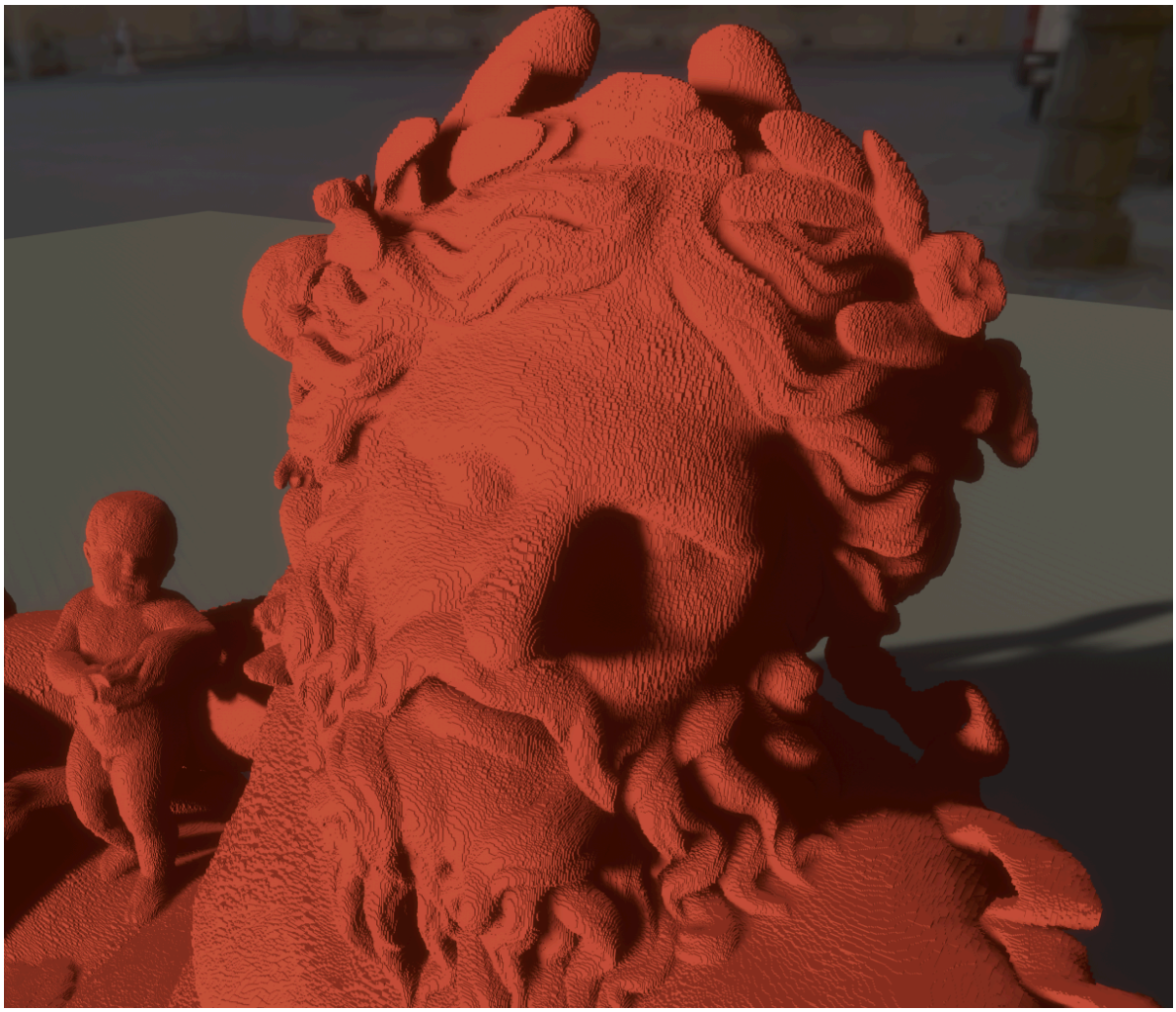
The Voxelis Bible

From Pixels to Worlds - An In-Depth Guide



*Need voxels? Reach for **Voxelis**. Tiny voxels, huge worlds, zero hassle.*
*Powered by **VoxTree** – a deliciously crafted SVO DAG with batching. Drop it into your Rust, C++, Godot, or Bevy project and start carving worlds down to centimetre-level detail – while your memory bill stays shockingly low.*

Artur Wyszyński · WILD PIXEL GAMES

Version 2.3 (Reflects Svo removal, VoxInterner renaming, LOD implementation & benchmarks, latest sources & benchmark format, hierarchy plans, raytracing potential, unified Vox* naming, added BlockId section, front page, styling, bestiary)

Project created by Artur Wyszyński — Wild Pixel Games

Document authored by Artur Wyszyński <artur.wyszynski@wildpixelgames.com>

Foreword

Welcome to the technical deep dive into Voxelis, a high-performance voxel engine core written in Rust, specifically designed for high performance and memory efficiency when dealing with potentially vast worlds composed of extremely small voxels. Unlike traditional voxel engines where a voxel might represent a cubic meter (like Minecraft), Voxelis operates at a much finer granularity (e.g., 4cm voxels), necessitating advanced techniques for storage and manipulation.

This document serves as a comprehensive guide to the architecture, design principles, underlying theory, and evolution of the Voxelis core. It aims to be a central reference for developers working with or onboarding onto the project, providing insights into "how" things work and "why" certain decisions were made. The Voxelis project is licensed under the permissive MIT OR Apache-2.0 licenses.

We will embark on a bottom-up exploration, starting from fundamental concepts and data structures, moving through the core storage system, the octree implementations, and finally touching upon higher-level structures and utilities like serialization and mesh voxelization.

Foreword 2
Part 1: Theoretical Foundations 5
1.1 What are Voxels? 5
1.2 History of Voxel Engines (Brief Overview) 5
1.3 Applications of Voxels 6
1.4 Advantages and Disadvantages vs. Other Representations 6
Part 2: Data Structures for Voxel Representation 7
2.1 3D Arrays (Flat Arrays) 7
2.2 Octrees 7
2.3 Sparse Voxel Octrees (SVO) 7
2.4 Directed Acyclic Graphs (DAG) 7
2.5 Perfect Spatial Hashing / Other Approaches 8
2.6 Comparative Analysis 8
2.7 The VoxTree Implementation 9
Part 3: Memory Optimization Techniques in Voxelis 10
3.1 BlockId: The 64-bit Node Identifier 10
3.2 Hash Consing (Interning) 11
3.3 Reference Counting 12
3.4 Generational Indices 12
3.5 Structure of Arrays (SoA) vs Array of Structures (AoS) 13
3.6 Data Compression (for Serialization) 13
3.7 Copy-on-Write (CoW) 14
3.8 Evolution and the Power of Batching 14
3.9 Scalability and Resolution Impact (Case Studies: Mars & Motunui) 15
Part 4: Modification and Query Operations 17
4.1 Single Voxel Operations (get/set) 17
4.2 Batch Operations (apply_batch) 17
4.3 Raycasting (Not Implemented in Core) 19
4.4 Regional Operations (fill, clear) 19
4.5 Concurrency and Thread Safety 19
Part 5: Voxel Rendering (Meshing) 21
5.1 Direct Rendering (Not Implemented) 21
5.2 Conversion to Meshes (Meshing) 21
5.3 Level of Detail (LOD) 22
Part 6: Voxelization 24
6.1 Mesh-to-Voxel Conversion 24
Part 7: World Management 25
Part 8: Serialization and I/O 27
Part 9: Implementation Language (Rust) 28

Part 10: Practical Implementation Examples (In Codebase) 29
Part 11: Problems, Challenges, and Solutions in Voxelis 30
Part 12: Case Studies and Competitive Context 31
Part 13: Future and Trends 32
Part 14: About Voxelis 34
Part 15: Bibliography and Resources (Generic) 35
Part 16: Appendices 36
16.1 ~~Glossary~~ The Voxelis Bestiary 36
16.2 Code Snippets 39
16.3 Benchmark Highlights 39

Part 1: Theoretical Foundations

Before diving into the specific implementation, let's establish the key theoretical concepts underpinning Voxelis.

1.1 What are Voxels?

Think of a 2D image. It's made of tiny squares called **pixels** (picture elements). Each pixel has a position (X, Y) and a value (e.g., color).
Now, imagine extending this into 3D. Instead of squares, we have tiny cubes called **voxels** (volume elements). Each voxel has a position (X, Y, Z) in a 3D grid and a value. This value can represent various things:

- **Material:** Is it air, rock, water, wood?
- **Color:** What color should be displayed at this point?
- **Density:** How dense is the material here (useful for simulations)?
- **Signed Distance:** How far is this point from the nearest surface (used in Signed Distance Fields - SDFs)?

Voxels allow us to represent the **volume** of objects and environments, not just their surfaces, which is the primary approach in traditional polygon mesh rendering.

1.2 History of Voxel Engines (Brief Overview)

Voxel technology isn't new. Early uses included medical imaging (CT/MRI scans are essentially voxel data) and some early games (like Comanche). These often used simpler storage methods and rendering techniques (like raycasting dense grids). Despite the long-standing dominance of polygon-based graphics, optimized for modern GPUs, voxel technology persists and experiences periodic renaissances, particularly in game development.
The challenge has always been performance and memory usage, especially for dynamic, large-scale worlds common in modern games. Engines like Minecraft popularized a block-based **aesthetic** (though it primarily uses voxels for terrain data storage and renders with polygons), while other games like Medieval Engineers or Moonglow Bay showcase different applications. The continued development of voxel engines and plugins, despite hardware favoring polygons, suggests that the inherent benefits of voxels – precise volumetric representation, ease of modification for destruction or procedural generation – outweigh the rendering performance challenges for certain game genres and applications. Modern approaches, like those used in Voxelis, build upon foundational concepts like Sparse Voxel Octrees (SVOs) and Directed Acyclic Graphs (DAGs) to tackle the historical memory and performance limitations, aiming to make high-detail, large-scale voxel worlds practical.

1.3 Applications of Voxels

- **Games:** World representation (Minecraft, No Man's Sky), destructible environments, terrain generation, specific visual styles.
- **Simulations:** Fluid dynamics, stress analysis, material simulation where volume matters.
- **Medical Imaging:** CT, MRI scans directly produce voxel data for visualization and analysis.
- **CAD/Manufacturing:** Representing complex internal structures of designed objects.
- **Scientific Visualization:** Representing volumetric datasets (e.g., atmospheric data, geological surveys).

1.4 Advantages and Disadvantages vs. Other Representations

Vs. Polygon Meshes:

- **Pros:** Naturally represent volume, easy volumetric modifications (destruction, carving), simpler collision detection within the grid, potentially easier procedural generation of complex volumes.
- **Cons:** Can require significantly more memory (especially naive implementations), rendering can be more complex (raycasting or complex meshing algorithms needed, GPU hardware less optimized), achieving smooth surfaces requires high resolution or advanced techniques.

Vs. Point Clouds:

- **Pros:** Implicitly represent connectivity and volume between points, easier to determine "inside" vs "outside".
- **Cons:** Generally require more storage than point clouds for the same visual fidelity of a *surface*, grid structure can be restrictive.

Voxelis Context: Voxelis aims to mitigate the memory cost disadvantage through advanced data structures (SVO DAGs) while leveraging the volumetric advantages for potentially highly detailed and dynamic environments.

Part 2: Data Structures for Voxel Representation

How do we actually store voxel data efficiently?

2.1 3D Arrays (Flat Arrays)

- **Implementation:** A simple 3D array data[x][y][z].
- **Pros:** Extremely fast random access (O(1) lookup if you know the coordinates). Simple to implement.
- **Cons: Massive memory consumption.** Requires storing every single voxel, even empty ones. Unfeasible for large or high-resolution worlds. A 1024x1024x1024 grid of single-byte voxels would require 1 GB of memory, scaling cubically.
- **Voxelis Context:** Not used directly for large-scale storage due to memory inefficiency.

2.2 Octrees

- **Implementation:** Tree structure where nodes represent cubic volumes, subdividing into 8 children for smaller volumes.
- **Pros:** Naturally represents space hierarchically. Good for spatial queries. Can represent empty space implicitly (no node needed).
- **Cons:** Accessing a specific voxel requires tree traversal (logarithmic complexity, O(log N) where N is the dimension). Can still be memory-intensive if not optimized for sparseness.
- **Voxelis Context:** Forms the fundamental basis of the storage system.

2.3 Sparse Voxel Octrees (SVO)

- **Implementation:** An octree where nodes are only created where data exists or varies. Empty space doesn't require nodes at lower levels. Branch nodes explicitly track which children exist.
- **Pros:** Significantly reduces memory compared to dense octrees or flat arrays for typical sparse worlds (lots of empty space or large uniform areas).
- **Cons:** Tree traversal overhead remains. Modifications can be complex.
- **Voxelis Context:** The core concept used. The mask field in BlockId directly supports sparseness by tracking child existence.

2.4 Directed Acyclic Graphs (DAG)

- **Implementation:** An SVO where identical subtrees (nodes with the same voxel value or identical children configurations) are stored only *once* and shared via pointers/references. Multiple parent nodes can point to the same child node.
- **Pros: Massive memory savings** by eliminating redundant data. If large parts of the

world are identical (e.g., repeating ground material, large air pockets), they are stored only once.

- **Cons:** Modifications become much more complex due to shared ownership (necessitates techniques like CoW). Requires efficient detection of identical nodes (hash consing). Garbage collection (like reference counting) is needed to manage shared node lifetimes. Historically considered slower for modifications.
- **Voxelis Context:** This is the core principle behind the **VoxTree** implementation and the **VoxInterner**. Hash consing via the patterns maps enables the DAG structure. The performance drawback is addressed via batching (see Section 3.8).

2.5 Perfect Spatial Hashing / Other Approaches

- **Perfect Spatial Hashing:** Maps 3D coordinates directly to memory locations using a hash function designed to have no collisions for the specific dataset. Can be very fast for static data but difficult to update. Less common in dynamic game engines.
- **VDB (and OpenVDB):** A highly influential hierarchical data structure widely used in VFX. Uses a B+-tree-like structure with wider branching factors and often stores data in small dense blocks within the hierarchy. Very efficient for sparse volumetric data and certain operations common in simulations and rendering, offering dynamic topology. While powerful, it may not achieve the same degree of structural sharing as a pure SVO DAG for highly repetitive game world content.
- **Voxelis Context:** Voxelis uses a classic SVO DAG approach, optimized for game-centric operations like batch updates and fine-grained detail, rather than adopting VDB or Perfect Hashing directly.

2.6 Comparative Analysis

Structure	Access Time	Memory (Dense)	Memory (Sparse)	Modification Complexity	Sharing	Voxelis Use
Flat Array	O(1)	Very High	Very High	Simple	No	No
Dense Octree	O(logN)	High	High	Moderate	No	No
SVO	O(logN)	High	Moderate	Moderate	No	Historical Attempt
SVO DAG	**O(logN)**	**High**	**Very Low**	**High (CoW)**	**Yes**	**Core (VoxTree)**
VDB	O(logN)ish	High	Low-Moderate	Moderate	Some	No
Perfect Hash	O(1)	Moderate	Moderate	Very High (Static)	No	No

Voxelis chooses the **SVO DAG** approach, implemented as **VoxTree**, prioritizing **extreme memory efficiency** for its target of very small voxels, while mitigating the high modification complexity through **batching**. This choice acknowledges the inherent complexity of modifying DAGs but strategically addresses it through an **optimized batching mechanism**, aiming to achieve a superior balance of memory usage and update performance compared to other structures in its target domain. The existence of mature alternatives like OpenVDB highlights the different trade-offs available, with Voxelis focusing specifically on DAG-based optimization for potentially highly repetitive game worlds.

2.7 The VoxTree Implementation

The core data storage structure in Voxelis is named **VoxTree**. It is Voxelis's highly optimized implementation of a Sparse Voxel Octree Directed Acyclic Graph (SVO DAG). While based on the SVO DAG concept, **VoxTree** incorporates several key enhancements and features that distinguish it:

- **Aggressive Hash-Consing:** Utilizes the **VoxInterner** (Section 3.1) to ensure maximal deduplication of both leaf and branch nodes, significantly reducing memory footprint for repetitive structures.
- **Batched Copy-on-Write:** Implements an optimized batch update mechanism (apply_batch, Section 4.2) that amortizes the cost of CoW modifications, achieving massive speedups (demonstrated up to ~250x faster than single set operations in benchmarks, see Section 3.8).
- **Zero-Cost Level of Detail (LOD):** Integrates LOD data directly into the main DAG structure by repurposing an existing field in branch nodes via the SoA layout, providing always-up-to-date LOD information with no extra memory or runtime rendering/meshing cost (see Section 5.3).

Essentially, **VoxTree** is the name given to the specific, high-performance SVO DAG implementation at the heart of the Voxelis engine.

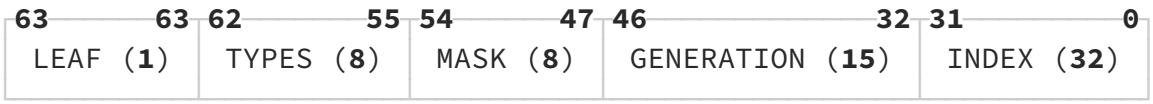
Part 3: Memory Optimization Techniques in Voxelis

Voxelis employs several techniques synergistically to achieve its memory goals.

3.1 BlockId: The 64-bit Node Identifier

At the heart of the **VoxInterner**'s memory management and node identification lies the BlockId struct. It's a compact 64-bit unsigned integer meticulously designed to pack crucial information about each node (leaf or branch) in the **VoxTree**. Understanding its structure is key to understanding how Voxelis achieves its efficiency.

Bit Layout:



- **Bit 63: Node Type Flag (LEAF)**
  - **1**: Indicates a **Leaf Node**. This node directly represents voxel data (e.g., material ID, color). It's a terminal node in the octree path.
  - **0**: Indicates a **Branch Node**. This is an internal node that points to up to 8 children, further subdividing the space.
- **Bits 62-55: Child Types (TYPES)**
  - *Only relevant for Branch Nodes.*
  - Each bit corresponds to one of the 8 potential children (indexed 0 to 7 according to Morton order).
  - **1**: The child at this position is a Leaf Node.
  - **0**: The child at this position is a Branch Node.
  - This field helps optimize traversals and operations by quickly identifying the type of children without needing to dereference them immediately.
- **Bits 54-47: Child Presence Mask (MASK)**
  - *Only relevant for Branch Nodes.*
  - Each bit corresponds to one of the 8 potential children.
  - **1**: A child node exists at this position.
  - **0**: No child exists (representing empty or uniform space covered by a higher-level node).
  - This mask is fundamental to the "Sparse" nature of the SVO, allowing the tree to skip storing nodes for empty regions.
- **Bits 46-32: Generation (GENERATION)**
  - A 15-bit counter used for **Generational Indexing** (see Section 3.4).
  - When a node's memory slot is recycled by the allocator, its generation number is incremented.

    - A BlockId is considered valid only if its embedded generation matches the current generation stored for its index in the **VoxInterner**. This prevents the "ABA problem" where an old BlockId might mistakenly reference a new node allocated at the same recycled index.
    - Maximum value is 0x7FFE (32,766).
- **Bits 31-0: Index (INDEX)**
    - A 32-bit index pointing to the actual node data (like children pointers or voxel value) stored in the **VoxInterner**'s Structure of Arrays (SoA) pools.
    - This allows addressing up to ~4 billion unique node slots within the interner.

Special Values:

- **BlockId::EMPTY** (0): Represents an empty node or space.
- **BlockId::INVALID** (u64::MAX): Represents an invalid or uninitialized ID.

Benefits: This packed representation allows Voxelis to store essential metadata about each node directly within its 64-bit identifier, minimizing memory overhead and enabling efficient checks and operations during tree traversal, modification, and memory management.

3.2 Hash Consing (Interning)

- **Concept:** As described in 2.4, ensuring only one copy of each unique node exists.
- **Voxelis Implementation (VoxInterner):**
    - Uses two PatternsHashmaps (Rust FxHashMap with a custom IdentityHasher) stored in the patterns array field: one for leaves (PATTERNS_TYPE_LEAF), one for branches (PATTERNS_TYPE_BRANCH).
    - **Hashing:**
        - Leaves: Hashed based on their voxel value T using compute_leaf_hash_for_value (which uses FxHasher).
        - Branches: Hashed based on their types, mask, and the BlockIds (specifically, their raw u64 values) of all 8 potential children using compute_branch_hash_for_children.
    - **Lookup/Creation (get_or_create_leaf, get_or_create_branch):**
        - Compute the hash for the node to be created/found.
        - Look up the hash in the appropriate patterns map.
        - **Cache Hit:** If found and the existing BlockId is valid (checked via generation), increment the existing node's reference count and return the existing BlockId. (For branches, also decrement ref counts of input children, as they won't be needed for a *new* node).
        - **Cache Miss:** If not found, allocate a new node slot (get_next_index_macro!), store the node's data (value or children) and its hash in the **VoxInterner**'s SoA pools, insert the hash -> new_BlockId mapping into patterns, set the ref count to 1, and return the new_BlockId.
- **Benefit:** This is the primary mechanism enabling the memory-efficient DAG structure

used by **VoxTree**.

3.3 Reference Counting

- **Concept:** Tracking how many references point to each shared node to know when it can be deleted.
- **Voxelis Implementation (VoxInterner):**
  - ref_counts: PoolAllocatorLite<u32>: Stores the count for each node index.
  - inc_ref(id) / inc_ref_by(id, count): Increment the count for the node's index.
  - dec_ref(id) / dec_ref_by(id, count): Decrement the count. If it reaches zero:
    - Remove the node's entry from the appropriate patterns map (using its stored hash).
    - Call recycle(id).
  - dec_ref_recursive(id): Handles the recursive decrement for branch nodes when they are freed. Uses dec_ref_rec_stack (a preallocated Vec<BlockId>) to manage the traversal iteratively, preventing call stack overflows. It intelligently decrements child counts directly if > 1 or pushes them onto the stack if == 1.
  - recycle(id): Resets the node's data slots, increments the generation counter for that slot, and adds the index to the free_indices list.
- **Benefit:** Provides deterministic garbage collection essential for managing the shared nodes in the DAG.

3.4 Generational Indices

- **Concept:** Solving the ABA problem in reference counting / recycled allocators. An old reference (BlockId) should not be able to accidentally access a *new* node allocated at the same recycled memory index.
- **Voxelis Implementation:**
  - generations: PoolAllocatorLite<u16>: Stores a generation counter for each allocatable index in the **VoxInterner**.
  - BlockId: Embeds a 15-bit generation field.
  - VoxInterner::recycle(id): When an index id.index() is freed, its corresponding counter in the generations pool is incremented (wrapping around BlockId::MAX_GENERATION).
  - VoxInterner::get_or_create_*: When allocating a node at index, it reads the *current* generation g from generations.get(index) and embeds it into the newly created BlockId(..., generation: g, index: index).
  - VoxInterner::is_valid_block_id(id): Checks if id.generation() == *self.generations.get(id.index()). An old BlockId will have a stale generation number and fail this check, preventing use-after-free or accessing the wrong data.
- **Benefit:** Ensures the validity of BlockIds even when memory indices are recycled.

3.5 Structure of Arrays (SoA) vs Array of Structures (AoS)

- **Concept:**
  - **AoS:** Store all data for one object together in a struct, then have an array of these structs. E.g., struct Node { children: Children, value: T, ref_count: u32, ... }; Vec<Node>.
  - **SoA:** Store each *field* of the object in its own separate array. E.g., Vec<Children>, Vec<T>, Vec<u32>, all indexed by the same logical node index.
- **Voxelis Implementation (VoxInterner):** Uses SoA via multiple PoolAllocatorLite instances (children, values, ref_counts, generations, hashes).
- **Why SoA?**
  - **Cache Locality:** When an algorithm processes the *same field* for many different nodes sequentially (e.g., dec_ref_recursive iterating and modifying ref_counts, or traversals accessing children), SoA keeps that specific data contiguous in memory. This leads to fewer cache misses and better performance compared to AoS, where unrelated node data would be pulled into the cache line alongside the needed field.
  - **Flexibility:** Can potentially have different memory management strategies or alignment requirements for different fields (though Voxelis uses PoolAllocatorLite for all here).
  - **Enables Zero-Cost LOD:** As described later (Section 5.3), the SoA layout allows storing aggregated LOD values for branch nodes in the values array (which is otherwise unused for branches) without any extra structural overhead.
- **Trade-off:** Accessing *all* fields for a *single* node might be slightly slower with SoA (multiple memory accesses) compared to AoS (one contiguous block), but the operations that benefit from SoA (like GC, traversal, LOD storage) are often more performance-critical in this context.

3.6 Data Compression (for Serialization)

- **Concept:** Reducing the size of data when saving to disk or transmitting over a network.
- **Voxelis Implementation (io::export::export_model_to_vtm, io::import::import_model_from_vtm):**
  - Uses the zstd compression library via the zstd crate.
  - The Flags::COMPRESSED bitflag in the VTM header indicates whether the main data block is compressed.
  - During export, if the flag is set, the serialized **VoxInterner**/chunk data is compressed using zstd::stream::Encoder (level 7) before writing size and data.
  - During import, if the flag is set, the data block is decompressed using zstd::stream::Decoder before parsing.
- **Benefit:** Significantly reduces file size, especially for large models, at the cost of some CPU time during saving/loading.

3.7 Copy-on-Write (CoW)

- **Concept:** As described in 1.4, modifying immutable structures by creating copies of affected nodes along the path.
- **Voxelis Implementation (VoxTree::set, set_at_depth_iterative):**
  - set operations do not modify existing nodes in the **VoxInterner** directly.
  - They use get_or_create_leaf and get_or_create_branch to build a *new* path from the root.
  - get_or_create_* ensures that if an identical node already exists (due to hash consing), it's reused, minimizing the actual "copying". Only truly *new* node configurations result in allocations.
  - The **VoxTree** struct updates its root_id to point to the new path's root and decrements the old root's ref count correctly.
- **Benefit:** Enables modifications on the immutable, shared DAG structure while preserving structural sharing and ensuring existing references remain valid.

3.8 Evolution and the Power of Batching

A crucial part of Voxelis's design journey involved addressing the performance bottleneck of Copy-on-Write (CoW) modifications inherent in the SVO DAG structure (**VoxTree**).
Initially, the pure CoW approach in **VoxTree**, while memory-efficient, proved slow for individual set operations, especially in dynamic scenarios. Each set required potentially costly path copying and hash consing lookups/insertions (get_or_create_*). An attempt was made to create a faster "Dynamic" variant (Svo) that likely aimed for in-place updates or CoW without branch deduplication, but it suffered from implementation issues and was ultimately removed.
The Breakthrough: The implementation of **batching** specifically for **VoxTree** dramatically changed the landscape. By grouping multiple modifications into a Batch object and applying them using the optimized VoxTree::apply_batch algorithm (which processes changes bottom-up, amortizing traversal and hash consing costs), the write performance of **VoxTree** improved drastically, often by orders of magnitude.
Benchmark Evidence: The performance difference is starkly illustrated by comparing single set operations (single) versus applying a pre-calculated batch (batch) in the unified benchmarks for **VoxTree** (from benches.md, testing on an Apple M3 Max):

Operation (Pattern, Depth 5 / 32^3)	Single Op Time	Batch Op Time	Speedup Factor
Uniform Fill	~5.17 ms	~23.1 µs	**~224x**
Terrain Surface Only	~181 µs	~10.9 µs	**~16.6x**
Sum-based Fill (High Entropy)	~5.92 ms	~194 µs	**~30.5x**

Analysis & Conclusion:

- Batching provides **massive speedups** across all tested modification patterns compared to individual set calls on the **VoxTree**.
- The efficiency gains are particularly dramatic for dense updates (set_uniform) but remain significant even for sparse (set_terrain) or high-entropy (set_sum) data.
- This demonstrates that the **batch application algorithm itself is highly efficient** at reconstructing the necessary parts of the DAG, minimizing redundant work even when hash consing benefits are lower.
- Implementing optimized batching rendered the separate Svo implementation attempt redundant. Batched **VoxTree** provides both the extreme memory efficiency of the DAG **and** high-performance modifications, making it the superior, unified solution. This justifies the complexity of the apply_batch algorithm.

3.9 Scalability and Resolution Impact (Case Studies: Mars & Motunui)

While synthetic benchmarks are useful for isolating specific operations, testing the engine on large, complex, real-world data provides invaluable insights into its practical scalability and the impact of resolution choices. Two significant tests involved voxelizing large mesh datasets: a detailed Mars terrain model and the Disney Motunui island dataset.

Test Setups:

- **Mars:** ~25M triangle mesh, 3.2km x 3.2km x 800m area. Tested at Depth 6 (4cm voxels) and Depth 5 (8cm voxels). Chunk size: 2.56m.
- **Motunui:** ~0.5M triangle mesh, ~9km x 3km x 9km bounding box. Tested at Depth 5 (32cm voxels). Chunk size: 10.24m.
- **Process:** The Voxelizer tool was used with triangle-cube intersection and parallel processing across chunks.

Key Comparative Results:

Metric	Mars @ Depth 6 (4cm)	Mars @ Depth 5 (8cm)	Motunui @ Depth 5 (32cm)
Input Tris (Millions)	~25	~25	~0.5
Chunks Processed (Millions)	~3.1	~3.1	~3.0
Voxelization Time	~33 min	~4.5 min	~22 min
Unique Nodes (Millions)	~147.3	~49.5	~17.7
VTM Size (Approx)	~2.5 GB	~816 MB	~285 MB
Cache Hit Rate	~99.5% (~200:1)	~99.1% (~113:1)	~99.8% (~500:1)

Analysis and Implications:

- **Proven Scalability & Memory Efficiency:** The engine successfully processed both massive datasets. The number of unique nodes stored is consistently orders of magnitude smaller than the potential voxel count, confirming the **extreme effectiveness of the SVO DAG hash consing** on complex, real-world geometry. Final file sizes are manageable.
- **Computational Cost of Resolution:** Voxelization time is highly sensitive to resolution. Halving the resolution for Mars (Depth 6 -> 5) gave a >7x speedup. The Motunui test, despite fewer input triangles and fewer final nodes than Mars@5, took longer, potentially due to different geometric characteristics (e.g., larger triangles requiring more voxel intersection checks per triangle).
- **Non-Linear Complexity Scaling:** Reducing resolution doesn't reduce node count proportionally to volume, highlighting the SVO DAG's ability to capture detail efficiently. The significantly lower node count for Motunui compared to Mars@5 (despite similar chunk counts) suggests Motunui has less fine geometric detail or larger uniform areas at that 32cm resolution.
- **HashMap Performance is Critical at Scale:** Generating **tens to hundreds of millions** of unique branch nodes confirms that the performance of the VoxInterner::patterns HashMap is crucial at this scale. Accessing and inserting into a single map holding ~20-150 million entries likely incurs significant CPU cache penalties. This strongly reinforces the need to investigate **sharding the branch_patterns HashMap** (e.g., hash % N) as a key future optimization for large-scale performance.
- **Cache Hit Rate Varies with Data:** The extremely high hit rate for Motunui (~500:1) suggests massive structural repetition (perhaps water, sky, internal island mass), showcasing the DAG's strength in such scenarios. The still high but lower rates for Mars (~200:1, ~113:1) reflect its more varied terrain geometry.

In conclusion, these case studies provide compelling practical evidence of Voxelis's strengths in handling large, detailed datasets with remarkable memory efficiency. They also clearly identify the performance characteristics related to resolution and dataset complexity, pinpointing the central hash map access as a primary target for future scalability optimizations.

Part 4: Modification and Query Operations

How users interact with the voxel data.

4.1 Single Voxel Operations (get/set)

- **get(interner, position):** (VoxOpsRead trait)
  - Implemented by **VoxChunk** and **VoxTree**, ultimately calling the utility get_at_depth.
  - Performs a standard octree traversal from the root, using child_index_macro_2 at each level to choose the next child based on the position.
  - Returns Some(T) if it reaches a leaf node containing the position, None otherwise (empty space).
  - **Complexity:** O(D) where D is the depth of the target voxel (or max depth if empty). Performance shown in voxtree_get_* benchmarks.
- **set(interner, position, voxel):** (VoxOpsWrite trait)
  - Implemented by **VoxChunk** and **VoxTree**.
  - Implements CoW + Hash Consing via set_at_depth_iterative (or remove_at_depth if voxel == T::default()).
  - Descends to find the target location, then reconstructs the path upwards using get_or_create_leaf and get_or_create_branch.
  - Updates the **VoxTree**'s root_id and correctly decrements the old root's ref count.
  - **Complexity:** O(D) traversals + O(D) hash computations/lookups/allocations in the worst case. Performance is heavily impacted by CoW overhead and is significantly slower than batch operations, as shown in benchmarks (e.g., voxtree_set_uniform/32/single vs voxtree_batch_set_uniform/32). **Should be avoided for modifying multiple voxels.**

4.2 Batch Operations (apply_batch)

- **apply_batch(interner, batch):** (VoxOpsBatch trait)
  - Implemented by **VoxTree**. Calls set_batch_at_depth_iterative.
  - **Algorithm:** A highly optimized bottom-up reconstruction:
    1. **Handle Fill:** Checks batch.to_fill() first. If set, returns a single get_or_create_leaf node, effectively replacing the entire tree.
    2. **Process Leaf Level (Depth max_depth-1):** Iterates through relevant entries in batch.masks (where changes exist). For each corresponding parent node position (derived via Morton code path_index), it determines the original node state at that position (by traversing down the *original* tree). It then constructs the *new* children array for this node by:
       - Copying existing children from the original node (or using the uniform leaf_node_id if the original was a leaf).

        - Overwriting specific children by calling interner.get_or_create_leaf for values specified in batch.values.
        - Calculates the new types and mask.
        - **Optimization:** If all 8 resulting children are identical leaves, use the leaf's BlockId directly, collapsing the branch.
        - Otherwise, call interner.get_or_create_branch to get the canonical BlockId for this new branch configuration.
        - Stores the resulting BlockId in a temporary buffer (current_level_data) indexed by the Morton path_index. Records the processed path index.
    3. **Integrate Upwards (Levels max_depth-2 to 0):** Iteratively reconstructs the tree level by level.
        - Uses the paths list (containing Morton codes of modified nodes at the *current* level being processed) and the precomputed PATH_MASKS to efficiently group sibling nodes that share the same parent at the *next level up*.
        - For each group of siblings:
            - Retrieves the children BlockIds (calculated in the previous step) from current_level_data.
            - Determines the original parent node state at this level (by traversing the original tree).
            - Copies any *unmodified* children from the original parent node (incrementing their ref counts temporarily).
            - Constructs the final children array, types, and mask for the new parent node.
            - **Optimization:** Checks if all children are identical leaves; if so, uses the leaf ID directly (after decrementing temporary ref counts), collapsing the branch.
            - Otherwise, calls interner.get_or_create_branch to get the canonical parent BlockId (this handles ref count adjustments correctly based on cache hit/miss).
            - Stores the resulting parent BlockId in the *next* level's buffer (next_level_data), indexed by the parent's Morton code prefix. Adds the parent's path prefix to next_paths.
        - After processing all paths for a level, swaps the current/next buffers and path lists and moves up (target_depth -= 1).
    4. **Final Root:** The BlockId remaining in the buffer after processing level 0 is the new root ID for the entire tree.
- **Efficiency:** This bottom-up, level-by-level approach with sibling grouping and hash consing (get_or_create_branch) dramatically reduces redundant work compared to individual set calls. **This is the preferred method for all multi-voxel modifications in Voxelis.**
- **Complexity:** More complex implementation, but amortized cost per modification

is much lower than individual set calls, especially for dense or localized changes. Benchmark results show its superior performance across various data patterns.

4.3 Raycasting (Not Implemented in Core)

- **Concept:** Finding the first voxel intersected by a ray cast into the scene. Essential for interaction, rendering, visibility checks.
- **Algorithms:** Various octree traversal algorithms exist (e.g., Amanatides & Woo).
- **Voxelis Context:** The core library (voxelis crate reviewed) does *not* seem to include a raycasting implementation. This would likely be built on top, using the get or lower-level **VoxInterner** access methods. A potential future direction is GPU-based raycasting (see Section 13).

4.4 Regional Operations (fill, clear)

- **fill(interner, value):** (VoxOpsWrite trait)
  - Optimized implementation. Decrements the old root's ref count. Uses interner.get_or_create_leaf(value) to get the ID for a single leaf node representing the entire filled volume. Sets this as the new root_id. Very fast (nanosecond range, see voxtree_fill benchmark).
- **clear(interner):** (VoxOpsWrite trait)
  - Effectively fill(interner, T::default()). Decrements the old root's ref count and sets root_id to BlockId::EMPTY. Very fast (picosecond range for already empty, nanosecond/microsecond range depending on previous content complexity, see voxtree_clear_* benchmarks).

4.5 Concurrency and Thread Safety

- **Challenge:** Multiple threads might need to read or write to the voxel world simultaneously.
- **Voxelis Solution:**
  - The **VoxInterner** itself is *not* internally thread-safe (due to unsafe in PoolAllocatorLite and mutable access to HashMaps/Vecs).
  - Thread safety is achieved at the **VoxModel** level by wrapping the **VoxInterner** in Arc<RwLock<VoxInterner<i32>>>.
  - Arc: Allows shared ownership of the **VoxInterner** across threads.
  - RwLock (from parking_lot): Allows multiple concurrent readers (interner.read()) or one exclusive writer (interner.write()).
  - Operations requiring only read access to the store (like get, is_empty, to_vec, generate_mesh_arrays) can acquire a read lock and run concurrently.
  - Operations modifying the store (like set, fill, clear, apply_batch, which call get_or_create_* or dec_ref*) must acquire a write lock, ensuring exclusive access

during mutation.

- **Chunk-Level Parallelism:** Operations like VoxModel::serialize and Voxelizer::voxelize_mesh use rayon to process *chunks* in parallel, but each chunk operation still needs to acquire the appropriate lock (read or write) on the *shared* **VoxInterner** when accessing/modifying it.

Part 5: Voxel Rendering (Meshing)

Voxelis focuses on storage and modification. To visualize the data, it needs to be converted into something renderable, typically a triangle mesh.

5.1 Direct Rendering (Not Implemented)

- **Techniques:** Ray Marching or Ray Tracing directly operate on the voxel data (often SVOs) without creating an intermediate mesh. Can produce high-quality results but can be computationally expensive. A potential future direction for Voxelis is discussed in Section 13.
- **Voxelis Context:** Not implemented in the core library.

5.2 Conversion to Meshes (Meshing)

- **Goal:** Generate a triangle mesh representing the *surface* of the voxel data.
- **Voxelis Implementation (VoxChunk::generate_mesh_arrays):**
  - Implements a **Naive Cubic / Blocky Meshing** algorithm.
  - **Process:**
    1. Flattens the octree into a dense 3D array using octree.to_vec(lod). (Potential bottleneck, especially at high LOD, as shown in voxtree_to_vec_* benchmarks). The lod parameter allows generating the mesh at different resolutions.
    2. Iterates through every voxel in the (potentially LOD-reduced) array.
    3. For each solid voxel, checks its 6 neighbors within the same LOD level.
    4. If a neighbor is empty/outside the chunk, generate a quad (2 triangles) for the face between the solid voxel and the empty space.
    5. Uses SIMD (f32x8) to optimize vertex position calculations.
  - **Characteristics:** Simple to implement, produces blocky meshes suitable for some art styles. Performance shown in voxtree_naive_mesh_* benchmarks.
- **DAG Benefit - Mesh Deduplication:** Because the SVO DAG (**VoxTree**) ensures identical subtrees share the same BlockId, an application using Voxelis (like the vtm-viewer) can implement mesh deduplication. It can generate the mesh for a specific BlockId (at a given LOD) only once and then reuse that same mesh instance for all chunks or parts of chunks that reference the same BlockId. This was observed to yield >96% mesh reduction in a test scene with repeating structures ("kominy").
- **Other Common Algorithms (Not Implemented in Voxelis Core):**
  - **Marching Cubes:** Classic algorithm, produces smoother surfaces by considering voxel values at cube corners to determine surface intersection. Can suffer from ambiguous cases.
  - **Dual Contouring / Surface Nets:** More advanced algorithms aiming for sharper features and better topology, often using Hermite data (surface normals) or point

samples.
    - **Greedy Meshing:** Optimizes the cubic meshing approach by merging adjacent, coplanar faces into larger rectangles, significantly reducing the final polygon count. Often complex to implement correctly.
- **Voxelis Context:** The current meshing is basic but functional and supports LOD. Implementing a more advanced algorithm like Greedy Meshing is a planned future optimization. Market analysis suggests a trend towards higher visual quality, making advanced meshing techniques increasingly relevant.

5.3 Level of Detail (LOD)

- **Concept:** Using lower-resolution representations (fewer polygons or larger voxels) for distant objects to improve rendering performance. Traditional approaches often involve generating and storing separate LOD trees or simplified meshes, incurring memory and maintenance overhead.
- **Voxelis Implementation: Zero-Cost, Always-Up-to-Date LOD:** Voxelis implements a highly efficient, integrated LOD system directly within the main **VoxTree** structure, leveraging the SoA layout of the **VoxInterner**. This addresses a common challenge in voxel engines where LOD management can be complex and costly.
    - **Storage:** The values: PoolAllocatorLite<T> pool in **VoxInterner**, which normally only stores voxel data for *leaf* nodes (T), is repurposed to store an *aggregated value* (also of type T) for *branch* nodes. This field was previously unused and effectively wasted for branches.
    - **Aggregation:** When a branch node is created or found via VoxInterner::get_or_create_branch, it calculates an "average" value of its 8 children. This calculation is performed by the VoxelTrait::average(children: &[T]) -> T method, which must be implemented for the voxel type T. The default implementation provided for numeric types performs a **mode calculation** (finds the most frequent voxel value among the children, useful for material IDs). Voxel types representing colors or other data could override this to implement different aggregation strategies (e.g., numerical average, mipmap-like filtering). The computed average value is then stored in the values slot corresponding to the branch node's index.
    - **Zero-Cost:** This approach adds **no extra memory overhead** (reuses an existing, previously unused field in the SoA layout) and **no new data structures** (no separate mipmap chains or LOD trees). LOD information is inherently part of the main octree DAG.
    - **Trivial Extraction:** To get the voxel data at a specific LOD level (Lod), simply traverse the octree down to the desired depth (lod.level()) and read the value stored at that node (using interner.get_value(node_id)), regardless of whether it's a leaf or a branch. The branch node's stored value represents the aggregated appearance of its entire subtree at that coarser level. The VoxChunk::generate_mesh_arrays and VoxOpsMesh::to_vec methods accept an

Lod parameter to generate data at the specified level using this mechanism.
    - **Always Up-to-Date:** Because the aggregated value is computed during node creation/retrieval (as part of the hash consing process for branches in get_or_create_branch), the LOD information is always consistent with the current state of the tree after any modification (set or apply_batch). No separate LOD update passes or mipmap generation steps are needed.
- **Performance Impact:**
    - **Non-batched set:** Introduces a small overhead, as the average value needs to be recomputed for nodes along the path being created via CoW. (Observed ~4ms -> ~5ms in octree_set_sum benchmark).
    - **Batched apply_batch:** No measurable performance impact. The cost of aggregation is amortized during the bottom-up reconstruction and is negligible compared to other batch processing costs.
    - **Reading/Meshing:** Zero overhead. Accessing the LOD value is just reading a field from the **VoxInterner**. Benchmarks (voxtree_to_vec_*, voxtree_naive_mesh_*) clearly show that operations at higher LOD levels (lower effective depth) are significantly faster, confirming the efficiency of accessing coarser data. For example, voxtree_to_vec_sphere/64 drops from ~1ms at LOD 0 to ~15µs at LOD 2 and ~30ns at LOD 5.
- **Benefit:** Voxelis provides a first-class, highly efficient, always-consistent LOD system with virtually no runtime or memory cost, making it easy and cheap to implement LOD rendering or processing techniques. This is a significant advantage over systems requiring explicit LOD generation and storage.

Part 6: Voxelization

Converting other representations (like polygon meshes) into voxels.

6.1 Mesh-to-Voxel Conversion

- **Voxelis Implementation (Voxelizer):**
  - **Input:** io::Obj structure (parsed from OBJ file).
  - **Preprocessing (build_face_to_chunk_map):** Determines which mesh faces might intersect which chunks to avoid checking all faces against all chunks. Crucial optimization.
  - **Core Logic (voxelize_mesh):**
    1. Iterates through chunks (potentially in parallel using rayon).
    2. For each chunk, iterates through potentially intersecting faces.
    3. Calculates the overlapping AABB between the face and chunk.
    4. Determines the range of local voxel indices (local_min_voxel, local_max_voxel) corresponding to the overlap.
    5. Iterates through every voxel within this local range.
    6. Performs a precise **Triangle-Cube Intersection Test** (triangle_cube_intersection) for each voxel against the face.
    7. If they intersect, adds the voxel modification (setting to 1) to a per-chunk Batch.
    8. After processing all faces for a chunk, applies the Batch using chunk.apply_batch.
  - **Accuracy vs. Performance:** Uses a precise geometric test, which is accurate but can be computationally intensive. The spatial partitioning and parallel processing help performance. Using apply_batch per chunk further optimizes the writes to the **VoxInterner**.
  - **Alternative (simple_voxelize):** Faster but less accurate method based only on voxelizing mesh vertices.
- **Other Techniques (Not Used):** Rasterization-based approaches (rendering the mesh into a 3D grid), Signed Distance Field generation.

Part 7: World Management

Managing potentially vast worlds requires hierarchical structures beyond the single octree within a chunk. Voxelis plans a multi-level hierarchy to handle scale, LOD, and streaming efficiently.

- **Level 0: VoxTree & VoxInterner:** The core data structure (**VoxTree** being the SvoDag implementation) representing the voxels within a fixed cubic region, leveraging the shared, deduplicated node storage provided by **VoxInterner**. This is the fundamental building block.
- **Level 1: VoxChunk:** A direct wrapper around a **VoxTree** instance. It adds a 3D integer position (position) within a larger grid (measured in chunk units) and knows its physical size in world units (chunk_size). It forwards VoxOps* calls to the underlying **VoxTree**. This remains the primary unit of voxel data manipulation.
- **Level 2: VoxColumn (Conceptual/Planned):** Represents a vertical stack of **VoxChunks** at a specific XZ coordinate. Internally, it would likely use a sparse structure (e.g., HashMap<i32, VoxChunk> mapping Y-coordinate to **VoxChunk**) to store only the chunks that actually contain data, avoiding memory allocation for empty vertical space deep underground or high in the sky. This enables efficient handling of worlds with significant vertical variation or emptiness.
- **Level 3: VoxSector (Conceptual/Planned):** Represents a 2D area (e.g., 32x32 **VoxColumns**) on the XZ plane. This acts as a larger management unit. It's planned as an enum with two variants:
    - **VoxSector::Full**: Contains the actual grid (e.g., HashMap<IVec2, VoxColumn>) of **VoxColumns**, holding the full-resolution data for that area. Used for regions close to the camera/player.
    - **VoxSector::Proxy**: Represents the same XZ area but uses a *single, scaled* ***VoxColumn*** containing stacked and scaled ***VoxChunks*** with a lower max_depth. For example, if a Full sector uses chunks with max_depth=5 (32^3 voxels of size 8cm each, covering 40.96m x 40.96m), a Proxy sector could use a single chunk with max_depth=0 (1 voxel of size 40.96m), max_depth=1 (2^3 voxels of size 20.48m), or max_depth=2 (4^3 voxels of size 10.24m) etc., to represent the same area with drastically reduced detail and memory. The aggregated value for the proxy chunk's voxels would be derived from the engine's built-in LOD system (Section 5.3). This Proxy acts as a **spatial Level of Detail** mechanism for distant areas. **VoxSectors** could transition between Full and Proxy states based on distance or visibility.
- **Level 4: VoxRegion (Conceptual/Planned):** A potential higher level grouping (e.g., 16x16 **VoxSectors**). This would likely be the primary unit for **streaming data to and from disk (Out-of-Core support)**.
    - **Streaming Challenge:** Saving/loading parts of a shared DAG is complex. Simply saving the BlockIds within a **VoxRegion** isn't enough, as those IDs are only valid within the current **VoxInterner** instance.

- **Proposed Solution:** When saving a **VoxRegion**, it would embed a *subset* of the global **VoxInterner** containing only the nodes (leaves and branches) actually referenced within that **VoxRegion** (and their descendants). This requires traversing the DAGs within the sector to identify necessary nodes. An ID mapping (similar to the full VoxModel::serialize) would be needed to store these nodes with file-local IDs. When loading a **VoxRegion**, its embedded node data would be treated like a blueprint, using get_or_create_* operations to integrate these nodes back into the *global, live* **VoxInterner**, automatically handling deduplication if identical nodes already exist from other loaded sectors.

This hierarchical approach (**VoxChunk** -> **VoxColumn** -> **VoxSector** (Full/Proxy) -> **VoxRegion**) provides a framework for managing enormous worlds by combining spatial partitioning, sparse storage, spatial LOD (VoxSector::Proxy), and a mechanism for streaming (**VoxRegion** with embedded node data).

Part 8: Serialization and I/O

- **VTM Format:** Custom binary format designed for Voxelis - VoxTreeModel.
  - **Header:** Magic, version, flags (compression), dimensions, metadata (chunk size, world bounds, name), MD5 checksum.
  - **Data:** Serialized **VoxInterner** DAG + chunk data (position, root reference).
  - **Features:** VarInt encoding, zstd compression, MD5 integrity check.
- **Serialization Process (VoxModel::serialize):**
  1. Collect unique nodes from **VoxInterner**.
  2. Build index -> file_id map.
  3. Write unique leaves (mapped ID, value).
  4. Write unique branches (mapped ID, mask, mapped child IDs, *aggregated LOD value*).
  5. Write number of chunks.
  6. Serialize each chunk in parallel (position, mapped root ID).
  7. Combine node and chunk data, calculate MD5, compress (optional), write header and combined data.
- **Deserialization Process (VoxModel::deserialize):**
  1. Read header, verify magic/version, decompress data (if needed), verify MD5.
  2. Pass 1: Read leaves, call interner.deserialize_leaf, build file_id -> BlockId map for leaves.
  3. Pass 1: Read branches, call interner.preallocate_branch_id, store file_id -> (BlockId, file_child_ids, lod_value) map for branches.
  4. Pass 2: Link branches using maps, call interner.deserialize_branch (passing the lod_value).
  5. Read number of chunks.
  6. Deserialize each chunk, using maps to link to the correct root BlockId, and insert into the **VoxModel**'s chunks HashMap.
- **OBJ I/O:** Basic OBJ parsing for input, basic OBJ export for generated meshes (can export specific LOD levels).

Part 9: Implementation Language (Rust)

- **Choice:** Rust.
- **Benefits Leveraged:**
  - **Performance:** Zero-cost abstractions, control over memory layout (SoA), potential for C-like speed.
  - **Memory Safety:** Borrow checker prevents common errors like use-after-free, data races, especially crucial given the complexity of shared structures (DAG) and unsafe blocks in allocators.
  - **Concurrency:** Fearless concurrency enabled by the type system and libraries (Arc, RwLock, rayon).
  - **Ecosystem:** Crates like criterion, glam, parking_lot, rayon, zstd, byteorder, bitflags, rustc_hash provide powerful building blocks.
- **Challenges:** Steeper learning curve, managing lifetimes and borrowing can be complex, unsafe still requires careful handling.

Part 10: Practical Implementation Examples (In Codebase)

- **SVO DAG (VoxTree):** VoxTree struct + **VoxInterner** + set_at_depth_iterative using get_or_create_*.
- **Hash Consing:** VoxInterner::patterns + get_or_create_* methods.
- **RC + Generations:** VoxInterner::ref_counts/generations + BlockId + recycle + is_valid_block_id.
- **Batching:** Batch struct + VoxTree::apply_batch + set_batch_at_depth_iterative.
- **SoA:** Multiple PoolAllocatorLite instances in **VoxInterner**.
- **Zero-Cost LOD:** VoxInterner::values used for branches + VoxelTrait::average + get_or_create_branch calculating average.
- **Voxelization:** Voxelizer struct.
- **Serialization:** VoxModel::serialize/deserialize.

Part 11: Problems, Challenges, and Solutions in Voxelis

- **Challenge:** Memory use with small voxels. **Solution:** SVO DAG (**VoxTree**) with Hash Consing.
- **Challenge:** Slow CoW modifications. **Solution:** Optimized Batching (apply_batch).
- **Challenge:** Managing shared node lifetimes safely. **Solution:** RC + Generational Indices.
- **Challenge:** Avoiding GC pauses. **Solution:** Deterministic RC.
- **Challenge:** Avoiding stack overflows during GC. **Solution:** Non-recursive dec_ref_recursive with explicit stack.
- **Challenge:** Achieving good cache locality. **Solution:** SoA layout in **VoxInterner**.
- **Challenge:** Implementing LOD efficiently. **Solution:** Zero-cost LOD via value aggregation in branch nodes using SoA.
- **Challenge:** Persistent storage of DAGs. **Solution:** VTM format with ID mapping.
- **Challenge:** Concurrent access to shared store. **Solution:** Arc<RwLock<VoxInterner>>.
- **Challenge:** Accurate mesh voxelization. **Solution:** Triangle-Cube intersection tests + spatial partitioning.
- **Challenge:** Efficient mesh generation. **Solution:** Basic cubic meshing (potential bottleneck identified). Mesh deduplication possible at application level.
- **Challenge:** HashMap performance at extreme scale (10s-100s millions of nodes). **Solution:** Investigating sharding (hash % N) as a future optimization.
- **Challenge:** Implementation complexity. **Solution:** Modularity (traits, separate structs), Rust's safety features.
- **Challenge:** Usability and Documentation. **Solution:** Ongoing effort to provide comprehensive documentation (like this document) and clear APIs.

Part 12: Case Studies and Competitive Context

- **Minecraft-like:** Voxelis could represent chunks, but it's fine granularity and DAG structure (**VoxTree**) are overkill for simple blocks. However, the *batching* could speed up large modifications. Compared to engines like *Luanti (Minetest)*, Voxelis focuses more on the core DAG technology rather than extensive modding APIs (currently).
- **High-Detail Destruction/Terrain:** The fine granularity and efficient storage make Voxelis well-suited for simulating detailed destruction or terrain deformation. Batching handles numerous modifications efficiently. Compared to the *UE Voxel Plugin*, Voxelis's core might be leaner, but the UE plugin offers a vastly richer feature set (Voxel Graphs, procedural foliage, etc.) integrated directly into Unreal Engine. *Zylann's Voxel Tools* for Godot offers similar capabilities (smooth/blocky terrain, LOD, editing) but requires engine compilation (vs. Voxelis being a library).
- **Medical/Scientific:** The ability to store detailed volumetric data efficiently could be applicable, although specific rendering/analysis tools would be needed. Compared to *OpenVDB*, Voxelis uses a different core structure (Octree DAG vs. B+-tree-like grid hierarchy) optimized for potentially different access patterns.
- **Asset Store Solutions (Unity):** Compared to assets like Voxelica or Voxeland, Voxelis aims to be a more fundamental, potentially higher-performance core library rather than a ready-to-use Unity asset with editor integration.

Voxelis positions itself as a high-performance, memory-efficient core library centered around the **VoxTree** (SVO DAG) technology, featuring optimized batching and a unique zero-cost LOD system. Its strength lies in enabling extreme detail and scale, particularly where structural repetition can be leveraged by the DAG. It currently lacks the extensive tooling, high-level features (advanced procedural generation, specific rendering modes), or direct editor integration found in some larger plugins or dedicated platforms.

Part 13: Future and Trends

This section outlines ongoing work, planned features, and potential future directions for Voxelis, informed by current technological trends and the engine's architecture.

Current Development Focus:

- **Improved World Management:** Implementing the hierarchical structure described in Section 7 (**VoxColumns**, **VoxSectors** with Full/Proxy variants).
- **Asynchronous Loading/Unloading & Paging:** Implementing mechanisms to load and unload **VoxRegions** asynchronously from disk, including the logic for integrating their embedded **VoxInterner** data with the global store.
- **FFI (Foreign Function Interface):** Creating C/C++ bindings for the core library.
- **Engine Integration:** Building integration layers for Godot and Bevy.
- **Greedy Meshing:** Implementing Greedy Meshing as a more optimized alternative for blocky mesh generation.

Potential Future Directions:

- **GPU-Accelerated Raytracing:** A highly promising direction leveraging the existing architecture.
    - **Concept:** Copy the core **VoxInterner** SoA buffers (children containing BlockIds and values containing leaf/LOD data) to GPU Shader Storage Buffer Objects (SSBOs). Write a compute shader to perform octree traversal per-ray directly on the GPU.
    - **"Almost Free" LOD:** The shader can terminate traversal at any depth and read the pre-calculated aggregated LOD value directly from the values buffer, enabling efficient, view-dependent LOD rendering without mesh generation. This bypasses the need for traditional meshing entirely for rendering.
    - **Challenges:** Managing potentially huge buffers in VRAM, designing efficient GPU traversal algorithms (handling pointers/indices, branching divergence), synchronizing data between CPU writes and GPU reads.
    - **Potential:** Could offer extremely high-quality rendering without the overhead and limitations of polygon meshing, serving as a high-end alternative or replacement for meshing, especially powerful when combined with the built-in LOD.
- **HashMap Optimization (Sharding):** As identified in the large-scale tests (Section 3.9), investigating techniques like sharding the VoxInterner::patterns map (hash % N) to improve cache performance and potentially reduce lock contention when dealing with tens/hundreds of millions of unique nodes.
- **Advanced Meshing Algorithms:** Implementing Dual Contouring or Surface Nets for smoother, higher-fidelity surface generation, potentially as an alternative rendering path alongside raytracing or cubic/greedy meshing.
- **Advanced Procedural Generation:** Integrating more sophisticated procedural generation techniques beyond simple noise functions.

- **Physics Integration:** Deeper integration with physics engines for more complex destruction simulations or voxel-based physics interactions.
- **Modding/Scripting API:** Adding a scripting layer (like Lua, as in Luanti) could significantly increase usability and community engagement for game development.
- **Advanced Voxel Types:** Supporting more complex voxel data (e.g., SDFs, multiple data fields per voxel) could broaden applications.
- **Multiplayer Support:** Designing robust state synchronization for the DAG structure presents a significant challenge but is a potential niche.

Part 14: About Voxelis

Voxelis is a next-generation voxel engine core, designed and developed by **Artur Wyszyński**, an experienced engine and tools programmer who contributed to projects at **CD Projekt RED**, **Techland**, and **QLOC**.

Created under **Wild Pixel Games**, Voxelis focuses on ultra-dense voxel detail, large-scale worlds, and extreme memory efficiency, empowering developers to push the boundaries of voxel simulation across **Rust**, **C++**, **Godot**, and **Bevy**.

The project is licensed under the **MIT OR Apache-2.0** license.

Part 15: Bibliography and Resources (Generic)

- **Key Publications:** Search for papers on Sparse Voxel Octrees, Octree DAGs, Hash Consing, VDB, Mesh Voxelization, Octree Raycasting (e.g., SIGGRAPH proceedings).
- **Online Resources:** Blogs (e.g., 0fps.net), tutorials (e.g., Job Talle's articles), open-source projects (OpenVDB, various voxel engines on GitHub).
- **Books:** "Real-Time Rendering," "Game Engine Architecture."

Part 16: Appendices

16.1 ~~Glossary~~ The Voxelis Bestiary

> *"In the forgotten archives of lost civilizations, the ancients whispered of entities that shaped the very bones of worlds.*
> *Few dared to study their nature.*
> *Fewer still lived to write of them.*
>
> This is their bestiary."
>
> ♦

DAG — The Eternal Weaver

A mythical creature that spins endless webs without ever looping back. DAGs are the backbone of old voxel empires, their acyclic nature ensuring no path crosses itself. Those who master the Weaver wield worlds compressed into mere whispers of memory.

CoW — The Shard Lord

A cunning entity that refuses to alter itself directly. Instead, it forges copies with subtle differences, preserving its pure form through endless reflections. It rules over mutation without sacrifice.

Hash Consing — The Mirror Warden

Dwelling deep within the Interner crypts, the Mirror Warden ensures that every identical soul (subtree) is recognized and merged into one. None may duplicate their form in its domain.

LOD — The Distant Eye

A vigilant sentinel that shapes what is seen based on distance. The closer you approach, the finer the detail; from afar, only blurry giants stir in the mist.

Morton Code — The Twisted Mapper

A sorcerer who scrambles paths into perfect, cache-hungry orders. Where once lay chaos, it carves perfect sequences through the twisting dimensions.

RC — The Silent Tallyman

Invisible but omnipresent, this being counts every allegiance to a node. Betray it — let the count fall to zero — and the memory dies silently, without fanfare.

SVO — The Collapser

A giant who feasts on emptiness and sameness, folding vast fields of space into elegant, sparse relics. Empty lands fear its inexorable hunger.

SoA — The Shardsplitter

Dividing attributes among countless rows and lines, the Shardsplitter seeks speed over unity, ensuring the fastest warriors (SIMD) may strike without pause.

BlockId — The Rune of Binding

A talisman carved from 64 shards, binding the nature, age, type, and position of a node into a single powerful glyph. Without it, the world cannot recall its own shape.

VoxChunk — The Worldseed

A tiny, dense shard of the endless world. Within its cubic prison, universes are born — or broken.

VoxColumn — The Spine of the Earth

Rising tall from the world's bones, Columns gather Chunks along the vertical axis, bridging sky and abyss.

VoxInterner — The Cryptkeeper

Master of deduplication, guardian of sameness. It hoards every unique node, weaving endless DAGs from the fewest possible stones.

VoxModel — The Shapeshifter

A construct of VoxTrees, unbound by terrain. It drifts, transforms, and reshapes the world with imported wonders.

VoxOps — The Silent Artisans

A secretive guild who know how to carve, reshape, and rebuild the very DAGs of existence. Through batchcraft and merging, they alter reality faster than the eye can follow.

VoxRegion — The Overrealm

A grand dominion containing countless Sectors, orchestrating the pulse of the streaming world across horizons unseen.

VoxSector — The Shifting Plains
A vast land, sometimes full, sometimes proxy — dancing between detailed memory and distant blur, balancing cost and clarity.

VoxTree — The Primal Grove
The living heart of voxel structure — a singular tree, branching through dimensions, growing compact but limitless.

VoxWorld — The Primal Grove
Master of all, stitching Regions into a living, breathing cosmos. Without it, the world falls silent and unformed.

"Those who master the beasts of structure shall weave worlds beyond mortal dreams."

* This is a glossary. It just has more teeth and fewer apologies.

16.2 Code Snippets

(Selected illustrative code snippets can be added here, e.g., BlockId layout, core get_or_create_ logic. For now, refer to the project repository.)

16.3 Benchmark Highlights

The following tables summarize key performance results obtained using the criterion benchmarking suite (voxtree_bench) on an **Apple M3 Max CPU (128 GB RAM, macOS Sequoia 15.4.1, Rust 1.86.0, -C target-cpu=native, final profile)**. Full results are available in the benches.md and benches-tables.md files in the repository.

Batch vs Single Operation Speedup (VoxTree, 32^3 Octree - Depth 5)			
Operation (Pattern)	**Single Op Time**	**Batch Op Time**	**Speedup Factor**
Uniform Fill	~5.17 ms	~23.1 µs	~224x
Terrain Surface Only	~181 µs	~10.9 µs	~16.6x
Sum-based Fill (High Entropy)	~5.92 ms	~194 µs	~30.5x
Hollow Cube	~980 µs	~49.3 µs	~19.9x
Sphere	~2.99 ms	~42.9 µs	~69.7x

Conclusion: Batching provides significant to massive speedups for all multi-voxel modification patterns compared to individual set calls.

LOD Performance Scaling - to_vec Operation (Sphere Pattern, 64^3 Octree - Depth 6):

LOD Level	Effective Depth	Time
0	6 (64^3)	~1.08 ms
1	5 (32^3)	~133 µs
2	4 (16^3)	~15.8 µs
3	3 (8^3)	~2.03 µs
4	2 (4^3)	~203 ns
5	1 (2^3)	~30.8 ns

Conclusion: Extracting data at lower LOD levels is significantly faster, demonstrating the effectiveness of the zero-cost LOD implementation.