single pass.
- **Comprehensive performance profiling of alternative voxel meshing algorithms**: Several voxel meshing algorithms referenced in this thesis are not currently available for the latest version of Unreal Engine. These could be re-implemented and benchmarked using the dataset introduced in this work. Additionally, their performance could be evaluated in conjunction with decompression from specific states to simulate real-world usage scenarios.
- **Evaluation under networked conditions**: The performance of the Run Directional Meshing (RDM) algorithm could be further assessed in networked environments to reflect more realistic use cases where communication latency and synchronization are relevant factors.

- **Development of custom compression techniques**: By leveraging the concept of edit indexing, an intermediate representation could be designed that enables direct interaction through index-based modification. This approach opens up possibilities for novel compression schemes that efficiently track and encode changes in voxel data.
- **RLE voxelization**: A future work could focus on voxelizing directly into a Run-Length Encoded format to reduce memory usage and improve performance. This would require integration into the voxelization stage itself.

9.3 Closing Words

This thesis presents a technical contribution to real-time voxel meshing, introduces a performance profiling framework for voxel meshing algorithms in Unreal Engine, and offers a practical examination of the key performance factors that influence these algorithms. The proposed method, Run Directional Meshing, demonstrates that interactive voxel meshing can be performed directly on compressed voxel data. This advancement contributes to the field of voxel-based rendering and highlights the advantages of using compressed voxel representations.

Looking forward, one of the most promising directions for future improvement is the potential integration of RDM with Nanite, Unreal Engine's virtualized geometry system. Although Nanite does not currently support runtime procedural meshes, future updates could allow for automatic and efficient generation of level-of-detail data during runtime. This would reduce the need for extensive preprocessing and simplify the workload on the CPU in this regard. Quad merging would likely continue to play an important role in creating optimized complex colliders. If meshes generated by optimized RDM, especially those derived from RLE-compressed voxel data, could be converted into Nanite clusters at runtime, it could significantly reduce computational overhead while improving both scalability and visual fidelity. This capability would enable voxel engines to support high-resolution models with minimal preprocessing and allow for real-time interaction, deformation, and streaming of large voxel environments.

In summary, this thesis serves as both a prototype and a demonstration of a new approach to voxel meshing. It represents a foundational step toward building scalable, memory-efficient voxel systems. It is my hope that this work will inspire further development and innovation in this area.

Bibliography

[1] Akenine Möller, T.; Haines, E.; Hoffman, N.; Pescea, A.; Iwanickia, M. et al. *Real-time rendering* ebook. 4th Editionth ed. New York: A K Peters/CRC Press, september 2018. 1198 p. ISBN 978-1-1386-2700-0.

[2] Ampla Network. *Making a voxel engine* online. Ampla Network, 11. november 2020. Available at: https://dev.to/amplanetwork/making-a-voxel-engine-46h8. [cit. 2025-05-01].

[3] Arbore, R.; Liu, J.; Wefel, A.; Gao, S. and Shaffer, E. Hybrid Voxel Formats for Efficient Ray Tracing. In: Bebis, G.; Patel, V.; Gu, J.; Panetta, J.; Gingold, Y. et al., ed. *Advances in Visual Computing*. Cham: Springer Nature Switzerland, 2025, p. 125–138. ISBN 978-3-031-77392-1.

[4] Arnebäck, E.; Lundén, A.; Bärring, F.; Löfman, A.; Hage, J. et al. *Bloxel: Developing a Voxel Game Engine in Java using OpenGL*. Gothenburg, Sweden, 2015. Bachelor's thesis. Chalmers University of Technology and University of Gothenburg. Examiner: Arne Linde.

[5] Beyer, J.; Hadwiger, M. and Pfister, H. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Comput. Graph. Forum*. Chichester, GBR: The Eurographs Association & John Wiley & Sons, Ltd., december 2015, vol. 34, no. 8, p. 13–37. ISSN 0167-7055. Available at: https://doi.org/10.1111/cgf.12605.

[6] Bhatia, H.; Hoang, D.; Morrical, N.; Pascucci, V.; Bremer, P.-T. et al. AMM: Adaptive Multilinear Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2022, vol. 28, no. 6, p. 2350–2363.

[7] Blackflux. *Meshing in Voxel Engines – Part 1* online. Blackflux, 23. february 2014. Available at: https://blackflux.wordpress.com/2014/02/23/meshing-in-voxel-engines-part-1/. [cit. 2024-12-27].

[8] Blackflux. *Meshing in Voxel Engines – Part 2* online. Blackflux, 1. march 2014. Available at: https://blackflux.wordpress.com/2014/03/01/meshing-in-voxel-engines-part-2/. [cit. 2024-12-27].

[9] Blackflux. *Meshing in Voxel Engines – Part 3* online. Blackflux, 2. march 2014. Available at: https://blackflux.wordpress.com/2014/03/02/meshing-in-voxel-engines-part-3/. [cit. 2024-12-27].

[10] Botsch, M.; Kobbelt, L.; Pauly, M.; Alliez, P. and Levy, B. *Polygon Mesh Processing*. Taylor & Francis, 2010. Ak Peters Series. ISBN 9781568814261. Available at: https://books.google.cz/books?id=8zX-2VRqBAkC.

[11] Bruin, P. W. de; Vos, F. M.; Post, F. H.; Frisken Gibson, S. F. and Vossepoel, A. M. Improving Triangle Mesh Quality with SurfaceNets. In: Delp, S. L.; DiGoia, A. M. and Jaramaz, B., ed. *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 804–813. ISBN 978-3-540-40899-4.

[12] Cadence Design Systems. *What is Meshing? | Mesh Generation Overview*. Available at: https://www.cadence.com/en_US/home/explore/what-is-meshing.html.

[13] Cambronero, N. A. *DigBuild: Sandbox Voxel Game and Engine Using Vulkan*. 2021. Bachelor's thesis. Universidad de Salamanca. Supervisor González, D. G. V. and Santana, D. J. F. D. P.

[14] Cohen Or, D. and Kaufman, A. Fundamentals of Surface Voxelization. *Graphical Models and Image Processing*, 1995, vol. 57, no. 6, p. 453–461. ISSN 1077-3169. Available at: https://www.sciencedirect.com/science/article/pii/S1077316985710398.

[15] Crassin, C. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. 2011. Dissertation. Grenoble University. Available at: http://maverick.inria.fr/Membres/Cyril.Crassin/thesis/CCrassinThesis_EN_Web.pdf,Thesis.

[16] Denisova, E.; Manetti, L.; Bocchi, L. and Iadanza, E. Voxelization: Multi-target Optimization for Biomedical Volume Rendering. In: Badnjević, A. and Gurbeta Pokvić, L., ed. *MEDICON'23 and CMBEBIH'23*. Cham: Springer Nature Switzerland, 2024, p. 232–241.

[17] Doitrand, A.; Fagiano, C.; Irisarri, F.-X. and Hirsekorn, M. Comparison between voxel and consistent meso-scale models of woven composites. *Composites Part A: Applied Science and Manufacturing*, 2015, vol. 73, p. 143–154. ISSN 1359-835X. Available at: https://www.sciencedirect.com/science/article/pii/S1359835X15000743.

[18] Dusterwald, S. *Procedural Generation of Voxel Worlds with Castles*. Hamilton, New Zealand, 2015. Master of Science (MSc). University of Waikato. Available at: https://hdl.handle.net/10289/9819.

[19] Ebert, D. S.; Musgrave, F. K.; Peachey, D.; Perlin, K. and Worley, S. *Texturing and Modeling: A Procedural Approach*. 3rdth ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1558608486.

[20] Engel, K.; Hadwiger, M.; Kniss, J. M. and Rezk Salama, C. Real-Time Volume Graphics. In: Magnenat Thalmann, N. and Bühler, K., ed. *Eurographics 2006: Tutorials*. The Eurographics Association, 2006.

[21] Epic Games. *Unreal Engine 5.4 Documentation* online. Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-4-documentation?application_version=5.4. [cit. 2025-03-21].

[22] Epic Games. *Fab, Epic's new unified content marketplace, launches today!* 22. october 2024. Available at: https://www.unrealengine.com/en-US/blog/fab-epics-new-unified-content-marketplace-launches-today.

[23] Eppstein, D. Graph-Theoretic Solutions to Computational Geometry Problems. In: Paul, C. and Habib, M., ed. *Graph-Theoretic Concepts in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 1–16. ISBN 978-3-642-11409-0.

[24] Fang, G.; Chen, C.; Meng, S. and Liang, J. Mechanical analysis of three-dimensional braided composites by using realistic voxel-based model with local mesh refinement. *Journal of Composite Materials*, 2019, vol. 53, no. 4, p. 475–487.

[25] Faraj, N.; Thiery, J.-M. and Boubekeur, T. VoxMorph: 3-scale freeform deformation of large voxel grids. *Computers & Graphics*, 2012, vol. 36, no. 5, p. 562–568. ISSN 0097-8493. Available at: https://www.sciencedirect.com/science/article/pii/S0097849312000593. Shape Modeling International (SMI) Conference 2012.

[26] Fatahalian, K.; Boulos, S.; Hegarty, J.; Akeley, K.; Mark, W. R. et al. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery, july 2010, vol. 29, no. 4. ISSN 0730-0301. Available at: https://doi.org/10.1145/1778765.1778804.

[27] Forstmann, S. and Ohya, J. Visualization of Large RLE-Encoded Voxel Volumes. In: *FIT2007*. Tokyo, Japan: Waseda University, GITS, 2007, p. 207–208.

[28] Furht, B., ed. Mesh, Three Dimentional. In: Furht, B., ed. *Encyclopedia of Multimedia*. 2nd ed. Boston, MA: Springer US, 2008, p. 419–420. ISBN 978-0-387-78414-4. Available at: https://doi.org/10.1007/978-0-387-78414-4_109.

[29] Gibson, S. F. F. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In: Wells, W. M.; Colchester, A. and Delp, S., ed. *Medical Image Computing and Computer-Assisted Intervention — MICCAI'98*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, p. 888–898. ISBN 978-3-540-49563-5.

[30] Guan, B.; Lin, S.; Wang, R.; Zhou, F.; Luo, X. et al. Voxel-based quadrilateral mesh generation from point cloud. *Multimedia Tools and Applications*, Aug 2020, vol. 79, no. 29, p. 20561–20578. ISSN 1573-7721. Available at: https://doi.org/10.1007/s11042-020-08923-5.

[31] Guo, Y.-C.; Cao, Y.-P.; Wang, C.; He, Y.; Shan, Y. et al. VMesh: Hybrid Volume-Mesh Representation for Efficient View Synthesis. In: *SIGGRAPH Asia 2023 Conference Papers*. New York, NY, USA: Association for Computing Machinery, 2023. SA '23. ISBN 9798400703157. Available at: https://doi.org/10.1145/3610548.3618161.

[32] Hadwiger, M.; Al Awami, A. K.; Beyer, J.; Agus, M. and Pfister, H. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2018, vol. 24, no. 1, p. 974–983.

[33] He, T.; Hong, L.; Kaufman, A.; Varshney, A. and Wang, S. Voxel based object simplification. In: *Proceedings Visualization '95*. 1995, p. 296–303.

[34] Jensen, K. T. *25 Years Later: The History of Unreal and an Epic Dynasty* online. PCMag, 22. may 2023. Available at: https://www.pcmag.com/news/25-years-later-the-history-of-unreal-and-an-epic-dynasty. [cit. 2025-03-02].

[35] Ju, T.; Losasso, F.; Schaefer, S. and Warren, J. Dual contouring of hermite data. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery, july 2002, vol. 21, no. 3, p. 339–346. ISSN 0730-0301. Available at: https://doi.org/10.1145/566654.566586.

[36] Kahler, R.; Simon, M. and Hege, H.-C. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 2003, vol. 9, no. 3, p. 341–351.

[37] Kalshetti, P.; Rahangdale, P.; Jangra, D.; Bundele, M. and Chattopadhyay, C. Antara: An Interactive 3D Volume Rendering and Visualization Framework. *CoRR*, 2018, abs/1812.04233. Available at: http://arxiv.org/abs/1812.04233.

[38] Keen Software House. *Space Engineers*. Available at: https://store.steampowered.com/app/244850/Space_Engineers/. Figure taken from Steam page.

[39] Kim, H. and Swan, C. Voxel-based meshing and unit-cell analysis of textile composites. *International Journal for Numerical Methods in Engineering*, february 2003, vol. 56, p. 977 – 1006.

[40] Knittel, G. Verve. *Computer Graphics Forum*, 1993, vol. 12, no. 3, p. 37–48. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1230037.

[41] Knoll, A.; Thelen, S.; Wald, I.; Hansen, C. D.; Hagen, H. et al. Full-resolution interactive CPU volume rendering with coherent BVH traversal. In: *2011 IEEE Pacific Visualization Symposium*. 2011, p. 3–10.

[42] Krüger, M.; Gilbert, D.; Kuhlen, T. W. and Gerrits, T. Game Engines for Immersive Visualization: Using Unreal Engine Beyond Entertainment. *PRESENCE: Virtual and Augmented Reality*, july 2024, vol. 33, p. 31–55. ISSN 1054-7460. Available at: https://doi.org/10.1162/pres_a_00416.

[43] Laan, R. van der; Scandolo, L. and Eisemann, E. Lossy Geometry Compression for High Resolution Voxel Scenes. *Proc. ACM Comput. Graph. Interact. Tech.* New York, NY, USA: Association for Computing Machinery, may 2020, vol. 3, no. 1. Available at: https://doi.org/10.1145/3384541.

[44] Lagae, A.; Lefebvre, S.; Cook, R.; DeRose, T.; Drettakis, G. et al. A Survey of Procedural Noise Functions. *Computer Graphics Forum*, 2010, vol. 29, no. 8, p. 2579–2600. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01827.x.

[45] Laine, S. and Karras, T. Efficient sparse voxel octrees. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2010, p. 55–63. I3D '10. ISBN 9781605589398. Available at: https://doi.org/10.1145/1730804.1730814.

[46] Laine, S. and Karras, T. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation. In:. 2011. Available at: https://api.semanticscholar.org/CorpusID:16748383.

[47] Langer, S. A. and Reid, A. C. *Calculating Voxel-Polyhedron Intersections for Meshing Images*. Zenodo, october 2021. Available at: https://doi.org/10.5281/zenodo.5559225.

[48] Lengyel, E. *Mathematics for 3D Game Programming and Computer Graphics*. 3thth ed. Cengage Learning, 2012. ITPro collection. ISBN 9781435458871. Available at: https://books.google.cz/books?id=qtVvCgAAQBAJ.

[49] Lengyel, E. S. *Voxel-based terrain for real-time virtual simulations*. USA, 2010. Dissertation. University of California at Davis. ISBN 9781124025490. AAI3404919.

[50] Lin, J. *The Perfect Voxel Engine* online. voxely, 18. september 2021. Available at: https://voxely.net/blog/the-perfect-voxel-engine/. [cit. 2025-04-26].

[51] Lo, S. H. Finite element mesh generation and adaptive meshing. *Progress in Structural Engineering and Materials*, 2002, vol. 4, no. 4, p. 381–399. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/pse.135.

[52] Lorensen, W. E. and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1987, p. 163–169. SIGGRAPH '87. ISBN 0897912276. Available at: https://doi.org/10.1145/37401.37422.

[53] Lv, C.; Lin, W. and Zhao, B. Voxel Structure-Based Mesh Reconstruction From a 3D Point Cloud. *IEEE Transactions on Multimedia*, 2022, vol. 24, p. 1815–1829.

[54] Lysenko, M. *An Analysis of Minecraft-like Engines* online. 0FPS blog, 14. january 2012. Available at: https://0fps.net/2012/01/14/an-analysis-of-minecraft-like-engines/. [cit. 2024-11-05].

[55] Lysenko, M. *Meshing in a Minecraft Game* online. 0FPS blog, 30. june 2012. Available at: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. [cit. 2024-11-05].

[56] Lysenko, M. *Meshing in a Minecraft Game (Part 2)* online. 0FPS blog, 7. july 2012. Available at: https://0fps.net/2012/07/07/meshing-minecraft-part-2/. [cit. 2024-11-18].

[57] Lysenko, M. *Smooth Voxel Terrain (Part 1)* online. 0FPS blog, 10. july 2012. Available at: https://0fps.net/2012/07/10/smooth-voxel-terrain-part-1/. [cit. 2025-4-27].

[58] Lysenko, M. *Smooth Voxel Terrain (Part 2)* online. 0FPS blog, 12. july 2012. Available at: https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/. [cit. 2025-4-27].

[59] Mattoo, S. *I Tried Best Game Engines of 2025: Top 7 Picks* online. G2, 29. january 2025. Available at: https://learn.g2.com/best-game-engine. [cit. 2025-03-21].

[60] Max Slater. *Making a voxel engine* online. Max Slater, 26. august 2018. Available at: https://thenumb.at/Voxel-Meshing-in-Exile/#. [cit. 2025-05-01].

[61] McDonald, N. *High Performance Voxel Engine: Vertex Pooling* online. Nick's Blog, 4. april 2021. Available at: https://nickmcd.me/2021/04/04/high-performance-voxel-engine/. [cit. 2025-04-27].

[62] McDougall, J. *Minecraft review* online. PCGamer, 25. december 2011. Available at: https://www.pcgamer.com/minecraft-review/. [cit. 2025-03-22].

[63] Mileff, P. and Dudra, J. Simplified Voxel Based Visualization. *Production Systems and Information Engineering*, january 2019, vol. 8, p. 5–18.

[64] Miller, M.; Cumming, A.; Chalmers, K.; Kenwright, B. and Mitchell, K. Poxels: polygonal voxel environment rendering. In: *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–236. VRST '14. ISBN 9781450332538. Available at: https://doi.org/10.1145/2671015.2671125.

[65] Molpeceres, V. H. *FNG Documentation* online. Available at: https://drive.google.com/file/d/1jv-DOR46-gL2tK7pstMnQL6x6LWa7z2d/view. [cit. 2025-04-22].

[66] Morton, G. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966. Available at: https://books.google.cz/books?id=9FFdHAAACAAJ.

[67] Neural Concept. *What Is Meshing: Unlocking the Power of 3D Geometry*. Available at: https://www.neuralconcept.com/post/what-is-meshing-unlocking-the-power-of-3d-geometry.

[68] O'Conor, K. *GPU Performance for Game Artists*. 3. march 2017. Available at: http://www.fragmentbuffer.com/gpu-performance-for-game-artists/.

[69] Peck, J. *FastNoiseLite*. Available at: https://github.com/Auburn/FastNoiseLite/tree/master. Noise generator.

[70] Richermoz, A. and Neyret, F. GigaVoxels DP: Starvation-Less Render and Production for Large and Detailed Volumetric Worlds Walkthrough. *Proc. ACM Comput. Graph. Interact. Tech.* New York, NY, USA: Association for Computing Machinery, august 2024, vol. 7, no. 3. Available at: https://doi.org/10.1145/3675389.

[71] Robinson, M. *Voxel World Optimisations*. 29. july 2019. Available at: https://vercidium.com/blog/voxel-world-optimisations/.

[72] Robinson, M. *Further Voxel World Optimisations*. 11. january 2020. Available at: https://vercidium.com/blog/further-voxel-world-optimisations/.

[73] Rodrigues, L. S.; Riehm, F.; Zachow, S. and Israel, J. H. VoxSculpt: An Open-Source Voxel Library for Tomographic Volume Sculpting in Virtual Reality. In: *2023 9th International Conference on Virtual Reality (ICVR)*. 2023, p. 515–523.

[74] Schlüter, M.; Kwasnitschka, T.; Bernstetter, A. and Karstens, J. *Rendering Large Volume Datasets in Unreal Engine 5: A Survey*. 2025. Available at: https://arxiv.org/abs/2504.07485.

[75] Schütz, M.; Herzberger, L. and Wimmer, M. SimLOD: Simultaneous LOD Generation and Rendering for Point Clouds. *Proc. ACM Comput. Graph. Interact. Tech.* New York, NY, USA: Association for Computing Machinery, may 2024, vol. 7, no. 1. Available at: https://doi.org/10.1145/3651287.

[76] Short, T. and Adams, T. *Procedural Generation in Game Design*. 1st ed. New York: CRC Press, 2017. ISBN 9781315156378.

[77] Spacefarer. *Voxel Meshing for the Rest of us* online. Spacefarer, 16. february 2023. Available at: https://playspacefarer.com/voxel-meshing/. [cit. 2025-04-27].

[78] Spatial Corporation. *Meshing Definition | What is Mesh Generation* https://www.spatial.com/glossary/what-is-meshing. 2025. [cit. 2025-4-15].

[79] Steinbrenner, J. P.; Wyman, N. J.; Jefferies, M. S.; Karman, S. L. and Shipman, J. Implementation of a Size Field Based Isotropic Hex Core Meshing Algorithm. In: *AIAA Scitech 2020 Forum*. AIAA, 2020, 2020-1408.

[80] System Era Softworks. *ASTRONEER*. Available at: https://store.steampowered.com/app/361420/ASTRONEER/. Figure taken from Steam page.

[81] Tokarev, K. *How Voxels Became 'The Next Big Thing'* online. 80Level, 27. may 2018. Available at: https://medium.com/@EightyLevel/how-voxels-became-the-next-big-thing-4eb9665cd13a. [cit. 2025-04-26].

[82] TriAxis Games. *Realtime Mesh Docs* online. 2023. Available at: https://rmc.triaxis.games/. [cit. 2025-04-22].

[83] Turner, E. and Zakhor, A. Watertight Planar Surface Meshing of Indoor Point-Clouds with Voxel Carving. In: *2013 International Conference on 3D Vision - 3DV 2013*. 2013, p. 41–48.

[84] Tustvold. *Voxel Engine – Data Storage* online. Subterranean Software, 8. august 2014. Available at: https://www.subterraneansoftware.com/run-based-voxel-engine/. [cit. 2025-05-01].

[85] Tustvold. *Voxel Engine – Mesh Generation* online. Subterranean Software, 11. august 2014. Available at: https://www.subterraneansoftware.com/voxel-engine-mesh-generation/. [cit. 2025-05-01].

[86] Tuxedo Labs. *Teardown*. Available at: https://store.steampowered.com/app/1167630/Teardown/. Figure taken from Steam page.

[87] Vigo, M.; Pla, N.; Ayala, D. and Martínez, J. Efficient algorithms for boundary extraction of 2D and 3D orthogonal pseudomanifolds. *Graphical Models*, 2012, vol. 74, no. 3, p. 61–74. ISSN 1524-0703. Available at: https://www.sciencedirect.com/science/article/pii/S1524070312000112.

[88] Voxel Plugin. *Voxel Plugin Documentation* online. Available at: https://docs.voxelplugin.com/. [cit. 2025-05-7].

[89] Wang, Z.; Teo, J.; Chui, C.; Ong, S.; Yan, C. et al. Computational biomechanical modelling of the lumbar spine using marching-cubes surface smoothened finite element voxel meshing. *Computer Methods and Programs in Biomedicine*, 2005, vol. 80, no. 1, p. 25–35. ISSN 0169-2607. Available at: https://www.sciencedirect.com/science/article/pii/S0169260705001239.

[90] Wickramasinghe, U.; Remelli, E.; Knott, G. and Fua, P. Voxel2Mesh: 3D Mesh Model Generation from Volumetric Data. In: *Medical Image Computing and Computer Assisted Intervention – MICCAI 2020: 23rd International Conference, Lima, Peru, October 4–8, 2020, Proceedings, Part IV*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 299–308. ISBN 978-3-030-59718-4. Available at: https://doi.org/10.1007/978-3-030-59719-1_30.

[91] Wilder, M. W. *An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain*. 2015. Available at: https://ideaexchange.uakron.edu/honors_research_projects/217. Williams Honors College, Honors Research Projects.

[92] Wu, Q.; Bauer, D.; Doyle, M. J. and Ma, K.-L. Interactive Volume Visualization via Multi-Resolution Hash Encoding Based Neural Representation. *IEEE Transactions on Visualization and Computer Graphics*, 2024, vol. 30, no. 8, p. 5404–5418.

[93] Zadick, J.; Kenwright, B. and Mitchell, K. Integrating Real-Time Fluid Simulation with a Voxel Engine:. *The Computer Games Journal*, september 2016, vol. 5.

[94] Zhang, C.; Zhou, Y. and Zhang, L. *Voxel-Mesh Hybrid Representation for Real-Time View Synthesis*. 2024. Available at: https://arxiv.org/abs/2403.06505.

[95] Zhang, H. *Mesh Generation for Voxel-Based Objects*. 2005. Dissertation. West Virginia University. Available at: https://doi.org/10.33915/etd.2676.