School of Computation, Information and Technology – Informatics
Technical University of Munich

Master's Thesis in Informatics

Enhancing Performance and Scalability of the Space Foundation System through Chunking and Parallelization

Jonathan Bucher

School of Computation, Information and Technology – Informatics
Technical University of Munich

Master's Thesis in Informatics

Enhancing Performance and Scalability of the Space Foundation System through Chunking and Parallelization

Verbesserung der Performance und Skalierbarkeit des Space-Foundation-Systems durch Chunking und Parallelisierung

Author:	Jonathan Bucher
Supervisor:	Prof. Dr. rer. nat. John Doe
Advisors:	M.Sc. Daniel Dyrda
Submission Date:	September 10, 2025

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, September 10, 2025

JONATHAN BUCHER

Abstract

This thesis improves the scalability of the Space Foundation System (SFS), a designer-focused game-world abstraction that represents environments as graph of semantically meaningful locations. The reference SFS implementation employs a frontier-based, multi-seed, obstacle-aware voxel flood-fill. It is effective for medium-sized problems but struggles to scale efficiently on larger or higher-resolution scenes. We introduce a chunked parallel algorithm that partitions the voxel domain into independent chunks, resolves chunk borders and cutoff artifacts deterministically, and preserves the semantic correctness of the reference method. The proposed Unity implementation is designed for performance, employing the C# Job System, Burst Compiler, compact per-voxel layouts, and pragmatic serialization. It demonstrates substantial generation-time reductions for large scenes and a fundamental improvement in runtime query scaling: query cost becomes effectively constant when relevant chunks reside in memory, rather than increasing linearly with total voxel volume diameter. This shift enables arbitrarily large relative speedups as scene size or resolution grows; measured results show up to 71× faster partitioning and over 500× faster runtime queries in representative workloads, with further gains expected for larger problem sizes. The evaluation also reveals that single-threaded physics overlap queries dominate runtime cost in many scenes (>90% in our tests) and that the chunked design increases implementation complexity. Overall, this thesis makes the SFS practical for much larger scenes and provides a roadmap for mitigating remaining bottlenecks. The implementation is available on Github [1].

Keywords: Chunking, Parallelization, Voxel Flood-Fill, Unity, Spatial Indexing, Game Engineering, Game Space

[1] `https://github.com/dyrdadev/space-foundation-system`

Acknowledgements

I would like to thank Daniel Dyrda for his invaluable guidance and support throughout the course of this thesis. His thoughtful feedback and readiness to answer my questions were invaluable to the completion of this thesis.

I would also like to thank Michael Grupp for this LaTeX template.

Contents

1 Introduction

In digital games, space is more than an empty container for gameplay. It shapes how players navigate challenges, perceive the world, and experience a narrative [54, 55]. Game designers often work with rich spatial concepts such as regions, pathways, and points of interest, using them to determine where events occur, how encounters are structured, and which mechanics are available at a given time. This high-level, conceptual view of space rarely matches the low-level constructs—coordinates, colliders, raycasts—that most game engines provide.

The *Space Foundation System* (SFS) by Dyrda and Belloni [15] was proposed to address this gap. Instead of treating location as a raw geometric value, the SFS introduces a higher-level abstraction: the game world is partitioned into discrete, semantically meaningful locations, connected in a graph structure. This representation supports gameplay logic that depends on where the player or objects are located, without relying solely on physics-based detection. Designers benefit from working with constructs that align with their creative intent, rather than reverse-engineering their ideas into engine-level mechanics.

1.1 The Original SFS Model

The SFS resolves these issues by representing space as a *location graph*, derived from *anchors* (central reference points) and *delimiters* (boundaries). This enables runtime queries such as adjacency checks, pathfinding, and location-change subscriptions. By enriching raw spatial data with semantic meaning, it simplifies implementation, supports dynamic features, and provides a consistent framework for location-aware systems. With that, it is a powerful tool for designers, as it allows for high-level development, testing, and player experience measurements, as well as the creation of adaptive gameplay systems. Furthermore, it can facilitate automated balancing, real-time analytics, accessibility adaptations, and rapid prototyping, ultimately supporting more efficient workflows and richer player experiences.

1.2 From Theory to Implementation

The original SFS paper [15] established the theoretical model and a sequential algorithm for generating location graphs, but no full implementation was initially available. A later proof-of-concept Unity plugin filled this gap, using a frontier-based flood fill on a voxel grid to construct the graph from predefined anchors and delimiters. At runtime, this system can determine the anchor containing a given object's position and answer spatial queries accordingly. Although it demonstrates the viability of the SFS concept, performance issues emerge in very large worlds, where graph creation and real-time queries slow down considerably.

1.3 Problem Statement and Research Questions

The reference implementation of the Space Foundation System serves as a commendable proof of concept. It successfully demonstrates the feasibility of the underlying ideas and delivers promising initial results. Its strength lies in its simplicity and the fact that it achieves functional correctness with minimal overhead, making it an excellent starting point for further development.

Despite its strengths, the implementation reveals several limitations. It struggles to scale efficiently and fails to fully utilize available system resources. Performance bottlenecks arise due to the absence of parallel processing and memory-efficient data handling, and the system does not take advantage of Unity's performance-oriented technologies such as the C# Job System or the Burst compiler.

This leads to the central research question of this thesis:

> **How can chunking and parallelization improve the Space Foundation System's efficiency during both editor time and runtime?**

In addition, this work aims to answer the following related questions:

- How do both the sequential and the chunked parallel implementation compare in terms of performance?
- What is their theoretical complexity?
- Which optimizations does Unity offer for such systems?
- What architectural changes are necessary for the chunked implementation?

1.4 Contribution

This thesis builds on that foundation by introducing a new implementation of the SFS that focuses on scalability and runtime performance. We refer to the proof-of-concept's original approach as the *Sequential Algorithm*, and our improved system as the *Chunked Parallel Algorithm*. Inspired by established practices in game engine architecture [30] and spatial data processing [13], we partition the game world into independent *chunks*, enabling the algorithm to run in parallel for significant performance gains.

Our system retains the conceptual benefits of the SFS while integrating Unity's high-performance programming features[66], such as the C# Job System [70] for parallel execution and the Burst compiler[65] for optimized low-level code. The result is an implementation that scales more effectively to large worlds, responds faster to spatial queries, and moves the SFS closer to a robust, production-ready tool for game development.

2 Background

2.1 Conceptual Framework of the Space Foundation System

The Space Foundation System (SFS) [15] treats a game's environment as a network of connected locations. A location defines a specific volume within the game world, and its scale or level of detail can vary widely[3]. This approach breaks a continuous game world into manageable segments, linked together in a location graph. Integrated into the game engine as a location-aware API, it extends the usual `Transform` components so that objects can check and subscribe to their current location in real time. This avoids the need for custom trigger zones or collision checks and makes it easier to implement spatial rules, visual design choices, and story events [15].

The method builds on ideas that are common not only in games but also in architecture [3]. Many spatial constructs come from the way buildings and cities are planned and understood. In this context, two core elements are used to define spatial integrity: *anchors* and *delimiters*.

Anchors are central features that act as reference points for a space. They act as the origin of a location—similar to how a landmark defines the character of its surroundings [44]. In a game, an anchor could be something like a central fountain in a plaza or a fireplace in a living room. Space can then be understood in relation to the anchor—what is part of it and what lies beyond.

Delimiters, on the other hand, are boundaries that separate one area from another. They might be walls, fences, or other structures that break the visual or physical flow of space. They are thought of as lines or planes that mark what is on each side [3]. In digital environments, delimiters help to signal clear transitions, such as moving from one room to another or from outdoors to indoors.

By shifting from raw coordinate handling to this higher-level understanding of locations, the SFS makes spatial logic in games easier to design and maintain.

2.2 Spatial Semantics in Game Development: Current Practice and Gaps

While contemporary game development offers designers a variety of tools for narrative authoring, quest management, level design, and telemetry, the handling of spatial semantics—representing meaningful in-game locations as first-class concepts—remains fragmented across production pipelines. In most engines, spatial logic is implemented through low-level constructs such as trigger volumes, collision shapes, or navigation areas, with semantic meaning encoded through naming conventions, tags, or ad hoc metadata [48]. Designers place such volumes in level editors to detect player presence (*e.g.*, `OnTriggerEnter` in Unity, `BeginOverlap` in Unreal Engine) and attach scripted

events to initiate quests, spawn entities, or progress narrative states [48, 25, 64, 16].

For semantic annotation, AAA studios often extend these primitives with tagging systems (*e.g.*, Unreal Engine's Gameplay Tags), custom components, or in-house registries of named locations [23, 26]. Narrative integration is commonly achieved by linking authored content from dedicated narrative tools, such as Articy:draft or open-source systems like Ink and Yarn Spinner, to engine-side triggers [27, 28, 43, 39]. While such integrations allow designers to connect story beats to spatial triggers, the underlying location model remains tied to engine geometry rather than an abstract, engine-independent graph of semantically meaningful spaces.

In AAA practice, spatial semantics are frequently implicit: quest logic or AI behaviours query for overlaps with tagged volumes, search for nearby objects in spatial databases (*e.g.*, Unreal's Smart Objects), or use navmesh area types to gate actions [24]. Large-scale systems, such as Ubisoft's DNA telemetry platform, handle spatial partitioning for analytics or distributed simulation, but these are infrastructure-level solutions rather than designer-facing abstractions [26]. Open-source engines like Godot provide analogous primitives (Area2D/3D nodes with metadata), but likewise require per-project implementation to achieve high-level semantic modelling [16].

This fragmented approach stands in contrast to the *Space Foundation System* (SFS) [15], which explicitly models the game world as a graph of semantically meaningful locations. By decoupling location semantics from raw geometry and exposing them as a designer-first abstraction, the SFS offers a unified mechanism for integrating quest logic, AI behaviours, world streaming, and analytics. Given that current practice relies on disparate, engine-specific solutions—often stitched together with custom tools—an approach like SFS could streamline workflows, improve maintainability, and better align spatial representation with designers' creative intent.

2.3 Chunking

Chunking is a foundational technique used in both general computer science and game development, where large datasets, computations, or spatial environments are divided into smaller, more manageable units called *chunks*. These chunks are typically of fixed size and structure, and they serve to optimize various aspects of system performance, such as memory usage, computation time, and responsiveness.

In general computer science, chunking is employed in areas such as memory management, data processing, and network communication. For instance, memory allocators may divide heap memory into chunks to avoid fragmentation and to make allocation and deallocation more predictable and efficient. In distributed computing systems, large datasets are chunked and distributed across different processing nodes, as seen in the MapReduce framework [12]. Similarly, in data streaming protocols like HTTP/1.1 chunked transfer encoding, data is transmitted in a series of chunks, allowing the receiver to begin processing before the entire message is received.

In game development, chunking is particularly crucial in the context of large or procedurally generated worlds [57]. Game environments are often partitioned into spatial chunks, which are units of the game world that can be loaded, unloaded, or updated independently. This approach ensures that only the parts of the world relevant to the player's current position are kept in memory and actively processed. It reduces memory

consumption, minimizes CPU usage, and allows for scalable world design.

2.3.1 Advantages of Chunking

The primary advantage of chunking lies in its ability to improve performance through spatial and temporal locality. In both computer systems and games, chunking reduces memory overhead by avoiding the need to load or process the entire dataset or world simultaneously. This leads to better cache utilization and more efficient use of system resources [32]. Additionally, chunking facilitates parallel and asynchronous processing. Different chunks can be handled independently by separate threads or processes, enabling improved scalability and responsiveness. In systems that require real-time processing, such as interactive games or high-frequency trading systems, the reduction in latency afforded by chunked processing can be critical [56].

Another key benefit is the modularity it provides. Chunks serve as natural boundaries for system logic, making debugging, testing, and updating specific areas more manageable. This modularity also improves the ability to stream or serialize game data, which is beneficial in networked multiplayer environments or cloud gaming contexts.

2.3.2 Disadvantages of Chunking

Despite its advantages, chunking introduces additional complexity into system design. The logic required to manage the dynamic loading and unloading of chunks can become intricate, particularly when objects span multiple chunks or when chunk boundaries interact with physics, AI, or visibility systems.

Furthermore, improperly sized chunks may result in inefficient performance. If chunks are too small, the system incurs overhead from managing a large number of them, including metadata and synchronization costs. If they are too large, the memory and computational benefits of chunking are diminished. Synchronization and concurrency issues also arise when multiple threads or systems interact with the same or adjacent chunks, necessitating locking mechanisms or other safeguards that may degrade performance.

2.3.3 Chunking in *Minecraft*

A prominent example of chunking in video games is found in *Minecraft* [1] , which divides its virtually infinite 3D world into chunks of 16×16×384 blocks. This system enables efficient memory and CPU usage by loading only the chunks within a certain radius of the player, while dynamically unloading those that fall outside the view distance.

Chunking also facilitates procedural generation, as terrain is created on-the-fly when new chunks are needed. Each chunk is saved independently, allowing for modular world storage and efficient loading.

However, Minecraft also shows the limitations and drawbacks of chunking. Terrain pop-in can occur when moving quickly or using large view distances, disrupting visual continuity. In addition, in-game systems like automated farms may stop functioning when their chunks are unloaded, prompting players to use mechanisms known as chunk loaders to keep specific areas active.

[1] Minecraft. Mojang Studios, 2011. `https://www.minecraft.net/`.

2.4 Hardware and Cache Optimization

Modern x86 CPUs feature deep memory hierarchies and wide vector units. Exploiting cache locality is essential for high performance. Shatdal et al. [58] demonstrated that cache-conscious algorithms can outperform naïve implementations by 8% to 200%, depending on the specific workload and data access patterns.

Key optimization strategies include:

1. **Data layout and alignment**: Store voxel data contiguously (e.g., in arrays or struct-of-arrays), aligned to cache-line or vector boundaries for efficient vector loads.
2. **Block processing (tiling)**: Partition the 3D grid into sub-volumes that fit in cache; process each fully before moving to the next to maximize temporal reuse [74].
3. **Access pattern optimization**: Use sequential memory access in inner loops to avoid cache thrashing; minimize random or strided accesses [51].
4. **False sharing avoidance**: Assign separate cache lines to different threads to prevent unnecessary invalidations in multithreaded code.

These techniques reduce effective memory latency and bandwidth demand by ensuring that once data is loaded into L1/L2 cache, it is reused extensively.

Space-filling curves such as Morton (Z-order) and Hilbert curves are often used to linearize multi-dimensional data while preserving spatial locality, improving cache performance in large or scattered datasets [47, 73]. However, in our case, this optimization proved less critical, as each chunk contains only highly condensed data and remains relatively small—on the order of kilobytes. For larger chunk sizes, however, such techniques may offer worthwhile performance gains.

2.5 SIMD Vectorization

Modern x86 CPUs feature SIMD (Single-Instruction-Multiple-Data) extensions such as SSE, AVX, AVX2, and AVX-512, operating on 128–512 bits per instruction. These allow one instruction to process multiple elements in parallel, with theoretical throughput gains of up to $16\times$ when 512-bit registers are fully utilized [34]. In practice, overheads reduce this, but substantial speedups remain. For example, Franchetti et al. show fully-vectorized FFT (Fast Fourier Transform) kernels achieving $\sim 3.3\times$ speedups over scalar code [20].

In this work, SIMD instructions are not used directly but leveraged indirectly through the Unity Burst compiler's auto-vectorization capabilities. Burst operates on a carefully selected subset of C#, which is designed to be amenable to auto-vectorization by the compiler. This approach enables effective SIMD utilization without manual management of vector intrinsics or explicit SIMD programming. Once data is in cache, SIMD packing can execute up to sixteen 32-bit operations per instruction on AVX-512 hardware, translating to several-fold speedups in compute-bound kernels.

2.6 Parallel Threads and Multi-Core Scaling

Modern CPUs provide multiple cores, making thread-level parallelism essential. Unity's C# Job System enables safe multithreaded execution across all available CPU cores using worker threads—typically matching the number of hardware threads—to efficiently parallelize tasks and reduce context-switching overhead [71, 70]. It employs a work-stealing scheduler, allowing idle threads to "steal" tasks from busy ones to balance load and maximize CPU utilization [71, 7].

To achieve effective multithreading, data should be partitioned into independent segments assigned to separate threads[6]. This approach minimizes contention and false sharing while improving cache locality. Each thread performs lock-free updates by working on its own data segment and accumulates results locally, merging them only after processing to reduce synchronization overhead.

Together, SIMD (intra-core) and multithreading (inter-core) enable near-linear scaling with core count, bounded primarily by Amdahl's law [1] and memory system limitations.

2.7 Unity and Performance Improvements

The reference implementation of the SFS is realized as a plugin for the Unity Engine. Unity, as a widely adopted real-time development platform, offers a flexible environment for developing interactive systems and simulations. Its extensive ecosystem, strong C# support, and modular component-based architecture make it particularly suitable for rapid development and iterative design workflows. Moreover, Unity's support for plugins allows for the modular extension of engine capabilities without the need to modify the engine source code. This is especially advantageous for experimental or research-driven projects like SFS, which benefit from being encapsulated, reusable, and easily maintainable across different Unity projects.

The sequential implementation of the SFS primarily relies on standard Unity features and C# data structures. The system makes use of efficient native types such as `HashSet` for rapid lookup operations and lightweight `struct`s for small data containers.

2.7.1 Data-Oriented Technology Stack

Despite this solid baseline, the sequential implementation does not yet take full advantage of the more advanced performance-oriented tools and paradigms offered by Unity and C#. Unity's Data-Oriented Technology Stack (DOTS) is a performance-focused framework designed to help developers write highly optimized, scalable code by organizing data in memory-efficient ways[66]. It includes the Entity Component System (ECS) for structuring game logic around data, the C# Job System for safe multithreading, and the Burst Compiler for compiling code to highly optimized native instructions.

DOTS is particularly well-suited for projects involving large numbers of entities or complex simulations, as it improves CPU performance by maximizing cache efficiency and parallel execution. However, it comes with a steep learning curve, limited compatibility with traditional GameObject/MonoBehaviour workflows, and some features (e.g., physics or animation) are still in development or not fully integrated, which can make it

harder to adopt for smaller projects or legacy codebases.

2.7.2 Entity Component System

The Entity Component System (ECS) is a core part of Unity's DOTS framework that enables highly efficient data organization and parallel processing by separating data (components) from behavior (systems)[67]. It is particularly useful for large-scale simulations or games with thousands of entities, as it improves performance through cache-friendly memory layouts and multithreaded execution. However, ECS introduces a fundamentally different programming model compared to Unity's traditional GameObject and MonoBehaviour workflow, which can increase complexity and reduce compatibility with existing systems. For this reason, we chose not to integrate ECS into the Space Foundation System. Adopting ECS would have forced all users of the plugin to restructure their projects around it, creating a significant barrier to adoption that we aimed to avoid.

2.7.3 C# Job System

To support multithreaded workloads, our new implementation of the Space Foundation System makes extensive use of the C# Job System, a core component of Unity's DOTS [71, 70]. The Job System allows developers to write parallelized code that runs efficiently across multiple CPU cores, enabling significant performance gains for compute-heavy tasks. Unlike traditional threading approaches in C#, Unity's Job System emphasizes safety, performance, and determinism. It uses a scheduling mechanism to ensure that jobs run in the correct order and enforces strict rules on memory access to prevent race conditions.

Blittable Types and Constraints

A key constraint of the C# Job System is that jobs can only operate on *blittable types*—value types with a fixed memory layout that can be copied directly to unmanaged memory. These types include primitive types like `int`, `float`, and structs that contain only other blittable fields. As a result, jobs cannot directly access reference types, such as classes, arrays, `MonoBehaviours`, or any object derived from `UnityEngine.Object`, nor can they call most of Unity's managed APIs, including the rendering or physics systems. This limitation necessitates a data-driven architecture where logic is moved off the main thread and data is passed in and out of jobs in a controlled and predictable manner.

Unity itself remains largely single-threaded at its core, especially in subsystems like rendering, UI, and physics. While it supports asynchronous operations in areas such as asset loading or file I/O, most Unity APIs are not thread-safe and must be executed on the main thread. Consequently, the Job System becomes essential for offloading CPU-intensive work safely, enabling parallel execution while avoiding unsafe memory access or thread contention. However, this imposes a design trade-off: developers must carefully isolate and prepare data to be passed into jobs and later synchronize the results back to the main thread when necessary.

Available Data structures

To facilitate safe data sharing between jobs and across threads, Unity provides a set of specialized data structures known as *Native Containers*. These containers are blittable, thread-safe collections designed specifically for use within the Job System and the Burst Compiler. Here are some examples being used by the chunked parallel algorithm in our implementation:

- `NativeArray<T>`: A fixed-size array with fast, low-level memory access.
- `NativeList<T>`: A resizable list similar to `List<T>`.
- `NativeBitArray`: A compact array for managing individual bits efficiently.
- `NativeQueue<T>`: A thread-safe FIFO queue.
- `NativeHashSet<T>`: A high-performance set for storing unique elements.
- `NativeHashMap<TKey, TValue>`: A key-value map with constant-time access.
- `NativeParallelMultiHashMap<TKey, TValue>`: A map allowing multiple values per key, supporting parallel writes. It is particularly useful for grouping elements.

These Native Containers differ from standard Unity collections in that they are allocated in unmanaged memory, support deterministic disposal via the `Dispose()` method or the `[DeallocateOnJobCompletion]` attribute, and are compatible with Burst-compiled jobs. Their use is essential when working within the Job System, as managed collections (such as `List<T>` or `Dictionary<TKey, TValue>`) are not permitted due to their heap allocation and garbage collection overhead.

Unavailable Data structures

However, while Unity provides a range of Native Containers, it lacks certain specialized data structures that can be useful in performance-critical algorithms. Notably, there is no built-in *priority queue* (i.e., a heap-based data structure that allows access to the element with the highest or lowest priority). This absence can be limiting for certain algorithmic patterns—such as pathfinding with A* or task scheduling—where a priority queue would offer optimal performance characteristics. In such cases, developers must either implement custom solutions using existing containers (e.g., manually managing a heap with a `NativeArray<T>`) or restructure their algorithms to avoid the need for priority-based access altogether.

Despite these limitations, the C# Job System—together with Native Containers—offers a powerful foundation for building high-performance, data-oriented systems in Unity. When used appropriately, they enable scalable, multi-core execution while maintaining memory safety and determinism, aligning well with the architectural goals of modern game and simulation development.

2.7.4 Burst Compiler

To further enhance performance, the Space Foundation System leverages Unity's Burst Compiler. Burst is a just-in-time (JIT) compiler that translates C# jobs into highly optimized native code using LLVM[65]. It applies low-level optimizations such as vectoriza-

tion, loop unrolling, and efficient memory access patterns, which can significantly reduce CPU workload and improve cache utilization. By compiling job code to platform-specific machine instructions, Burst can achieve performance levels close to hand-written C or C++ in many scenarios. However, using Burst requires adherence to strict constraints. Only a subset of C# is supported, and many common language features—such as exceptions, dynamic memory allocation, and virtual method calls—are disallowed. In our case, adopting Burst necessitated a substantial refactoring of the sequential implementation to remove unsupported patterns and ensure that all data passed into jobs was blittable and statically typed. Despite the efforts, Burst yields a significant speedup in the chunked parallel implementation (see Section 8.2.6).

3 Related Work

The problem addressed in this work—multi-seeded, obstacle-aware segmentation of volumetric environments—intersects with several well-established research areas, including spatial partitioning, pathfinding, distance transforms, and parallel graph algorithms. Existing approaches span from classical computational geometry and grid-based search methods to GPU-accelerated algorithms and recent machine learning techniques. While each method offers unique strengths, their suitability varies depending on dimensionality, obstacle handling, accuracy requirements, and parallelization potential. This section reviews representative techniques across these domains, highlighting their relevance to the Space Foundation System (SFS) and identifying the constraints that ultimately guide the design choices for our new implementation.

3.1 Space Subdivision Methods

While the Space Foundation System (SFS) currently relies on a voxel-based flood-fill approach for spatial segmentation, other space partitioning methods are widely used in computer graphics, robotics, and game development. This section briefly reviews common approaches, starting with tree-based methods, followed by navigation meshes, and finally voxel grids, before explaining why voxels are the most practical choice for the present implementation.

3.1.1 Tree-Based Partitioning

Binary Space Partitioning (BSP) was originally developed for visible surface determination in 3D scenes [21], but has since been adapted for a wide range of applications such as Constructive Solid Geometry (CSG), ray tracing, image compression and machine learning[69, 63, 50, 17]. Related hierarchical partitioning schemes include *octrees* [46, 40, 41], which recursively divide space into eight octants, and *k-d trees* [5], which split along axis-aligned planes to accelerate multidimensional searches. Hybrid techniques, such as those described by [75], combine voxel-based methods with polygonal partitioning to reconstruct complex indoor environments, particularly when handling irregular or non-Manhattan geometries. A broader overview of these and related techniques is provided in [11].

3.1.2 Navigation Meshes

Navigation meshes (navmeshes) represent traversable regions of an environment as a set of convex polygons[2, 72]. These are widely used for pathfinding because they reduce navigation problems to graph searches between convex cells. However, navmeshes are typically two-dimensional and rely heavily on convexity assumptions, making them well

suited for ground-based navigation but less appropriate for fully volumetric or obstacle-aware space segmentation tasks such as those in SFS.

3.1.3 Voxel Grids and Uniform Subdivision

Uniform grids divide space into equally sized cubic cells (voxels), enabling simple indexing and adjacency computation. This representation can be extended into hierarchical grids such as octrees or loose octrees to improve memory efficiency in sparse scenes. However, hierarchical structures introduce additional complexity for massively parallel algorithms, particularly for flood-fill propagation and neighbor indexing.

3.1.4 Choice of Voxel Representation

For the purposes of the SFS implementation, voxels provide a pragmatic and effective representation. Their structural simplicity and regular subdivision of space make them well-suited for parallel processing, as adjacency and indexing can be expressed through straightforward arithmetic on grid coordinates. This allows each voxel to access a fixed set of neighbors in constant time, which simplifies the implementation of multi-seed flood-fill and wavefront propagation algorithms while also supporting inherent parallelism.

Voxel data can be stored in contiguous arrays or 3D textures, which in turn facilitates SIMD optimizations, efficient Burst compilation in C#, and GPU compute kernels with minimal pointer chasing—factors that are highly relevant for performance in this context. Moreover, voxel grids align naturally with volumetric propagation techniques such as multi-seed distance fields, occupancy-aware segmentation, and connected component labeling, which can be expressed in terms of grid-based breadth-first search or wavefront algorithms.

Finally, voxel grids lend themselves to partitioning into fixed-size chunks. This approach enhances spatial locality, enables streaming and paging, and allows for parallel chunk processing. In combination, these properties make voxels a natural and effective fit for the implementation strategy pursued in this work.

3.2 Criteria for Candidate Algorithms

Given the considerations discussed in the previous section, the scope of this work is restricted to voxel-based algorithms. Specifically, we seek an algorithm that meets the following criteria:

- **Voxel-based:** The algorithm must operate on a three-dimensional grid of voxels.
- **Multi-seeded:** The algorithm must support multiple anchors (often referred to as *seeds* in the literature).
- **Obstacle-aware:** The exploration process must account for delimiters, which block or influence the filling process.
- **Parallelizable:** The algorithm should be designed for efficient parallel execution, while preserving determinism and ensuring consistent subspace assignments for each seed.

In summary, the goal is to identify or develop a multi-seeded, obstacle-aware, 3D voxel-based flood fill algorithm that can be executed in parallel without compromising consistency or correctness.

3.3 Multi-Seed BFS and Multi-Source Dijkstra

The sequential implementation of the SFS is based on a frontier-based breadth-first search (BFS) approach, treating the voxel grid as an unweighted graph. All anchors (or seeds) are initially enqueued into a frontier queue, which then expands in uniform layers. Each voxel is labeled by the first seed that reaches it, ensuring an exact region assignment under the chosen grid metric (e.g., Manhattan or Chebyshev distance). Blocked voxels are never enqueued, allowing the expansion to wrap around obstacles efficiently. This approach has two beneficial properties: (i) exact and consistent region labeling without storing explicit distances, and (ii) natural handling of delimiters. The main drawback is its poor parallelizability, due to the global dependency on the frontier.

For accurate distances such as Euclidean, the BFS is replaced with a multi-source Dijkstra's algorithm [29], which assigns each seed a starting distance of zero and expands voxels in order of increasing distance using weighted edges (e.g., 1 for orthogonal moves, $\sqrt{2}$ or $\sqrt{3}$ for diagonals). This method supports varying travel costs and produces accurate distance fields, but requires a priority queue with $\mathcal{O}(N \log N)$ complexity. A continuous variant is the Fast Marching Method (FMM), which propagates distances through the domain while respecting obstacles as infinite-cost boundaries, producing smooth Euclidean fields at higher computational cost[29].

3.3.1 Relation to Voronoi Diagrams and Sweep Line Methods

A *Voronoi diagram* [4] partitions space into regions based on proximity to a set of sites. The SFS without explicit delimiters can be interpreted as a voxelized 3D Voronoi diagram. While many algorithms for Voronoi diagrams exist, incorporating obstacles such as delimiters is non-trivial and often requires simplifying assumptions, for example restricting obstacles to axis-aligned bounding boxes (AABBs) [38].

In 2D, Fortune's sweep line algorithm [19] efficiently computes general Voronoi diagrams by moving a line across the domain and updating the diagram incrementally. However, this approach is limited to two dimensions and has no straightforward generalization to 3D [19].

3.3.2 Parallel Extensions for Multi-Seed BFS and Voronoi Diagrams

Several works extend classic multi-seed region growing to exploit parallel hardware while preserving deterministic labeling. In voxel grids, obstacles are naturally handled by excluding blocked voxels from the frontier, so propagation halts at boundaries.

Early work by Singh et al. [60] assigned each seed to a separate processor, allowing regions to grow concurrently. Gaihre et al. explores adaptive execution techniques and workload-aware optimizations for BFS traversal strategies to optimize on GPU architectures [22]. Park et al. [49] demonstrated a GPU-based 3D implementation for medical volumes, mapping seed expansions to CUDA threads and achieving significant speedups.

More recently, Couder-Castañeda et al. [10] proposed a lock-based OpenMP approach, where threads claim voxels before processing to ensure exclusive ownership. These methods adapt naturally to irregular connectivity, but naive parallelization can lead to contention when frontiers meet; synchronization mechanisms such as locks or atomics resolve conflicts at the expense of additional overhead.

From a graph-theoretic perspective, the voxel grid can be treated as a graph in which obstacles remove nodes or edges. Then et al. [68] introduced a multi-source breadth-first search (MS-BFS) framework that interleaves expansions from all seeds in a data-parallel manner. This approach often uses bit-vector frontiers to represent active voxels, enabling efficient handling of overlapping waves while avoiding fine-grained locking. Since voxel labels are assigned to the first wave that reaches them, MS-BFS produces deterministic results and remains inherently obstacle-aware, as connectivity constraints are already embedded in the graph representation.

The Jump Flood Algorithm (JFA) is a fast, approximate method for generating Voronoi diagrams and distance fields on a grid using parallel computation [52]. It works by repeatedly "jumping" information from each site across the grid in progressively smaller steps, allowing distant pixels to quickly acquire approximate nearest-site data. With each pass, the jump distance is halved, refining the accuracy, making JFA especially well-suited for implementation on GPUs due to its efficiency and parallel nature. In total, $\mathcal{O}(logN)$ passes are required to fill an image. Rong and Tan [53] discuss variants of the JFA, also for 3d, but in its vanilla form it assumes a free grid without obstacles so every voxel can freely inherit the nearest-seed label. The delimiters and anchors' maximum spreading distance of the SFS however pose a problem. Incorporating these constraints into JFA is possible in principle, but it requires efficient checks for voxel intersections or ray tests, which may negate JFA's performance advantages in the presence of many obstacles.

For 3D grids, Hsieh and Tai [33] propose a region-map scheme where each voxel is quickly labeled by scanning in GPU memory: they incrementally build a voxel-space map of nearest sites so that lookups are trivial. Govindaraju et al. [61] developed a GPU algorithm that computes a discretized Euclidean distance field slice-by-slice using linear factorization, effectively computing a generalized Voronoi of arbitrary primitives. Building upon these approaches, Dorn et al. [14] further improve efficiency and accuracy for complex CAD scenarios by employing a GPU-based voxelization technique combined with an error-bounded wavefront propagation and a novel Voronoi Voxel History (VVH) data structure. These GPU methods can be very fast for large voxel grids, at the cost of specialized hardware and (sometimes) approximate results.

3.4 Distance-Transform Methods

Distance transform (DT) algorithms compute, for each voxel, its distance to the nearest feature. One can treat the seeds as "features" and compute a multi-seed DT. Classic algorithms (e.g. chamfer transform [9] or the 3D extension of Felzenszwalb and Huttenlocher's EDT [18]) run in linear time but typically ignore obstacles – they compute Euclidean distance in free space, not geodesic distance around obstacles. To account for obstacles, one approach is to first mask out solid voxels (so the transform only fills free space) or to compute a geodesic distance transform by dynamic programming on the grid (which essentially reduces to graph BFS/Dijkstra as above). In practice, one may compute a raw DT on the free voxel mask and then relabel by nearest seed, but this can

incorrectly assign voxels "behind" obstacles. More advanced work (e.g. the CSC algorithm [45]) instead restricts the DT to each seed's Voronoi cell volume, e.g. by computing bounding volumes for each cell via mesh connectivity, so distance computation is limited to relevant voxels. For mesh or implicit inputs, GPU-based DT algorithms can build slices of the 3D distance field efficiently [62].

3.5 Machine Learning Approaches

Several recent publications propose flood filling networks (FFNs) [37, 31, 36] and other machine learning-based approaches for the task of region segmentation in 2D images, particularly within the fields of object detection and image analysis. While these methods are effective in their respective domains, they are not directly applicable to the problem addressed in the Space Foundation System (SFS).

One key reason is that the SFS requires a deterministic and analytically correct solution. In contrast, learning-based methods generally yield approximate solutions and are not guaranteed to respect constraints like obstacles and maximum spreading ranges in all cases[36]. Additionally, the reviewed approaches report significant training times and computational overhead of FNNs[37, 36], which would be impractical for integration into SFS, where responsiveness and reproducibility are essential. Additionally, FFNs demand large amounts of ground truth data for training [31], which is not readily available and difficult to generate in the context of the SFS.

While machine learning could be a potential direction for future exploration—especially for scenarios where exact constraint satisfaction is less critical—it will not be pursued further in this work due to the need for precision, interpretability, and algorithmic control in the current context.

3.6 Towards a Parallel SFS Algorithm

The current SFS implementation uses a frontier-based multi-seed breadth-first search on the unweighted voxel graph. This implementation is simple and exact with respect to the chosen grid metric, and it benefits from the property that layer-wise propagation obviates per-voxel distance storage.

However, frontier-based BFS is inherently global, and naive parallelization is difficult because of the sequential dependency of frontier layers. The literature suggests several alternative strategies to reduce contention and increase parallelism. Multi-source BFS variants interleave expansions in a data-parallel manner, which can avoid fine-grained locking. Lock-based region growing enforces exclusive ownership with atomic operations or mutexes, which is straightforward but introduces synchronization overhead. Multi-source Dijkstra or Fast Marching are suitable when accurate geometric distances are required, but they bring higher complexity. GPU algorithms and approximate schemes can achieve very large speedups for certain applications, but struggle with effective handling of delimiters.

Drawing on insights from the reviewed literature and empirical observations, the design decisions underlying the new chunked parallel SFS algorithm are informed by established best practices in parallel computing and tailored to the specific performance

characteristics of the target architecture. A chunk-based architecture mitigates global contention by partitioning the domain; the use of C# Jobs and Burst enables high-throughput parallel execution on the CPU; and carefully defined per-chunk ownership and merge rules eliminate the need for coarse-grained global synchronization while preserving algorithmic correctness.

4 Methodology

4.1 Research Approach

This work begins with a detailed description and analysis of the existing sequential Space Foundation System (SFS) algorithm and its implementation in Unity. The process starts with introducing the Unity-based implementation and all core components, followed by the creation of the location graph, performance analysis, and time and space complexity evaluation, with a stronger emphasis on time complexity. The suitability of the algorithm for the given problem is discussed, along with an examination of runtime query performance and its theoretical complexity.

The research follows an engineering-oriented approach, developing a chunk-based system that incorporates findings from related work, namely theoretical optimizations like chunking and parallelization and practical using C# Jobs and the Burst compiler. The primary goal is to improve performance and scalability in complex scenarios. This includes theoretical improvements based on the work–span model and practical experiments to measure the effects of C# Jobs and Burst. Not all available tools are used: for example, Unity's Entity Component System (ECS) is deliberately excluded to maintain adaptability for non-ECS developers. Correctness is a strict constraint, ensuring identical results to the sequential version, and the implementation must integrate cleanly into the Unity ecosystem and the existing API interfaces of the SFS.

4.2 Scope and Boundaries

The methodology is restricted to Unity-specific optimizations, excluding approaches such as custom C++ implementations, alternative physics engines, or GPU-based processing. GPU processing is omitted due to the significant increase in implementation complexity, hardware constraints, and the difficulty of implementing flood fill algorithms on GPUs as highlighted in related work. The solution is intended to remain lightweight and not depend on specialized hardware.

The focus lies on chunking as a spatial subdivision method and Exploiting C# Jobs and Burst for parallel execution and low-level optimization. In addition, the emphasis is primarily on performance and scalability improvements, with less focus on memory usage and storage efficiency of the results.

4.3 Overall Workflow

The research process consists of four main stages:

1. **Baseline Analysis** – Introduction of the sequential algorithm, complexity analysis, and bottleneck identification.

2. **Proposed Method** – Design and implementation of the chunked parallel algorithm. This includes a high-level overview, handling of issues such as cutoff voxels, a detailed breakdown of key algorithmic components, and complexity analysis using the work–span model. Both SFS creation and runtime queries are analyzed.
3. **Optimizations** – Targeted performance improvements to exploit modern hardware capabilities, including caching, chunking architecture, lock-free parallel computing strategies and memory efficiency optimizations.
4. **Evaluation** – Benchmarks comparing the sequential and chunked algorithms, covering execution time, scalability, optimal chunk sizes, and effects of compiler optimizations.

5 The Sequential Algorithm

This chapter presents the design and implementation of the existing sequential algorithm for the Space Foundation System (SFS). Figure 5.1 shows the SFS creation in the scene *Villa*.

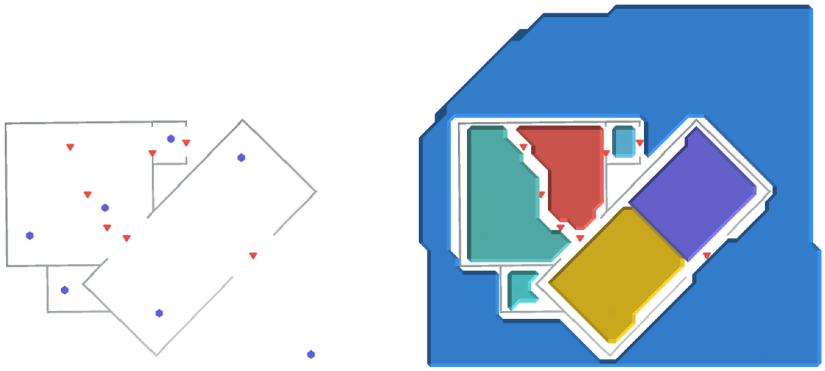


Figure 5.1: SFS creation in the Scene *Villa*. Initial state (left) and result after creation (right). Anchors are denoted as blue hexagons, Delimiters as red triangles.

5.1 Space Foundation System Components

The current implementation of the Space Foundation System (SFS) is developed as a Unity plugin (Unity 2022.3.58f1 LTS, Mono scripting backend, .NET 2.1, C# 8.0). It provides a framework for defining, managing, and querying spatial subdomains in a voxel-based environment. The core algorithm is a multi-seeded, obstacle-aware, three-dimensional voxel flood fill that uses a frontier-based Breadth-First Search (BFS) approach to partition the scene. The flood fill starts from predefined anchors and expands within the scene while respecting both explicit and implicit boundaries. This process produces a location graph and voxel occupancy data that are later used for efficient runtime spatial queries.

5.1.1 The SpaceFoundation

The central component of the system is `SpaceFoundation`, which acts as the single point of access to the SFS context in both the Unity Editor and at runtime. It stores configuration parameters such as the voxel size and a physics layer mask for detecting delimiters. In the Editor, it also manages preprocessing tasks including the assignment of unique anchor identifiers and collection of adjacency information between anchors and delimiters. The results of the SFS creation are stored in a `SpaceFoundationData`

ScriptableObject, which contains the nodes (anchors) and edges (delimiters) of the location graph, and the border voxels of each anchor. At runtime, `SpaceFoundation` loads this data and performs location queries through voxel raycasting.

5.1.2 Anchors

Anchors define the reference origins of spatial locations. They are implemented by attaching an `Anchor` component to a Unity `GameObject`, using the object's transform as the coordinate reference. Each anchor has a unique identifier and a maximum spreading distance that limits the extent of its location. During the flood fill process, each anchor keeps track of its occupied voxels. The `SpaceFoundation` keeps track of each anchor's border voxels and writes them to the `SpaceFoundationData` after the algorithm completes. In runtime mode, anchors no longer store these voxel sets in memory; instead, the `SpaceFoundation` retrieves the border voxels from the precomputed dataset.

5.1.3 Delimiters

Delimiters are objects that block or restrict the expansion of anchors during the flood fill process. They can be explicit, such as colliders placed in the scene to represent walls or logical barriers, or implicit, formed when two growing anchor regions meet without a physical delimiter between them. A `Delimiter` component stores its own identifier and records the list of neighboring anchors, which is saved in and retrieved from the `SpaceFoundationData`. In addition to objects with delimiter components, any collider whose layer matches the configured mask in `SpaceFoundation` is treated as an explicit delimiter, with such colliders grouped and processed as one during the algorithm.

5.1.4 Editor Interface

The generation of SFS data is initiated from the Unity Editor through the `SpaceFoundation` editor window. Running the process executes the flood fill, creates or updates the `SpaceFoundationData` asset, and generates visual meshes of the computed anchor volumes using the marching cubes algorithm[42]. These meshes serve primarily as a debugging and inspection tool. At runtime, the system relies entirely on the precomputed data for fast spatial lookups and interaction logic.

5.2 Algorithm for Creating the Location Graph

The generation of the location graph begins with the collection and initialization of *anchors* and *delimiters* in the scene. Each of these elements is assigned a unique identifier. Anchors are then mapped into the voxel grid, and for each anchor a corresponding voxel is created, represented by a `VoxelData` struct. Each voxel contains detailed information, including its spatial position, a reference to its associated anchor object, the position from which it was explored, and additional metadata (see Section 7.3.2 for more details).

Once all voxels have been created, they are inserted into the *frontier queue*. The core of the algorithm is an iterative process that operates on the frontier queue. While the queue is not empty, the algorithm dequeues the first voxel and attempts to expand the graph by

exploring its neighboring voxels through the `Explore` function. The overall procedure is summarized in the simplified pseudocode shown below:

Algorithm 1: Exploration of a voxel

```
1  Function Explore(currentVoxel):
2      pos ← currentVoxel.position
3      anchor ← currentVoxel.Anchor
4      fromPos ← currentVoxel.exploredFromPosition
5      fromVoxel ← voxelDataDict[fromPos]
6      if anchor.contains(pos) then
           // Case 1: Already inside anchor
7          return
8      if pos ∈ Explored then
9          voxel ← voxelDataDict[pos]
10         fromVoxel.setBorder()
11         anchor.addBorder(fromPos)
12         if pos ∈ delimiterDict then
               // Case 2: Revisited delimiter voxel
13             CheckAndResolveDelimiters(pos, anchor)
14         else
               // Case 3: Revisited non-delimiter voxel
15             voxel.setBorder()
16             voxel.Anchor.addBorder(pos)
17             RegisterImplicitDelimiter(voxel.Anchor, anchor)
18     else
19         voxelDataDict[pos] ← currentVoxel
20         Explored.add(pos)
21         if CheckAndResolveDelimiters(pos, anchor, delimLayer) then
               // Case 4: Newly discovered delimiter voxel
22             delimiterDict[pos] ← d000
23             currentVoxel.markDelimiter()
24             fromVoxel.setBorder()
25             anchor.addBorder(fromPos)
26         else if ¬ anchor.withinInfluence(pos) then
               // Case 5: Outside anchor influence
27             currentVoxel.setBorder()
28             anchor.addBorder(pos)
29         else
               // Case 6: Inside anchor influence
30             anchor.addToSubspace(pos)
31             foreach neighbor ∈ Neighbors(pos) do
32                 EnqueueVoxel(neighbor, anchor, pos)
```

The `Explore` function processes each voxel based on its relationship to the current anchor and the exploration space. Figure 5.2 illustrates all possible cases encountered during exploration. In this example there are two anchors, Blue and Orange, as well as some

delimiters depicted in Grey. Each numbered marker corresponds to one of the six possible cases. The `currentVoxel` is the voxel being processed (i.e., popped from the frontier queue), and the arrow indicates the direction of the `fromVoxel` (the voxel from which it was discovered).

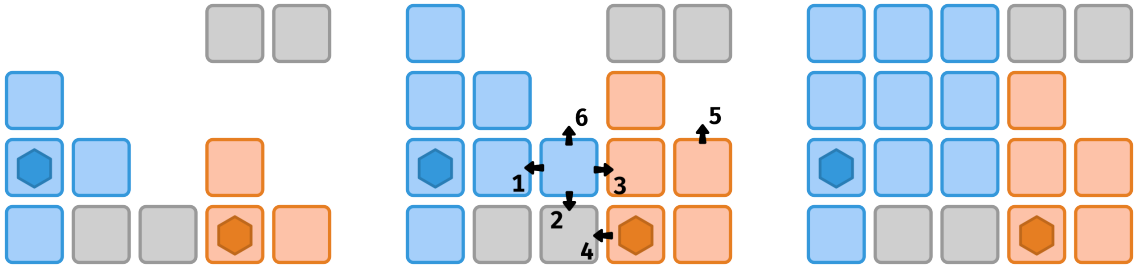


Figure 5.2: Exploration process. Left: after the first step. Middle: intermediate stage showing the six cases handled by the `Explore` function. Right: final result.

1. The `currentVoxel` already belongs to Blue's subspace. No action is taken.
2. The `currentVoxel` has already been explored (by Orange) and overlaps with a delimiter. The `fromVoxel` is marked as a border, and an *explicit* delimiter between the delimiter and Blue is registered—although in this instance, it already exists from a previous iteration.
3. The `currentVoxel` has already been explored and belongs to another anchor (Orange). Both the `currentVoxel` and `fromVoxel` are marked as border voxels, and an *implicit* delimiter between Blue and Orange is recorded.
4. The `currentVoxel` has not yet been explored. It is marked as explored, but since it overlaps with a delimiter, the `fromVoxel` is marked as a border.
5. The `currentVoxel` has not yet been explored but lies outside the influence range of Orange (assuming a maximum anchor distance of 2). It is not processed further, and the `fromVoxel` is marked as a border.
6. The `currentVoxel` is unexplored, within Blue's influence range, and not blocked by any delimiter. It is added to Blue's subspace, and all six of its neighboring voxels are enqueued for further exploration.

5.3 Performance Analysis

We analyze the time and space complexity of the sequential algorithm, which resembles a multi-seeded, obstacle-aware, 3D voxel-based flood fill with a Breadth-First Search (BFS) frontier-based approach. We show that algorithm exhibits linear scaling in both time and space with respect to the number of voxels explored. Physics overlap queries dominate the runtime, introducing a significant constant factor in scenes with complex geometry or dense collider configurations.

Notation

We introduce the following notation for clarity:

- W: Total work.
- S: Total space.
- N: Total number of processed voxels during execution.
- A: Number of anchors.
- D: Number of delimiters.
- K: Cost of a physics overlap query.
- F: Maximum size of the frontier queue.

5.3.1 Time Complexity

The algorithm consists of two stages: initialization and frontier-based expansion, including per-voxel exploration.

Initialization: Initialization is primarily composed of constant-time operations, with the exception of anchor seeding, which requires $W_{\text{init}} = \Theta(A + D)$ time as each of the A anchors and D delimiters are processed individually.

Exploration: The core of the algorithm is a loop that processes the frontier queue until it is empty. In each iteration, a voxel is dequeued, delimiter checks are performed via a physics overlap query and usually six neighbors are added to the frontier. Valid neighbors are then added to the frontier. Since each voxel is visited at maximum six times, the number of iterations is proportional to N, and the total cost of frontier queue operations (dequeueing, enqueueing, and membership checks using hash-based structures) is $\Theta(N)$.

Each voxel incurs a constant overhead for bookkeeping operations such as checking anchor associations and updating sets or dictionaries. The physics overlap query contributes a cost of K for each of the N voxels. Enqueuing up to six neighbors adds another constant overhead. Therefore, the cumulative cost of exploration is $W_{\text{expl}} = \Theta(N \cdot K))$.

Total Complexity: Combining all components, the overall runtime is given by:

$$W = W_{\text{init}} + W_{\text{expl}} = \Theta(A + D) + \Theta(N \cdot K) = \Theta(A + D + N \cdot K).$$

In most practical scenarios where the number of anchors and delimiters is small compared to the number of voxels explored (i.e., $N \gg A + D$, N), this simplifies to:

$$W = \mathcal{O}(N \cdot K)$$

5.3.2 Space Complexity

The algorithm maintains several auxiliary data structures throughout execution. The frontier is stored using both a queue and a hash set to allow for constant-time membership checks, collectively requiring up to $O(F)$ space, where $F \leq N$. A separate hash set tracks all explored voxels and grows linearly with N.

A dictionary stores per-voxel metadata, also contributing $O(N)$ to the total space. Additionally, the algorithm may track delimiter relationships between seed anchors, which in the worst case involves $O(A^2)$ anchor pairs. Finally, a reusable object pool is employed for voxel instances or data buffers, requiring $O(N)$ space.

Total Complexity: Taking all components into account, the total auxiliary memory usage is:

$$S = \mathcal{O}(N + A^2).$$

Again, assuming $N \gg A^2$, this simplifies to:

$$S = \mathcal{O}(N).$$

5.3.3 Performance Optimizations

Efficient execution is critical for the practical viability of the algorithm, particularly when operating on large and complex worlds. This section outlines the optimizations already incorporated into the implementation and identifies opportunities for further performance gains.

Implemented Optimizations

The implementation includes several measures aimed at reducing runtime and memory overhead.

First, an additional hash set is maintained alongside the frontier queue to enable constant-time membership checks, avoiding costly linear searches.

Second, object pooling is applied to `VoxelData` structs with the intention of reducing garbage collection overhead. The underlying idea is that once a voxel has been visited from all six directions, it will not be revisited and can be returned to a pool for reuse. However, object pooling may be of limited value here. Since `VoxelData` is a struct with a fixed size of 38 bytes, it typically resides on the stack rather than the heap, meaning it is not subject to garbage collection in most cases. As a result, the additional complexity introduced by pooling could offset its intended performance improvements.

Moreover, the algorithm operates entirely in a single-threaded context, leaving modern multi-core CPUs—ranging from 2 to 8 cores in typical consumer systems [35]— underutilized.

Potential Future Improvements

Several ways exist for improving performance beyond the current state. One possibility is to introduce parallelization, enabling the algorithm to utilize multiple CPU cores rather than operating in a single-threaded context. This change could significantly increase throughput, particularly on modern consumer systems typically with 2 to 8 cores depending on the market segment [35].

Another potential improvement involves adopting performance-oriented execution frameworks. Moving from standard C# to technologies such as the C# Job System and Burst

(see section 2.7) could reduce overhead by bypassing some of the runtime's safety mechanisms in favor of more direct, optimized code execution.

Finally, memory usage and caching behavior present scalability challenges. In large and complex worlds, maintaining extensive hash sets that store all voxels becomes unsustainable. Such structures consume considerable amounts of memory and often lead to poor cache performance due to random memory access patterns, as discussed in Section 2.4.

5.3.4 Suitability and Limitations

The reference algorithm demonstrates good performance characteristics. The core frontier-based BFS flood-fill approach is both conceptually simple and inherently well-suited for voxel-based spatial segmentation tasks. Theoretical performance analysis confirms this, showing that runtime and memory usage scale linearly with the number of visited voxels.

Beyond its simplicity, the algorithm exhibits several advantageous properties. It utilizes a frontier exploration queue that implicitly defines a total order over voxel exploration, eliminating the need for auxiliary data structures to track distances. Consequently, the first time a voxel is reached via an anchor, the associated path is guaranteed to be the shortest possible. This design eliminates the need to store or update explicit distance values.

Furthermore, voxel ownership is resolved deterministically and irrevocably: once a voxel is assigned to an anchor, that decision is final. This mechanism prevents ownership conflicts and simplifies the implementation by removing the need for conflict resolution, sorting, or post-processing steps during the exploration phase.

However, the algorithm exhibits scalability limitations. As the world size increases or the voxel resolution is decreased, the total number of voxels—and therefore the computational and memory requirements—grows cubically. This poses challenges for high-resolution or large-scale environments, particularly when system memory is constrained.

5.4 Runtime Queries

Once the SFS creation algorithm completes, the resulting data structure enables efficient runtime queries. In particular, it allows to determine whether the transform's current position lies within the subspace of any anchor, and if so, identify which one.

To balance memory efficiency and runtime performance, only the borders of anchor volumes are stored. While this minimizes memory usage, it requires additional computation during queries. The overall procedure is visually illustrated in Figure 5.3 and presented in Algorithm 2.

Figure 5.3 illustrates two runtime queries. Query locations are shown as black circles. Raycasts are drawn as black lines, with rounded squares marking the hit points where they intersect borders. Anchor volumes are lightly shaded, and borders are highlighted.

For the right query, located inside the blue volume, the rays traverse in opposite directions (up and down) and both hit blue border voxels, so the result is Blue. For the left query, which lies outside any volume, the downward ray hits the Yellow border, but the upward ray continues until its maximum distance without finding a volume. This leads

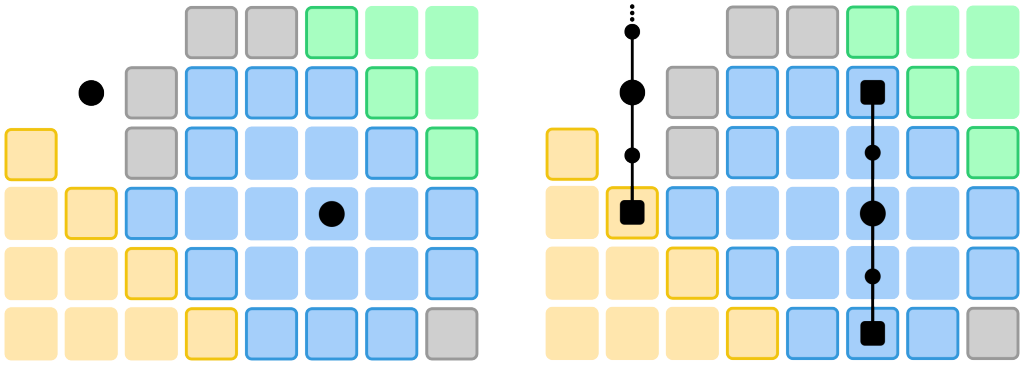


Figure 5.3: Location detection queries. Left: original query locations. Right: raycasts with hit points on borders and resulting anchor assignments.

to many hash set membership checks before concluding that no anchor was found.

Algorithm 2: Determine the current anchor of a transform using raycasting and neighborhood analysis

```
1  Function DetermineAnchor(transform):
2      pos ← transform.pos
3      upAnchor ← RaycastFirstAnchor(pos, up)
4      downAnchor ← RaycastFirstAnchor(pos, down)
5      if upAnchor = downAnchor and upAnchor ≠ null then
6          lastAnchors[transform] ← upAnchor
7          return upAnchor
8      anchorCandidates ← SphereCast(pos)
9      if lastAnchors[transform] exists and
        anchorCandidates.Contains(lastAnchors[transform]) then
10         return lastAnchors[transform]
11     lastAnchors[transform] ← null
12     return null

13 Function RaycastFirstAnchor(pos, dir):
14     voxelPos ← DetermineVoxelGridPosition(pos)
15     for i ← 0 to max_length do
16         targetPos ← voxelPos + dir · i
17         if borderDict.TryGetValue(targetPos, anchorId) then
18             return anchorDict[anchorId]
19     return null
```

The core of the query process is the `DetermineAnchor` function. It begins by casting rays in opposite directions (commonly `up` and `down`) from the transform's position using the `RaycastFirstAnchor` helper. If both raycasts intersect the same anchor, it is assumed that the transform lies within that anchor. The anchor is then cached for the transform and returned.

However, if the raycasts disagree or fail to hit any anchor, the transform is assumed to be in free space. In this case, a voxelized sphere cast is performed to identify nearby anchors. If the previously cached anchor for the transform is still within the neighborhood

returned by the sphere cast, it is considered a valid approximation, and the cached anchor is returned. This mechanism introduces a tolerance for transient gaps due to voxel underestimation, allowing a transform to move slightly outside the strict voxelized boundary of an anchor before it is considered to have exited.

If no matching anchor is found, the cached entry is cleared and `null` is returned.

The raycasting helper `RaycastFirstAnchor` works by converting the world position into voxel space and iteratively probing along a direction. At each voxel step, it checks the `borderDict` to see if an anchor is associated with that voxel. If an anchor is found, it is immediately returned; otherwise, the search continues until a maximum range is exceeded.

5.4.1 Time Complexity

To analyze the time complexity of runtime queries, we define the following parameters:

- D: The maximum world-space diameter of any anchor volume (i.e., the largest distance between opposing borders within a single anchor).
- v: The voxel size.
- D_{max}: The maximum raycast probing distance in world units.

The `DetermineAnchor` function (see Algorithm 2) performs two primary operations. First, it executes two directional raycasts (`RaycastFirstAnchor`) originating from the transform's position and extending in opposite directions (e.g., up and down) to detect the nearest anchor boundary. Second, as a fallback, it performs a sphere cast that scans the transform's immediate 1-voxel neighborhood for any previously known anchors.

The 1-voxel neighborhood used in the sphere cast contains a small, fixed number of surrounding voxels. Therefore, its computational effort is constant and independent of the input scene or voxel resolution. We treat this as a constant-time operation and exclude it from the asymptotic analysis.

Worst-Case Scenario: In the worst case, the transform lies outside all anchors, and both raycasts reach their maximum probing distance without finding any anchor. Each raycast performs at most D_{max}/v voxel steps, for a total of $\mathcal{O}(2D_{\text{max}}/v) = \mathcal{O}(D_{\text{max}}/v)$ operations.

Typical-Case Scenario: In practice, the majority of transforms are located inside anchor volumes. In this case, raycasts typically hit a boundary after only traversing up to the diameter D of the enclosing anchor. This results in an average-case complexity of:

$$\mathcal{O}\left(\frac{D}{v}\right)$$

The runtime complexity of the query scales linearly with the ratio between the anchor size (or probing distance) and the voxel size. Since the fallback sphere cast is constant-time and most queries terminate early due to hits within nearby anchors, the typical performance is highly efficient and well-suited for real-time applications.

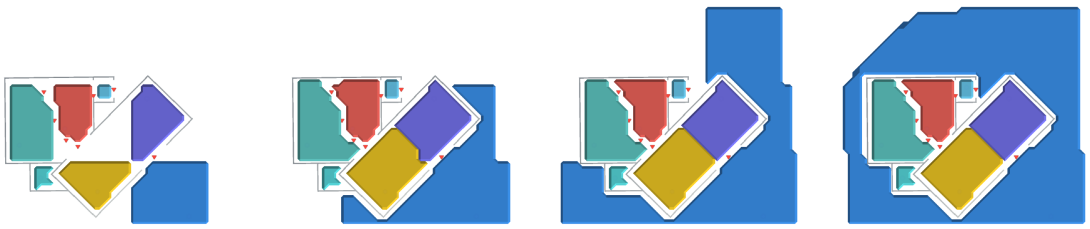
6 Chunked Parallel Algorithm

The central concept of the algorithm developed in this thesis is to divide the entire space into discrete *chunks*, which can be processed independently. This partitioning improves memory efficiency by increasing data locality, and it also simplifies the implementation of parallel execution. For a detailed background on the concept of chunking and its applications in computer science and game development, see Section 2.2.

This section provides a detailed explanation of the chunked algorithm. It begins by outlining a high-level overview. The core logic of the algorithm is then presented through pseudocode. Afterward, we analyze its runtime and memory usage. Implementation-specific considerations are discussed next, including parallelization strategies, burst compilability, and performance-oriented design choices. The section concludes by examining limitations and edge cases, particularly situations where the algorithm underperforms compared to its sequential version.

6.1 Algorithm Overview

This section presents a high-level overview of the chunked parallel algorithm. Due to its increased complexity compared to the sequential version, the explanation is divided into two parts: the current high-level overview and a detailed description provided in Section 6.3.



1. Iteration 2. Iteration 3. Iteration 5. Iteration

Figure 6.1: Selected iterations of the chunked parallel algorithm on the *Villa* scene.

Figure 6.1 shows selected iterations of the chunked parallel algorithm applied to the *Villa* scene. From left to right: the initial flood fill is followed by successive steps that propagate into neighboring chunks. Further details are provided in the algorithm description below.

Algorithm 3: Chunk-based voxel flood fill

```
1  Function ChunkedVoxelFloodFill(spaceFoundation):
2      anchors ← InitializeAnchors()
3      (dirtyChunks, chunkDict) ← CreateChunks(anchors, spaceFoundation)
4      ScheduleFloodFill(dirtyChunks)
5      while true do
6          HandleCutoffs(chunkDict)
7          newDirtyChunks ← SwapBorders(dirtyChunks, chunkDict)
8          if newDirtyChunks = ∅ then
9              break
10
11         dirtyChunks ← newDirtyChunks
12         ScheduleFloodFill(dirtyChunks)
13     PostProcess(chunkDict, spaceFoundation)
```

Similar to the sequential version, the algorithm begins by collecting and processing all anchors in the scene. Each anchor is assigned a unique identifier and mapped to a chunk based on its position in world space. The resulting chunks are inserted into the `dirtyChunks` hash set, which tracks chunks requiring processing each iteration.

For each chunk, a localized version of the sequential flood-fill algorithm is executed in parallel. This intra-chunk flood fill behaves similar to the sequential algorithm, but handles a lot more special cases and exceptions such as dealing with chunk boundaries. When the flood fill reaches the border of a chunk and would propagate beyond, the corresponding voxels are collected. Each chunk maintains six such borders—one for each face of the cube.

After this initial step, the algorithm enters its main iteration loop:

1. **HandleCutoffs:** For each chunk, voxels at the borders that have become disconnected (cut off) are identified and removed. This includes resolving the disconnected region and subsequently refilling it through a localized flood fill from valid border voxels. More details are provided in Section 6.1.1.
2. **SwapBorders:** For every chunk that was modified in the last iteration (marked as "dirty"), its six borders are examined. If a border contains voxel data, that data is transferred to the corresponding neighboring chunk by swapping both borders (e.g., left-to-right, top-to-bottom). If the neighboring chunk doesn't yet exist, it is created during this step. All affected neighboring chunks are then marked as dirty for the next iteration.
3. **ScheduleFloodFill:** All chunks marked as dirty are processed concurrently using the flood-fill algorithm to propagate voxel data appropriately. This essentially corresponds to the sequential algorithm, but locally for each chunk. See Section 6.3.1 for further details.

This loop continues until no dirty chunks remain at the end of an iteration.

PostProcess: In the final step, relevant data is collected and stored in ScriptableObjects. This includes explicit and implicit delimiters, anchors, volume information within each chunk, and temporary data used for marching cubes-based visualization. The stored data enables efficient runtime queries within the Space Foundation System.

6.1.1 Comparison to the Sequential Version

While the chunked parallel algorithm shares some foundational concepts with the sequential version, it introduces significant differences in structure and complexity due to its parallel nature.

Shared Structure – Setup and Execution. Both algorithms begin with a setup phase. In the parallel version, this phase includes additional work to initialize and maintain individual chunk structures. After setup, the sequential version proceeds through a single execution phase, during which all major operations are performed: flood-filling, collection of metadata like border voxels, explicit and implicit delimiters. The flood filling of the chunked parallel version is very similar to the sequential version, with the difference that no metadata is collected during execution. Apart from that, however, both have a similar core.

Simplicity of the Sequential Approach. The sequential version benefits from a strict, linear execution order. Once a voxel is assigned to an anchor, its state remains unchanged throughout the algorithm. This immutability simplifies reasoning about the algorithm's correctness and allows metadata to be collected in a single pass.

Complexity in the Parallel Version. In contrast, the parallel algorithm cannot maintain this strict execution order. Chunks are processed independently, and voxel ownership may change over time. For instance, a voxel initially claimed by one anchor may later be overwritten if a closer anchor from a neighboring chunk reaches it during a subsequent iteration. These overwrites introduce additional complexity and can result in disconnected voxel regions—referred to as *cutoffs*—which must be addressed explicitly. More on that in the following Section 6.1.1.

Due to the potential for overwrites during flooding, metadata such as delimiters and anchors cannot be considered final during the flood-fill phase. To manage this uncertainty and avoid inconsistencies, the algorithm defers metadata extraction to a dedicated post-processing step.

Another fundamental difference is the use of chunking. On the one hand, chunking greatly improves scalability by dividing arbitrarily large problems into fixed-size, manageable units. This regular structure also enables architectural simplifications and leads to desirable properties, which are discussed further in Section 7.3.1.

On the other hand, chunking introduces significant complexity, particularly at chunk boundaries. For instance, flood fill operations may be interrupted at chunk borders and must be resumed in subsequent iterations. Additionally, auxiliary data structures are required to track the origin of flood-filled voxels within each chunk, in order to prevent backfilling into the source neighbor.

6.2 Detecting and Dealing with Cutoffs

As mentioned previously, unlike the sequential algorithm, the chunked parallel has a more relaxed exploration order with deferred consistency. As a result, voxels may over-

write each other if a newly flooded voxel has a shorter distance to its anchor than the existing voxel. This seemingly simple behavior has a significant impact on the exploration process and can lead to edge cases—such as entire regions becoming disconnected from their original anchor. In this section, we examine how these cutoffs occur, how to detect them, and how to resolve them.

6.2.1 Relaxation of the Exploration Order

Before discussing the algorithm in detail, it is useful to clarify why the exploration order is relaxed in the first place. In order to enable parallel computing, the algorithm must be designed so that computational work can be divided into smaller, independent pieces. If we maintained strict dependencies between chunks, we would require synchronization mechanisms such as mutexes to prevent race conditions. These synchronization methods impose a significant overhead. Furthermore, strict ordering would require sharing each chunk's data structures across threads, which would also need to be synchronized, adding further cost.

The underlying cause of these dependencies lies in the relationships between anchors. In practice, anchors are not independent. Their growth is limited by the shortest distance to neighboring anchors, which constrains the extent of their respective flood fills. This constraint generally holds unless anchors fall within each other's reach without being separated by the spatial boundaries of the scene.

In the chunked parallel algorithm, the flooding of each chunk is treated as an entirely local process. Any resulting inconsistencies at chunk boundaries are addressed later in a separate stage. During the main computation, cross-chunk dependencies are intentionally ignored in order to maximize parallelism, and any conflicts are resolved afterwards. In typical spatial configurations, the cost of this deferred resolution is small compared to the performance gains achieved by avoiding synchronization in the main processing phase.

6.2.2 Overwrite Occurrences

Figure 6.2 illustrates a simplified scenario showing how inter-chunk borders are resolved and under what conditions voxel overwrites occur. Three stages of exploration are shown.

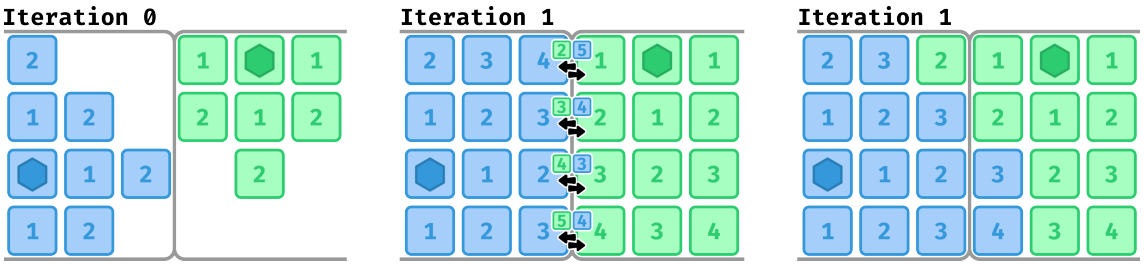


Figure 6.2: Inter-chunk border resolution and voxel overwrites. Left: initial fill. Middle: border voxels exchanged and neighbor chunks flood-filled. Right: final result after parallel fills and overwrites, where smaller distances (or, in ties, smaller anchor IDs) take priority.

On the left (iteration 0) there is an intermediate step of the initial chunk filling. In the

middle, after the first flood fill, the left chunk is filled by Blue and the right chunk by Green. At the shared border, voxels from both sides are exchanged and placed into the neighboring chunk, denoted with small voxels and accordingly updated distances. During the main loop, these swapped border voxels are processed and local chunk flood fills are executed in parallel.

Overwrites occur when a new voxel provides a shorter distance than the existing one. In case of equal distances, the voxel from the anchor with the smaller ID is preferred, ensuring a deterministic outcome. The right image shows the final result after this process.

6.2.3 Occurrences of Cutoff Voxels

Allowing voxels to be overwritten can lead to situations where previously flooded regions become disconnected from their corresponding anchor. More precisely, there may no longer be a continuous path consisting solely of voxels belonging to the same anchor. Under certain configurations, this disconnection can cause entire volumes to be cut off, which may block further exploration or persist without justification. Such artifacts arise as a byproduct of the weaker exploration order.

Figure 6.3 shows a configuration where a cutoff occurs. On the left, an intermediate step of the first flood fill iteration is shown, with Blue and Purple filling their respective chunks. In the middle, the completed first iteration is shown with only two selected border voxel transitions (others omitted for clarity). The Blue voxel with distance 4 at the border initiated a flood into the right chunk but is later overwritten by Purple, splitting the Blue volume into two. Without conflict resolution, the right image shows the resulting disconnected Blue regions.

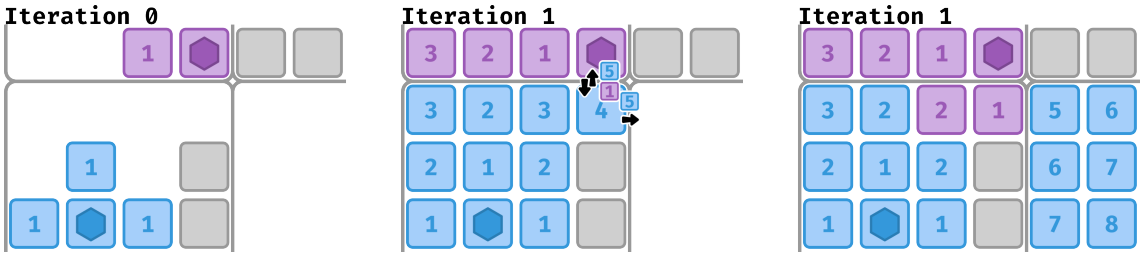


Figure 6.3: Cutoff configuration. Left: intermediate step of the first flood fill. Middle: completed iteration with border voxels, where Blue is overwritten by Purple, splitting the Blue volume. Right: final result without conflict resolution, leaving disconnected Blue regions.

6.2.4 Detecting Cutoff Voxels

To detect parts of a volume being disconnected from their originating anchor, we examine the connectivity of each voxel with respect to its anchor's main volume.

A voxel is considered cut off if there exists no path from it back to the anchor's origin, using only voxels that belong to the same anchor and have strictly lower distance values.

In practice, we inspect all six neighbors of a given voxel and search for one that satisfies both criteria. If no such neighbor exists, the voxel is classified as a cutoff voxel.

Detection Workflow. The detection of cutoff voxels occurs after the flood fill for a chunk is completed. At this stage, each voxel is individually examined using the above criteria. We distinguish between two types of voxels:

- **Non-border voxels:** These voxels lie entirely within the chunk. If any are found to be cutoff, they are immediately resolved (see Section 6.2.5 for the resolution algorithm).
- **Border voxels:** These lie on the chunk boundary. Since some of their neighbors may be outside the current chunk, full connectivity information may be missing. For example, such a voxel might only have 5, 4, or even 3 neighbors within the chunk. If no safe determination can be made based solely on the available neighbors, the voxel is added to a list of *potentially cutoff* voxels.

The `potentiallyCutoffVoxels` list is deferred for processing during the next iteration of the main algorithm, as detailed in Section 6.3.1.

6.2.5 Removal of Cutoff Sections

Once voxels are detected as cutoff candidates, they are either resolved immediately (for non-border voxels) or in a subsequent iteration (for border voxels that might still be connected across chunk boundaries). Regardless of timing, the actual removal is always performed by the function `RemoveCutoffVoxels`, which we examine in detail below in Algorithm 4. Before diving into the implementation, however, we present a high-level example in Figure 6.4 to illustrate the concept visually.

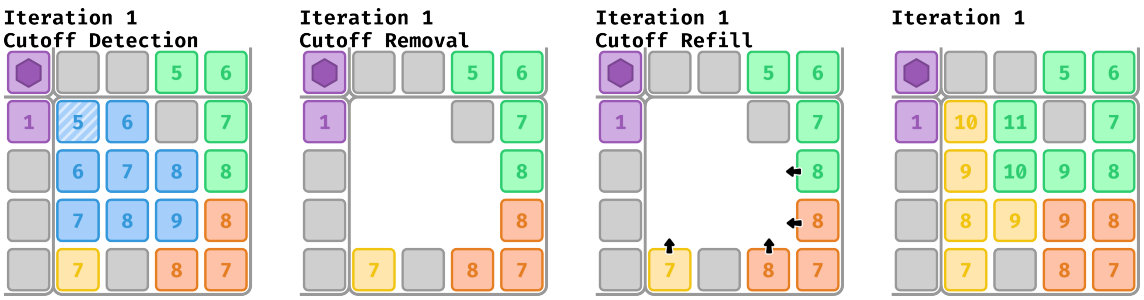


Figure 6.4: Cutoff removal process in four stages. From left to right: detection of a Blue cutoff voxel, removal of its volume via BFS, refilling of the affected chunk (with flooding voxels indicated by arrows), and the intermediate result after refilling.

Figure 6.4 illustrates four stages of the removal process, continuing from the cutoff example in Figure 6.3.

The first image shows the state after the second flood fill (iteration 1), where the Blue voxel with distance 5 is identified as a cutoff voxel because it lacks any Blue neighbor with a smaller distance. The second image shows the removal of the cutoff volume using BFS from the cutoff voxel, deleting all Blue voxels with greater distance in that volume. Since removing happens within a chunk, borders and other chunks are not affected by this. The third image depicts the refilling of the deleted chunks, where again only voxels within the affected chunk are considered, and flood propagation is indicated by arrows. The fourth image shows the state after refilling, though not final, since Purple continues to flood into the chunk in the next iteration.

Algorithm 4: Remove cutoff voxels and enqueue border voxels for exploration

```
1  Function RemoveCutoffVoxels(startIdx, cutoffVoxel):
2      Q ← {startIdx}
3      while ¬ Q.IsEmpty() do
4          idx ← Q.Dequeue()
5          voxel ← voxelArray[idx]
6          if voxel.anchorId ≠ cutoffVoxel.anchorId or
            voxel.distance < cutoffVoxel.distance then
7              continue
8          voxelArray[idx] = ∅
9          cutoffRemovedSet.add(idx)
10         SetVisited (idx, false)
11         foreach neighbor of idx do
12             if neighbor is within chunk then
13                 queue.enqueue(neighbor)
14     if cutoffRemovedSet = ∅ then
15         return
16     borderVoxels ← ∅
17     foreach idx ∈ cutoffRemovedSet do
18         foreach neighborVoxel of idx do
19             if neighborVoxel is valid and idx is in neighborVoxel's Anchor range then
20                 nv.distance ← nv.distance + 1
21                 borderVoxels.add({idx, nv})
22     StartExploration(Q1, Q2, borderVoxels)
```

The actual algorithm proceeds in three main steps:

1. **Cutoff Volume Removal:** Starting from the cutoff voxel index, we initialize a queue Q with this voxel. While the queue is not empty, we dequeue a voxel and verify that its anchor matches the cutoff voxel's anchor and its distance is greater than or equal to that of the cutoff voxel.

   These checks ensure that we only remove the isolated (invalid) region and avoid deleting shared paths or valid segments. Each valid voxel is reset (emptied), and its index is added to a set called `cutoffRemovedSet`. Then, we enqueue all valid neighbors (within the same chunk) to continue the removal process.

2. **Border Voxel Collection:** After the cutoff volume has been deleted, we loop through each index in `cutoffRemovedSet`. For each, we examine its neighbors and add them to a list `borderVoxels` if they are within the same chunk, are not blocked by a delimiter, are non-empty and would still be within their anchor's maximum distance from their origin.

3. **Flood Fill from Border Voxels:** Finally, the region is refilled by invoking `StartExploration` using the `borderVoxels` list. This process is equivalent to the normal flood-fill routine described in Section 6.3.2.

6.3 Detailed Algorithm Breakdown

In Section 6.1, we presented a high-level overview of the chunked parallel algorithm. This section provides a more in-depth analysis of its core components and execution logic. We focus on the flood-filling procedure within individual chunks and the exploration function at the heart of the algorithm, including its divergence from the sequential counterpart.

6.3.1 FloodFillJob Execution

The execution of the flood fill job is initiated by the `ScheduleFloodFillJobs` function, as shown in the high-level algorithm (see Algorithm 3). This function creates a job for each dirty chunk in the current iteration and submits them to the C# Job System Scheduler. Once scheduled, these jobs are picked up and processed across the following frames. A synchronization barrier ensures that `ScheduleFloodFillJobs` waits for all jobs to complete before continuing execution.

The main logic of each flood fill job resides in the `ExecuteChunkedFloodFillJob` method, which is executed when the job begins. Algorithm 5 details this process.

Algorithm 5: Chunked flood fill execution job

```
1 Function ExecuteChunkedFloodFillJob():
2     cutoffVoxels ← CollectAndSortBorderCutoffVoxels()
3     foreach voxel ∈ cutoffVoxels do
4         RemoveCutoffVoxel(voxel)
5     if cutoffVoxels = ∅ then
6         initialVoxels ← GetInitialExplorationVoxels()
7         StartExploration(initialVoxels)
8         DetectAndResolveCutoffVoxels()
```

There are two distinct contexts in which the flood fill job can be executed:

1. Invocation from `HandleCutoffBorders` This context arises in the first part of the main algorithm's loop, where the `HandleCutoffBorders` function processes voxels that were previously detected as possibly cut off, based on the limited information available within their chunk.

To verify if they are truly cut off, they are reconsidered in a later iteration after incorporating information about neighboring voxels outside their chunk. If they are still disconnected, they are added to a list of cutoff borders. Next, this list is sorted by distance and voxel ID. Then, each cutoff voxel is removed iteratively using the `RemoveCutoffVoxel` function (detailed in Algorithm 4).

2. Standard Invocation In this more common context, there are no pre-existing cutoff voxels. Instead, the algorithm collects all initial voxels, which consist of either the origin voxels of anchors in the very first iteration or all border voxels received from neighboring chunks via flooding.

These voxels are sorted by distance and anchor ID. The flood fill process then starts with the `StartExploration` method (see Section 6.3.2).

After the flood fill completes, cutoff voxels are identified and handled. Non-border voxels that are cut off are immediately removed using the `RemoveCutoffVoxel` function (see Algorithm 4). Border voxels that might be cutoff based on limited intra-chunk information are added to the list of possibly cut off voxels.

6.3.2 StartExploration

The `StartExploration` procedure, illustrated in Algorithm 6, manages the flood-fill process. It provides the structural wrapper for the `ExploreVoxel` function and is executed within the `FloodFillJob` in various contexts, depending on the current state of the algorithm.

Algorithm 6: Two-queue breadth-first exploration with sorted frontier

```
1  Function StartExploration(Q1, Q2, initialVoxels):
2      head ← 0
3      currQ ← Q1
4      nextQ ← Q2
5      Sort(initialVoxels)
6      EnqueueSameDist(currQ, head, initialVoxels)
7      while ¬ currQ.IsEmpty() do
8          while ¬ currQ.IsEmpty() do
9              ExploreVoxel(currQ.Dequeue(), nextQ)
10         d ← initialVoxels[head].voxel.distance
11         while has remaining voxels for distance d or nextQ not empty do
12             v ← EnqueueNextOrderedVoxel(initialVoxels, head, nextQ)
13             currQ.Enqueue(v)
```

The function takes two empty queues and a list named `initialVoxels`. During initialization, this list is sorted. Although sorting introduces additional overhead, it enables exploration to proceed in strict local order within the chunk, thereby mimicking the behavior of the sequential version of the algorithm.

The two queues act as a workaround to maintain voxel ordering. Ideally, a priority queue would be used for this purpose; however, the C# Job System does not support a compatible priority queue implementation (see Section 2.7.3).

The current queue is initialized using the `EnqueueSameDist` function. This function begins at the head index—initially set to 0—and iteratively enqueues voxels from the `initialVoxels` list as long as their distance matches that of the original head voxel. After each insertion, the head index is incremented. This process continues until a voxel with a greater distance is encountered or the end of the list is reached. As a result, the current queue is populated with all initial voxels sharing the smallest distance, establishing the foundation for the first flood-fill iteration.

The main loop proceeds by dequeuing and exploring each voxel in the current queue using the `ExploreVoxel` function. This function receives the next queue as a parameter, which it uses to enqueue neighboring voxels. It is guaranteed that the voxels enqueued

in the next queue all have a distance exactly one greater than those in the current queue.

Once the current queue is empty, the algorithm refills it with the next batch of voxels. This batch is assembled from two sources: the `nextQueue` and any remaining `initialVoxels` with the respective distance. Voxels from both sources are merged in order, based on their anchor IDs, to preserve sorting. Only `initialVoxels` that match the expected distance are considered, ensuring the same-distance invariant for each queue.

The loop continues until the current queue cannot be refilled, at which point the flood-fill process is complete.

6.3.3 ExploreVoxel

This section provides a detailed explanation of the `ExploreVoxel` function, the core component of the flood-fill algorithm. While it closely resembles the sequential implementation, it incorporates several key differences. The function is illustrated in Algorithm 7.

Algorithm 7: Explores a voxel and enqueues valid neighbors

```
1  Function ExploreVoxel(idx, fromVoxel, Q):
2      voxel ← voxelArray[idx]
3      if IsVisited(idx) or OverlapsWithDelimiter(voxel) or
         IsOutOfAnchorRange(idx, fromVoxel) then
4          return
5      SetVisited(idx)
6      if voxel < fromVoxel then
7          return
8      voxelArray[idx] ← fromVoxel
9      pos ← ReverseIdx(idx)
10     neighbor ← fromVoxel with distance + 1
11     foreach dir ∈ {±x, ±y, ±z} do
12         neighborIdx ← Idx(pos + dir)
13         if neighbor is inside chunk then
14             EnqueueOrdered(neighborIdx, neighbor)
15         else if neighbor crosses border then
16             if neighbor not entering from same border direction then
17                 AddToBorder(dir, neighbor)
```

`ExploreVoxel` takes three parameters: an index `idx`, representing a position within the chunk, `fromVoxel`, the voxel used to propagate the flood fill, and `Q`, a queue to which the neighbors are appended. The function begins by retrieving the current voxel at `idx`. It immediately returns if any of the following conditions is met: the voxel has already been explored in the current iteration, the voxel overlaps with a delimiter, or the propagation from `fromVoxel` would exceed its anchor's `maxDistance`.

Additionally, if the current voxel has a smaller distance than `fromVoxel`, or if the distances are equal but the current voxel's anchor ID is smaller, no overwrite is possible and the function exits early.

If none of the early-exit conditions apply, the function overwrites the voxel at `idx` with `fromVoxel`, increments its distance by one, and then iterates over its neighbors. For each neighbor, it determines whether the voxel lies within the same chunk or crosses into a neighboring chunk. Neighbors within the same chunk are added directly to the queue for further exploration, while border-crossing neighbors are placed into the appropriate border hash map. This placement triggers a border swap in the next algorithm iteration, ensuring correct propagation across chunk boundaries.

The most notable distinction from the sequential version of `Explore` is that no metadata is collected during this process. Since voxel values may be overwritten by better candidates in subsequent iterations, any intermediate metadata would not be reliable and add additional effort to maintain. Moreover, handling of neighboring voxels is slightly more complex, as the function must account for whether a neighbor lies within the current chunk or on its boundary.

6.4 Complexity Analysis

This section evaluates the complexity of the proposed algorithm from both theoretical and practical perspectives. Using the *Work–Span model* [59, 6], we characterize the algorithm's parallel execution and compare it to a sequential baseline. We derive terms for the time and space complexity of the chunked parallel algorithm and discuss worst-case and realistic inputs and their implication on performance.

6.4.1 Notation

We introduce the following notation for clarity:

- N: Total number of processed voxels during execution.
- A: Number of anchors.
- D: Number of delimiters.
- C: Number of chunks.
- s: Size of a chunk in one dimension.
- n: Number of voxels per chunk s^3.
- k: Cost to initialize a chunk (dominated by per-voxel physics overlap checks, run n times per chunk).
- I: Number of main loop iterations.

We denote:

- W: Total work under the Work–Span model.
- S: Critical path length (span).

6.4.2 The Work–Span Model

The work–span model (also called work-time or work-depth), originally introduced in [59], provides a framework for analyzing parallel algorithms in terms of total work and

critical-path span. Blelloch [6] later presented the model in a programming-oriented context, making it accessible and widely adopted for algorithm design and teaching. The framework analyzes parallel algorithms by separating computation into two components:

- **Work** W: The total number of unit operations executed if the algorithm is run on a single processor. This corresponds to the sequential cost.
- **Span** S: The longest chain of dependent operations, representing the theoretical minimum time with infinite processors.

According to Brent's Theorem [8], the time on P processors is bounded by:

$$T_P = \mathcal{O}\left(\frac{W}{P} + S\right).$$

This decomposition allows us to reason about both scalability and inherent sequential bottlenecks. In the following subsections, we derive W and S for each phase of the algorithm and discuss best-case and worst-case scenarios.

6.4.3 Exploration

The `StartExploration` procedure performs a level-order (BFS-style) propagation from a given list of `initialVoxels`, seeded either by initial anchors, a border flooding from adjacent chunks, or boundary of a previously removed cutoff volume. The function proceeds in distance-sorted order, expanding each voxel's neighbors and pushing new candidates into the next frontier. Since this happens inside of the FloodFillJob, we only take a look at the work, not the span as the job itself runs sequentially.

Sorting Initial Voxels At the start of the function, the `initialVoxels` list is sorted by distance. In the worst case, this list consists of all voxels on the chunk's surface. For a chunk of size $n = s^3$, the surface voxel count is $6 \cdot s^2 = 6 \cdot n^{2/3}$. Hence, the sort operation with a complexity of $\mathcal{O}(n \cdot \log n)$ costs:

$$W_{\text{sort}} = \mathcal{O}(n^{2/3} \cdot \log n^{2/3}) = \mathcal{O}(n^{2/3} \cdot \log n)$$

This cost is relatively minor compared to the full exploration and is incurred only once per chunk per job.

Flood Fill and Border Handling Let n_i denote the number of voxels visited in a chunk during this local exploration. In the worst case, $n_i = n$, i.e., the entire chunk is filled. For each visited voxel, the Explore function marks the voxel as visited, performs collision and visited checks, writes voxel data and enqueues up to 6 neighbors (or forwards them to borders). The collisions checks are mere lookups in a table which was created upon chunk creation. Each of these steps is $\mathcal{O}(1)$, so the work is linear in the number of visited voxels:

$$W_{\text{flood}} = O(n_i)$$

Total Complexity Let n_i be the number of visited voxels in the chunk.

$$W_{\text{explore}} = W_{\text{sort}} + W_{\text{flood}} = \mathcal{O}(n^{2/3} \log n) + \mathcal{O}(n_i) = \mathcal{O}(n^{2/3} \log n + n_i)$$

To simplify the bound, observe that:

$$\lim_{n\to\infty} \frac{n^{2/3} \log n}{n} = \lim_{n\to\infty} n^{-1/3} \log n = 0$$

Since $\log n$ grows slower than any positive power of n:

$$\lim_{n\to\infty} \frac{\log n}{n^{\varepsilon}} = 0 \quad \text{for any } \varepsilon > 0$$

That implies:

$$\mathcal{O}(n^{2/3} \log n) = \mathcal{O}(n)$$

Therefore, we can derive an upper bound for the total work:

$$W_{\text{explore}} = \mathcal{O}(n + n_i) = \mathcal{O}(n)$$

6.4.4 Cutoff Voxel Removal

The `RemoveCutoffVoxels` function handles the deletion of cutoff voxel volumes and refilling with surrounding voxels in a `StartExploration` pass. This happens during cutoff resolution when inconsistencies are detected between adjacent chunks. The steps are:

1. Remove all voxels that belong to the same anchor as the cutoff voxel and have a distance $\geq$ the cutoff voxel's distance.
2. Collect all neighbor voxels of the affected area.
3. Run `StartExploration` from the neighbor voxels to re-flood the chunk region.

We now estimate the work of each of these steps.

1. Cutoff Region Removal (BFS) Let r be the number of voxels removed in this process. In the worst case, this could be the entire chunk, so $r \leq n$.

2. Collection of neighbors After removal, each voxel in the removed set is checked for neighboring candidates to serve as new propagation seeds.

In the worst case, about half of our voxels are border voxels resulting in $\lceil \frac{n}{2} \rceil$ processed borders.

3. Re-Propagation via StartExploration The function calls `StartExploration` using the new set of border voxels. From prior analysis we know $W_{\text{explore}} = \mathcal{O}(n)$.

Total Work Combining all components, we get:

$$\begin{aligned} W_{\text{cutoff}} &= W_{\text{remove}} + W_{\text{neighbor-collection}} + W_{\text{explore}} \\ &= \mathcal{O}(n) + \mathcal{O}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n^{2/3} \log n + n_i) \\ &= \mathcal{O}(n) \end{aligned}$$

6.4.5 Main Loop

Phase I: Initialization

This phase sets up data structures and partitions the voxel space into chunks. It begins by collecting and initializing anchors and delimiters, which is linear in the number of entries, A and D, respectively. Next, the anchors are grouped into spatial chunks with a complexity of $\mathcal{O}(A)$. Finally, for each of these groups, a chunk is created and initialized with the corresponding anchors. During chunk creation, various data structures are initialized, including the delimiter array, storing delimiter information for each voxel. This requires a physics box overlap per voxel and is the most computationally expensive part of this process. Consequently, the overall complexity of chunk initialization is $\mathcal{O}(A \cdot k)$, where k represents the cost per chunk.

Work and Span Since all of these steps are performed sequentially, the span equals the work. Combining, we obtain:

$$W_{\text{init}} = S_{\text{init}} = \mathcal{O}(A + D) + \mathcal{O}(A) + \mathcal{O}(A \cdot k) = \mathcal{O}(A \cdot k + D)$$

Phase II: Initial Flood-Fill

After the voxel grid and chunks are initialized, flood-fill jobs are scheduled. Each chunk executes a multi-source frontier-based BFS expansion from its anchors, filling in reachable voxels up to delimiter boundaries or chunk edges.

Let n denote the total number of voxels per chunk and C_i the number of dirty chunks to process in iteration i. For each chunk j processed in this iteration, let $n_{i,j}$ represent the number of voxels processed. Define

$$n_{\text{i max}} = \max_j n_{i,j} \leq n$$

as the maximum number of voxels processed in any single chunk during iteration i.

The number of dirty chunks in the first iteration, C_1, is upper bounded by A, in the case of only one anchor per chunk.

Work Each chunk performs a flood fill over its local volume. Let W_i be the work for iteration i with $i = 1$ being the initial flood fill. The total work is proportional to the sum of all chunk sizes:

$$W_i = \mathcal{O}(\sum_{j=1}^{C_i} n_{i,j}) = \mathcal{O}(\sum_{j=1}^{C_i} n_{\text{i max}}) = \mathcal{O}(\sum_{j=1}^{C_i} n) = \mathcal{O}(C_i \cdot n)$$

Span Let S_i be the span analogous to W_i. We assume the cost of scheduling of flood fill jobs to be insignificant. Once the jobs are launched, each executes independently. The longest-running job (i.e., the chunk j with $n_{i,j} = n_{\text{i max}}$) dominates the span:

$$S_i = O(n_{\text{i max}}) = O(n)$$

Phase III: Main Loop Cutoff Resolution and Flood Filling

The main loop of Algorithm 3 runs for I iterations. Each iteration consists of three main components: cutoff resolution, border synchronization, and re-scheduling of flood fill jobs. We analyze the time complexity of each step in the Work–Span model.

Cutoff Resolution: In the `HandleCutoffs` function, the algorithm iterates over all chunks and inspects their `possiblyCutoffVoxels` lists. If any such voxels are found, a cutoff verification step is run. Should a voxel be deemed cutoff, the algorithm removes the corresponding subvolume and re-propagates labels using a localized flood fill (see Section 4).

The worst-case work for this step is proportional to the size of the chunk, i.e., $W_{\text{cutoff}} = \mathcal{O}(n)$ as we analyzed before in Section 6.4.4.

SwapBorders After cutoff handling, each dirty chunk synchronizes its boundary voxels with its immediate neighbors. For each dirty chunk, up to six faces (in the six cardinal directions) may be updated. If a neighboring chunk does not yet exist, it is created dynamically with cost k.

Assuming C_i dirty chunks participate in the synchronization step, the total work is $W_{\text{swap}} = \mathcal{O}(C_i \cdot k)$. Since this is done sequentially, the span equals the work: $S_{\text{swap}} = W_{\text{swap}}$. In practice, since most chunks are not modified in a given iteration, we have $C_i \ll C$.

ScheduleFloodFillJobs Finally, previously swapped chunks are re-evaluated independently via parallel flood fill jobs. We already analyzed both work W_i and span S_i in Phase II above.

Per-Iteration Complexity Combining the three steps, the total work and span per iteration are given by:

$$\begin{aligned} W_{\text{iter}} &= W_{\text{cutoff}} + W_{\text{swap}} + W_{\text{i}} \\ &= \mathcal{O}(n) + \mathcal{O}(C_i \cdot k) + \mathcal{O}(C_i \cdot n) \\ S_{\text{iter}} &= S_{\text{cutoff}} + S_{\text{swap}} + S_{\text{i}} \\ &= \mathcal{O}(n) + \mathcal{O}(C_i \cdot k) + \mathcal{O}(n) \end{aligned}$$

6.4.6 Total Complexity

Summing all up, we have for the total algorithm work W_{total}:

$$\begin{aligned} W_{\text{total}} &= W_{\text{init}} + W_1 + \sum_{i=2}^{I} W_{\text{iter}} \\ &= \mathcal{O}(A \cdot k + D) + \mathcal{O}(C_1 \cdot n) + \sum_{i=2}^{I} \mathcal{O}(n) + \mathcal{O}(C_i \cdot k) + \mathcal{O}(C_i \cdot n) \qquad \text{(extract } k \text{ to } \mathcal{O}(C \cdot k)) \\ &= \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \mathcal{O}(C_1 \cdot n) + \sum_{i=2}^{I} \underbrace{\mathcal{O}(n) + \mathcal{O}(C_i) + \mathcal{O}(C_i \cdot n)}_{=\mathcal{O}(C_i \cdot n)} \\ &= \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \sum_{i=1}^{I} \mathcal{O}(C_i \cdot n) \end{aligned}$$

Let us examine the dominant term:

$$\sum_{i=1}^{I} \mathcal{O}(C_i \cdot n).$$

With a few assumptions, we can simplify this expression. In the worst-case scenario, each chunk is processed up to A times. This is discussed in more detail in the worst-case input analysis (see Section 6.5).

Under this assumption, we get:

$$\sum_{i=1}^{I} \mathcal{O}(C_i) = \mathcal{O}(A \cdot C),$$

where C is the total number of chunks. Inserting into our dominant term, the total cost becomes:

$$\mathcal{O}(A \cdot C \cdot n).$$

Since $C \cdot n = N$, the total number of processed voxels, we can simplify further to:

$$\mathcal{O}(A \cdot N).$$

Including all relevant terms, the total work becomes:

$$W_{\text{total}} = \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \mathcal{O}(A \cdot N) = \mathcal{O}(A \cdot N)$$

since $A \cdot N$ dominates the rest of the terms.

Span Let us examine the complexity of the total span S_{total}:

$$\begin{aligned}
S_{\text{total}} &= S_{\text{init}} + S_1 + \sum_{i=2}^{I} S_{\text{iter}} \\
&= \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \mathcal{O}(n) + \sum_{i=2}^{I} \underbrace{\mathcal{O}(n) + \mathcal{O}(C_i) + \mathcal{O}(n)}_{=\mathcal{O}(n)} \qquad \text{(extracted } k \text{ to } \mathcal{O}(C \cdot k)\text{)} \\
&= \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \sum_{i=1}^{I} \mathcal{O}(n) \\
&= \mathcal{O}(C \cdot k) + \mathcal{O}(A + D) + \mathcal{O}(I \cdot n) = \mathcal{O}(C \cdot k + I \cdot n)
\end{aligned}$$

Thus, assuming an infinite number of parallel execution units, we conclude that the total span is dominated by the sum of chunk creations including collision checks and the number of iterations multiplied by the number of voxels per chunk.

Time analysis using the work span model

Under the Work–Span model, assuming P available cores:

$$T_P = \mathcal{O}\left(\frac{W_{\text{total}}}{P} + S_{\text{total}}\right) = \mathcal{O}\left(\frac{A \cdot N}{P} + C \cdot k + I \cdot n\right)$$

Interpretation: When $P \ll A \cdot N$, the term $\frac{A \cdot N}{P}$ dominates the runtime, indicating strong potential for parallel speedup as the number of processors increases.

Bottlenecks. The span term $S_{\text{total}} = O(C \cdot k + I \cdot n)$ imposes a hard lower bound on execution time. Regardless of how many processors are used, large values of $C \cdot k$ or $I \cdot n$ will limit scalability.

Ideal case. If $C \cdot k$ and $I \cdot n$ are small (e.g., constant or logarithmic), and the total work $A \cdot N$ is large, the algorithm can achieve near-linear speedup up to $P \approx \frac{A \cdot N}{S_{\text{total}}}$.

6.5 Input-Specific Complexity Analysis

We have previously demonstrated that the cost per iteration of the algorithm scales linearly with respect to the number of affected voxels and chunk jobs. However, the total

number of iterations I required to complete the full flooding process is highly input-dependent. This makes a precise complexity analysis difficult, particularly for inputs that deviate from uniform or idealized spatial distributions.

To make this analysis concise, we examine two representative input configurations that cover the range of algorithmic behavior:

- **Worst-case input:** A hand-crafted, adversarial scene layout that maximizes iteration count and voxel overwrites while minimizing per-iteration work. This configuration serves to reveal theoretical performance bottlenecks and the limitations of parallel execution in extreme cases.
- **Realistic input:** A more plausible scenario based on assumptions about delimiter sparsity. This input reflects common use cases in practice and provides insight into the algorithm's expected performance under non-adversarial conditions.

6.5.1 Worst-Case Input Construction

From the previous analysis we know that the total work W_{total} of the chunked algorithm is linear in the total amount of processed voxels N and linear in the number of anchors A. The span S_{total} however is highly dependent on the number of iterations. To analyze the worst-case behavior of the algorithm, we aim to construct an input that maximizes the number of iterations I in the algorithm's main loop, while keeping the total number of explored voxels N relatively small. The performance is then evaluated as the ratio between the number of iterations and the number of explored voxels.

Three characteristics contribute negatively to algorithmic efficiency:

1. **Sparse chunk job filling:** When many chunk jobs are initiated but each fills only a small number of voxels, the scheduling overhead dominates, and the work per chunk is not worth the cost.
2. **High overwrite frequency:** When a voxel is overwritten multiple times due to overlapping anchor regions, the effective work per voxel increases, potentially leading to cutoff sections with a large number of redundant operations.
3. **Low Per-Iteration Work:** Although a high iteration count does not increase the total work despite the scheduling overhead, it significantly affects parallel performance. Specifically, the span S_{total} is linear in N for the worst possible input. In the limit, this means the workload becomes effectively sequential, diminishing the benefits of parallel execution.

To maximize the number of iterations required, consider the contrived input depicted in Figure 6.5. The structure resembles a labyrinthine of delimiters or "snake-like" spatial configuration at the chunk borders. This setup forces the flood-fill algorithm to interrupt and restart each time it crosses a chunk boundary, thereby increasing iteration count significantly.

This pattern is particularly effective because, asymptotically, only three voxels are required per chunk border crossing as $N \to \infty$. Thus, the number of iterations I approaches roughly $N/3$. This is highly inefficient from a performance perspective, since the overall

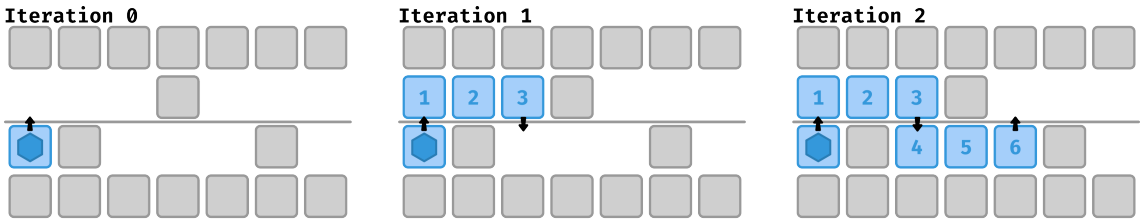


Figure 6.5: Worst Case Analysis: Maximizing algorithm iterations per voxel in a "snake-like" pattern.

complexity becomes:

$$S_{\text{total}} = \mathcal{O}(C \cdot k + I \cdot n) = \mathcal{O}\left(C \cdot k + \frac{N}{3} \cdot n\right) = \mathcal{O}(N)$$

Using this pattern, we can construct inputs where I grows linearly with N. Consequently, the span S_{total} becomes linear in N, implying that the parallel execution behaves as if it were sequential. This pessimistic scenario assumes idealized scheduling without overhead. In practice, scheduling incurs additional costs, further degrading performance.

Voxel Overwrites

Another important metric is the number of times a voxel is overwritten. The same snake pattern allows us to create inputs where specific voxels are overwritten by multiple anchors. This is achieved by deliberately slowing down the expansion of anchor subspaces to just three voxels per iteration. By chaining several such anchors with minimal spacing differences, we can force individual voxels to be overwritten repeatedly—once by each anchor.

Although this is a hand-crafted case, it demonstrates that certain inputs can lead to almost every voxel being overwritten by every anchor. In such cases, the computational complexity grows to:

$$\mathcal{O}(A \cdot N)$$

where A denotes the number of anchors and N the number of voxels. This is the upper bound of W_{total} which we derived earlier.

6.5.2 Realistic Input Assumptions

While the previously discussed worst-case construction reveals important theoretical limits, it is highly contrived and does not reflect typical input data encountered in practice. In this section, we introduce a more realistic scenario by making several simplifying but plausible assumptions about the spatial structure of the input.

Assumption: Delimiters are assumed to be sparsely placed in the scene. Furthermore, we assume that if a delimiter exists on a chunk boundary, it either completely blocks flood fill propagation between adjacent chunks (e.g., a solid wall), or allows near-complete flooding through a localized opening (e.g., a wall with a window). We explicitly exclude densely packed or adversarial geometries near chunk borders, such as the "snake-like"

pattern introduced in the worst-case analysis. Instead, we consider chunk boundaries to be largely permeable or entirely opaque, with minimal local complexity.

Under these assumptions, the number of overwrites per voxel is naturally bounded. Specifically, a voxel may be overwritten at most three times in the worst case. Consider the following scenario to illustrate this upper bound.

Constructing a Three-Way Overwrite Case. Let four chunks meet at a shared corner. Place four anchors—denoted A, B, C, and D—such that each anchor lies within one of the chunks and is located close to the shared corner. Define voxel V to reside in the diagonally opposite chunk from anchor A, such that the Manhattan distance from A to V is minimal (e.g., 3 units).

Assuming no delimiter restricts propagation at the chunk borders, the algorithm proceeds as follows:

1. In the first iteration, each anchor floods its own chunk. Assume anchor D reaches and writes voxel V.
2. In the subsequent iteration, anchor C's flood reaches into chunk D and overwrites V.
3. Similarly, anchor B floods into chunk C, again overwriting V.
4. Finally, anchor A floods through all three neighboring chunks to overwrite V a third time.

This example achieves three overwrites of the same voxel without introducing adversarial geometry. The overwrite chain is caused solely by chunk-border interruptions and varying anchor proximities.

Delay Bound by Border Crossings Since we assume no artificial obstructions at chunk boundaries by delimiters, the only delay in flood propagation stems from the need to cross up to three adjacent chunks—one per dimension. This introduces a constant upper bound on overwrite delay, which simplifies further analysis.

Reevaluating Iteration Count Under these assumptions, we can now re-evaluate the number of iterations I by considering the algorithm's behavior on the chunk level. At this level, the flood fill process resembles a breadth-first search (BFS), expanding from each anchor to neighboring chunks.

Let A_i denote an anchor, and let its maximum reach in any direction—assuming no delimiters—is determined by the ratio between its distance to the furthest reachable voxel and the chunk size. This gives a conservative upper bound on the number of chunks the anchor may flood in a straight line:

$$D_i = \left\lceil \frac{\text{maxAnchorDistance}(A_i)}{\text{chunkSize}} \right\rceil$$

This value, D_i, can increase if walls are present, since the anchor's path may be forced to take longer routes. In such cases, the set of affected chunks may form an approximate sphere, and the maximum exploration depth is proportional to the radius R of that sphere (in chunks). Let D_{max} be the maximum of all D_i over all anchors.

Span in Realistic Input. With this, the maximum number of iterations is upper-bounded by $D_{\text{max}} + 3$, where the additional 3 accounts for possible overwrite delays due to chunk-border crossings in three dimensions. Inserting this into our formula for the algorithm's span S_{total}, we get:

$$S_{\text{total}} = O(C \cdot k + I \cdot n) = O(C \cdot k + D_{\text{max}} \cdot n)$$

From a graph-theoretic perspective, the flood fill resembles BFS with a complexity of $\mathcal{O}(V + E)$, where V is the number of voxels and E the number of edges (neighboring voxel pairs). Since each voxel has a constant maximum number of neighbors, both V and E scale linearly with the total number of processed voxels N, yielding a sequential complexity of $\mathcal{O}(N)$.

The exploration depth D_{max} can be estimated using the radius R (in chunks) of the sphere that encompasses the reachable chunk space for a given anchor. The volume of that sphere is in $\mathcal{O}(R^3)$, so we can approximate R as:

$$R = \mathcal{O}(\sqrt[3]{N})$$

Substituting back, the span becomes:

$$S_{\text{total}} = \mathcal{O}(C \cdot k + R \cdot n) = \mathcal{O}(C \cdot k + \sqrt[3]{N} \cdot n)$$

This indicates a sublinear span with respect to N for the flood-fill portion of the algorithm under realistic assumptions.

Limitations Due to Chunk Creation Despite the sublinear behavior of the flooding phase, the overall span remains linear in the number of chunks C. This is because chunk creation is inherently sequential due to Unity's restriction on parallel physics queries in non-ECS projects. Creating each chunk requires physics-based voxel validation, which introduces non-parallelizable overhead.

If Unity allowed parallel physics queries outside of ECS, or if ECS were adopted, chunk creation could potentially be parallelized, enabling the entire algorithm to approach sub-linear span characteristics. However, as discussed in Section 2.7.2, we opted against using ECS due to project-specific constraints.

6.6 Space Complexity

In this section, we analyze chunked parallel implementation in terms of space requirements. The algorithm divides the 3D volume into C cubic chunks, each of side length s (so $n = s^3$ voxels per chunk). Note that $C \cdot n$ generally overestimates N due to padding and chunk boundaries, but simplifies worst-case analysis.

For each chunk, we maintain an array of voxels of length $n = s^3$, requiring $\mathcal{O}(n)$ space. Additionally, a bit array for the delimiters at each voxel using $\mathcal{O}(n)$ bits. To manage inter-chunk connectivity, we store two border hashsets on each of the six faces. In total, this

requires:

$$2 \times 6 \times s^2 = 12\, s^2 = \mathcal{O}(s^2)$$

As in the sequential case, the location graph uses $\mathcal{O}(A^2 + A \cdot D)$ space but remains negligible in practice in comparison to $C \cdot n$.

Across all C chunks, the aggregate space usage is:

$$S_{\text{par}} = C \cdot \big(\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(s^2)\big) + \mathcal{O}(A^2 + A \cdot D) = \mathcal{O}\big(C\, n + C\, s^2\big).$$

Since $n = s^3$, we can upper bind $C \cdot s^2$ with $C \cdot n$ and simplify:

$$S_{\text{par}} = \mathcal{O}(C \cdot n).$$

Comparison to the sequential version Both implementations exhibit linear space complexity in the size of the explored region. The chunked parallel version may incur a larger constant factor due to chunk padding and border metadata, but retains $O(N)$ scaling. This confirms that our parallelization strategy does not increase the asymptotic space requirement compared to the sequential baseline.

6.7 Chunked Runtime Queries

The runtime queries in the chunked algorithm build upon the concepts presented for the sequential implementation but introduce several key modifications that significantly improve performance and scalability. As the core logic remains similar, we omit the algorithm listing and instead focus on the differences and enhancements.

In the sequential version, the central data structure is the `borderDict`, which maps voxel positions to their corresponding anchors. To save memory, only voxels on anchor boundaries are stored.

The chunked system adopts a similar strategy but applies it in a localized manner. Instead of requiring the entire border voxel dictionary in memory, it suffices to identify the chunk containing the queried transform and load only that chunk. Each chunk stores its local anchor boundaries and also includes a representation of "empty" space in the form of an auxiliary anchor. This eliminates the need to explicitly handle empty regions during queries, as every position is effectively assigned to some anchor.

This change increases the required total storage slightly compared to the sequential implementation, but in return, it allows for significantly faster, fully localized queries, independence from voxel resolution during query time and constant-time complexity for most queries, bounded only by chunk size.

Figure 6.6 illustrates the differences between sequential and parallel runtime queries. Query locations are shown as black circles. Raycasts are drawn as black lines, with rounded squares marking the hit points where they intersect borders. Anchor volumes are lightly shaded, and borders are highlighted. On the left, the sequential version may require traversing the entire ray length to determine which anchor a voxel belongs to. On the right, the parallel version requires significantly fewer steps: for each query, it is sufficient to fire a single ray toward the nearest chunk border. Queries in empty space

are also more efficient, since empty volumes are stored as separate anchors. While this increases memory, it ensures consistent query results and allows queries outside of any chunk to terminate immediately. Additionally, all chunk borders are explicitly stored, which introduces a apparently large overhead for small chunk sizes, though for larger chunks it becomes less impactful as discussed in the following sections.

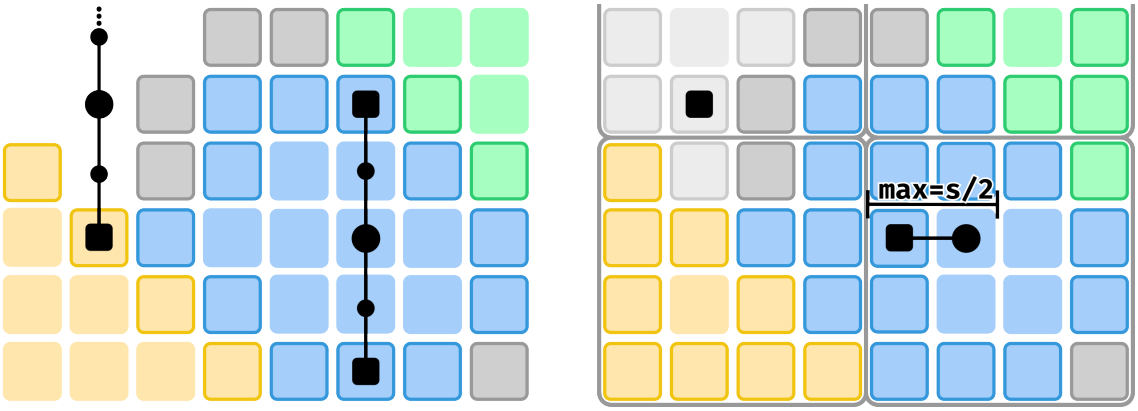


Figure 6.6: Sequential (left) vs. parallel (right) runtime queries. The parallel method requires only a single ray toward the nearest chunk border, with at most half the chunk size in steps, and handles empty space efficiently with dedicated anchors—at the cost of a higher memory footprint.

6.7.1 Time Complexity

The time complexity of runtime queries in the chunked system depends on the chunk size s. Due to the auxiliary anchor representing empty space, only a single raycast is necessary to determine the anchor containing a given transform. Moreover, since all space is covered, the search does not need to follow a fixed direction but can simply proceed toward the nearest boundary voxel.

Given a transform's position within a chunk, we determine the nearest border by checking along the shortest path to a boundary in any direction. In the worst case, this is bounded by $s/2$ steps (e.g., starting from the center of the chunk). Thus, the query complexity becomes:

$$\mathcal{O}\left(\frac{s}{2}\right) = \mathcal{O}(s)$$

Since s is typically small and constant after system initialization, this results in very fast anchor lookups, which is confirmed by benchmarks presented in Section 8.3.

By contrast, the sequential version had complexity $\mathcal{O}(D/v)$ for maximum distance to opposite borders D and voxel size v, which scales poorly with smaller voxel sizes or larger scene volumes. The chunked system removes this dependence on D and v, offering a considerable performance improvement for high-resolution or large-scale scenes.

In rare cases, such as when a transform lies near a chunk boundary, the fallback sphere cast may require access to up to 7 neighboring chunks (e.g., in a corner adjacency). If these chunks are already loaded in memory, the lookup remains fast and the complexity unchanged. Thus, the worst-case in-memory query cost remains $\mathcal{O}(s)$.

Chunk Loading Cost: So far, we have assumed all required chunks are already resident in memory. If a chunk must be loaded from disk or another storage layer, an additional cost c must be considered. The total worst-case complexity becomes:

$$\mathcal{O}(s + c)$$

Since c can be several orders of magnitude higher than the in-memory traversal cost, performance becomes I/O-bound in uncached scenarios. This effect is analyzed in detail in Section 8.3.

6.7.2 Memory Considerations

The increased performance of the chunked system comes with a storage trade-off. Each chunk must store not only the anchor border voxels but also an additional layer of voxels along its boundaries to support localized queries. For a chunk of size s, this surface storage requirement scales with the surface area:

$$\mathcal{O}(6 \cdot s^2)$$

Thus, while the chunked algorithm only stores surfaces (not full volumes), similar to the sequential version, it generally requires equal or greater storage due to this boundary duplication and auxiliary anchor representation. In terms of asymptotic complexity, storage remains in $\mathcal{O}(s^2)$ per chunk, comparable to the sequential method's per-anchor border storage. However, in practice, the chunked system always stores at least as many voxels, and often significantly more, than the sequential approach.

7 Optimizations

This chapter highlights optimizations that were either taken to make the chunked algorithm faster, or consciously not taken with a reasoning behind it.

7.1 Data Serialization and Storage in Unity

Both the sequential and chunked implementations of the Space Foundation System persist their computed results using Unity's `ScriptableObject`s.

7.1.1 ScriptableObjects in Unity

`ScriptableObject`s are a data container class in Unity that allows developers to store large amounts of shared data independent of scene objects. They are particularly useful for storing configuration data, game settings, or computational results, as in our case. One of the key benefits of using `ScriptableObject`s is their seamless integration into the Unity Editor, where they can be easily serialized, edited, and reused across the project.

When saved, `ScriptableObject`s are stored in a YAML-like format. This format is human-readable, version-control friendly, and simple to inspect and debug directly. These characteristics make it ideal during development, especially for testing and iterative adjustments. Additionally, Unity automatically manages the lifecycle of these assets, and they are deeply embedded in Unity's asset pipeline and serialization system, which simplifies development and reduces boilerplate code.

7.1.2 Limitations and Optimization Opportunities

Despite their usability and integration benefits, `ScriptableObject`s are not optimal in terms of storage efficiency. The YAML-based text format, while accessible, is verbose and not optimized for performance or space. A more compact and efficient approach would be to store and load the computed data as compressed binary files. These could contain only the raw numerical data without metadata or redundant structure, reducing file size significantly and improving load times.

Another optimization target is the way anchor identifiers are stored. Currently, anchors are identified using string-based IDs (e.g., `"a42"`). Although convenient for readability and debugging, string IDs are less space-efficient than numerical identifiers. For instance, the string `"a42"` occupies 3 bytes, whereas the numeric ID `42` can be represented in a single byte. Since millions of anchor voxels may be stored, these small differences in storage requirements accumulate, and optimizing them could yield substantial performance improvements.

7.2 Usage of Integer IDs

String-based identifiers are suboptimal in terms of performance and memory efficiency. Their primary advantage lies in their expressiveness and variable length, which can be useful for human-readable identifiers. However, within the context of the Space Foundation System, such flexibility is unnecessary. A 16-bit integer can represent $2^{16} = 65,536$ unique identifiers for anchors which is even for large and complex scenes sufficient.

7.2.1 Transition from Strings to Integers

To improve performance and reduce memory usage, the chunked implementation replaces string-based anchor IDs with 16-bit integers. To maintain compatibility with the sequential implementation and ensure correctness, an interface was introduced that maps between string and integer IDs. This mechanism allows the algorithm to accept string IDs externally (e.g., for debugging or testing purposes) while internally converting them to integers for processing. After computation, the results can be converted back to string format if needed.

Although this conversion introduces a small overhead, it is negligible compared to the performance benefits gained through reduced memory allocation and improved data locality.

7.2.2 Technical Considerations

String objects in C# are typically allocated on the heap due to their dynamic size, whereas primitive types such as integers are stored on the stack (when not boxed or captured). Heap allocation introduces additional indirection and can negatively affect cache performance. Strings' variable length also makes them poor fits for cache lines, potentially increasing cache misses during iteration (see Section 2.4.

Furthermore, strings in Unity are not blittable. A blittable type is one that has the same binary representation in managed and unmanaged memory. As a result, strings cannot be used directly within Unity's Job System or Burst Compiler, both of which require blittable, fixed-size types.

7.3 Architectural Changes and Voxel Struct Optimization

Dividing the system into spatial chunks enabled several architectural optimizations, particularly with regard to memory layout and performance. One major area of improvement lies in the design and handling of voxel data structures.

7.3.1 Sequential vs. Chunked Voxel Representation

As discussed in the background chapter (Section 2.4), memory layout and access patterns have a substantial impact on performance due to modern CPU cache hierarchies [58, 51]. The choice between a sequential and a chunked voxel representation directly reflects these considerations.

The sequential implementation uses hash sets to manage explored voxels and store their data. While this design is flexible—supporting arbitrary voxel positions in space—it introduces several inefficiencies:

- Each explored voxel must be explicitly stored in the hash set, increasing memory overhead and access latency.
- Positions are stored as `Vector3Int`, requiring $3 \times 4 = 12$ bytes per voxel for coordinates alone.
- Arbitrary positions prevent the use of direct array indexing, leading to irregular access patterns and reduced cache efficiency, as highlighted by Rivera and Tseng [51].

In contrast, the chunked implementation organizes voxels in fixed-size arrays within the chunks, aligning with cache-friendly data layouts [74] and Unity's DOTS principles. This enables several key optimizations:

- Memory can be allocated contiguously, improving spatial locality and leveraging hardware prefetching [58].
- Voxel positions are implicitly represented by array indices, removing the need for `Vector3Int` storage.
- Flat arrays produce predictable sequential memory access patterns, reducing cache misses and avoiding unnecessary memory indirections.

Each chunk has a fixed size s, containing s^3 voxels stored in a flattened 1D array. Conversion between 3D coordinates and 1D indices is handled with simple integer arithmetic:

$$\text{index} = x + y \cdot s + z \cdot s^2$$

$$(x, y, z) = \left(\text{index} \bmod s,\ \left(\left\lfloor \frac{\text{index}}{s} \right\rfloor \bmod s \right),\ \left\lfloor \frac{\text{index}}{s^2} \right\rfloor \right)$$

This mapping entirely removes the need to store explicit voxel coordinates, improving both memory footprint and access speed. Furthermore, by keeping chunk sizes small—for example, 16^3 voxels (16 KB)—each chunk fits comfortably within the L1 cache of our mid-range benchmarking system (see Section 8.1), making more complex spatial ordering schemes such as Morton or Hilbert curves unnecessary [47, 73].

7.3.2 Memory Footprint and Struct Comparison

Our goal was to minimize the memory footprint of the voxel data structures and ensure they are suitable for high-performance use with Unity's Burst Compiler and C# Job System. Table 7.1 shows a comparison of the sequential and chunked voxel struct representations:

Transition: Several fields from the sequential voxel struct were either removed or replaced in the chunked implementation to improve performance and reduce memory usage. The `position` field was omitted entirely, as voxel indices now implicitly encode spatial position within the chunk at no additional memory cost. The `anchor` reference was replaced with a compact ID, which both reduces memory consumption and ensures the struct remains blittable. The `exploredFromPosition` field, previously used for backtracking during flood-fill, was removed by redesigning the exploration logic to avoid

Field	VoxelData	Type	ChunkVoxelData	Type
position	12	Vector3Int	– (array index)	–
anchor	8	Anchor	2	ushort
exploredFromPosition	12	Vector3Int	–	–
isBorder	1	bool	1 bit (external)	–
hitsDelimiter	1	bool	1 bit (external)	–
floodedFromChecked	4	int	–	–
distance	–	–	2	ushort
Total	**38 bytes**		**4 bytes + 2 bits**	

Table 7.1: Memory Comparison of VoxelData vs. ChunkVoxelData

relying on positional history.

Similarly, the isBorder flag is no longer necessary, since border voxels are now handled in a dedicated post-processing step following the main flood-fill operation. The hitsDelimiter boolean was replaced by a shared bit array, reducing its memory footprint from a full byte per voxel to a single bit. Lastly, the floodedFromChecked field, originally used to track how many directions had visited a voxel for object pooling,was omitted. Since structs in C# are stack-allocated and do not incur garbage collection overhead, object pooling offers no meaningful performance gain in this context.

In the chunked system, voxel exploration is not strictly sequential. To resolve conflicts and maintain consistency during parallel or out-of-order processing, each voxel includes a distance field. This field records the distance from the origin of flood-fill and is used to determine priority when multiple fills compete for the same voxel.

7.4 Burst and C# Jobs

As mentioned in the background chapter (see Section 2.7), several aspects must be considered to improve performance. For SIMD vectorization, we rely on the Unity Burst compiler's auto-vectorization capabilities (see Section 2.5), writing C# code with a reduced set of language features to simplify both vectorization and memory management [34, 20]. By following these constraints—such as using blittable types and avoiding managed memory—we enable Burst to translate our job code into optimized native instructions that efficiently utilize the CPU's SIMD units.

For multithreading, we employ the Unity C# Job System (Section 2.4), which is Unity's default and recommended approach for parallelism [71, 70]. The Job System schedules work across all available CPU cores using a work-stealing scheduler [7], enabling scalable parallel execution while avoiding common pitfalls such as false sharing [6]. This combination of SIMD within cores and multithreading across cores aligns with Unity's Data-Oriented Technology Stack (DOTS) principles, ensuring both computation and memory performance are maximized for voxel chunk processing.

7.5 Physics Queries

A substantial portion of the computational workload in both the sequential and parallel implementations stems from physics overlap checks. In the chunked parallel version, two improvements were made to optimize this process. First, each voxel undergoes at most one collision check, rather than potentially multiple redundant checks as in the sequential version. Second, the implementation replaces `Physics.OverlapBox` with `Physics.OverlapBoxNonAlloc`, which reduces memory allocations during queries. However, in practice, this change did not result in a noticeable performance difference.

Despite these optimizations, physics queries remain a major bottleneck, primarily because they scale linearly with the number of voxels. Additionally, Unity's built-in physics engine runs exclusively on the main thread and is not thread-safe. While Unity does offer parallel physics queries through the Entity Component System (ECS), these are tightly integrated into the DOTS architecture and require significant structural changes to adopt.

8 Benchmarks

This chapter presents an extensive evaluation of the algorithm's real-world performance across multiple dimensions. Benchmarks were conducted on four distinct Unity scenes, each varying in size and geometric complexity. We compare the performance of the sequential and chunked parallel algorithms under various voxel resolutions, chunk sizes, and runtime configurations. Key aspects analyzed include chunk size optimization, the role of Burst compilation, scalability under high-resolution conditions, and the overhead introduced by physics collision checks. Furthermore, we assess single-threaded versus multithreaded execution and explore runtime query performance, including the effects of preloading and on-demand chunk access. These benchmarks serve to identify bottlenecks, validate theoretical expectations, and guide practical implementation choices.

8.1 Benchmark Setup

All benchmarks were conducted on a system equipped with an AMD Ryzen 5 7640U processor, featuring 6 cores and 12 threads. The CPU operates at a base frequency of 3.5 GHz and can boost up to 4.9 GHz. The system includes 384 KB of L1 cache, 6 MB of L2 cache, and 16 MB of L3 cache.

- **Operating System:** Fedora Linux 42 (Workstation Edition)
- **Kernel Version:** 6.15.8-200.fc42.x86_64
- **Memory:** 32 GB DDR5 at 5600 MHz
- **Unity Version:** 2022.3.58f1
- **Burst Compiler:** Version 1.8.21

We benchmarked both the sequential and chunked parallel implementations of the algorithm across four scenes of varying complexity. The scenes are ordered by increasing size and complexity from left to right, as shown in Figure 8.1 and Table 8.1.

	House	**Villa**	**Region**	**Mapper's Peak**
Dimensions	10^3	50^3	$100^2 \times 50$	$110^2 \times 50$
Colliders	105	33	1 553	1 360
Mesh Colliders	100	31	1 461	1 278
Mesh Collider Vertices	2 534	464	48 289	127 135
Mesh Collider Triangles	1 362	232	28 152	58 910
Terrain Collider Resolution	–	–	513	1 025
Anchors	6	7	74	28
Delimiters	6	7	186	73

Table 8.1: Unity benchmark scenes and their complexity in size and collider counts

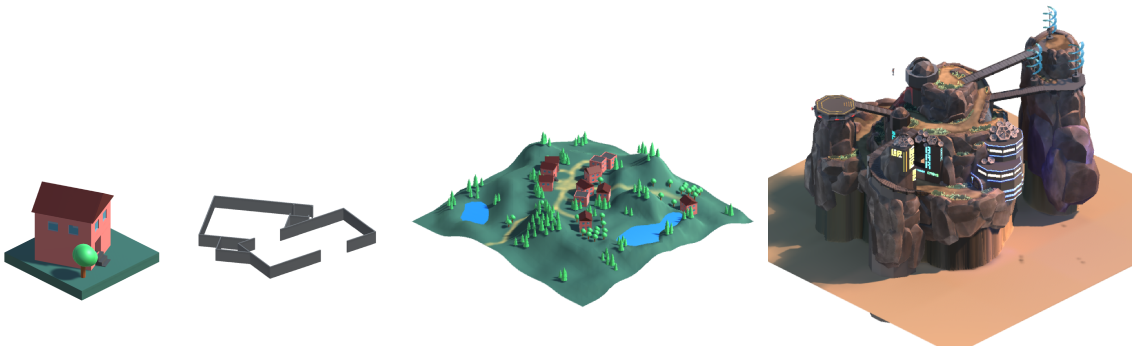


Figure 8.1: Benchmark Scenes with varying size and complexity: (a) House, (b) Villa, (c) Region, (d) Mapper's Peak (left to right).

8.2 Space Foundation Creation

Our first benchmarks focus on the creation of the space foundation, comparing the performance of both algorithms. We tested execution times across 20 runs and mostly show and discuss the mean value. For detailed statistics, including median, minimum, maximum, and standard deviation, refer to the Appendix 1. The chunk size is set to 16, unless stated differently.

8.2.1 Execution Time Comparison

We begin by comparing the total execution time between the sequential algorithm and the chunked parallel variant. The comparison is illustrated using data for voxel sizes of 1.0 and 0.25.

Voxel Size 1.0: Table 8.2 compares the execution times of the sequential algorithm (denoted as *Seq*) and the chunked parallel algorithm (denoted as *Chu*) across four benchmark scenes, with a voxel size of 1.0.

Alg.	Scene	Time	Speedup	Voxels	Iteration
Seq	House	5.948	0.960 7	679.0	–
Chu	House	6.191	0.960 7	16 380	5
Seq	Villa	25.94	0.821 7	9 290	–
Chu	Villa	31.57	0.821 7	73 730	8
Seq	Region	1 724	3.216	397 100	–
Chu	Region	536.0	3.216	974 800	10
Seq	Mapper's Peak	3 062	2.258	490 300	–
Chu	Mapper's Peak	1 356	2.258	892 900	9

Table 8.2: Mean SFS creation times (ms) and corresponding speedups for different scenes, along with processed voxel counts and algorithm iterations (voxel size 1.0, chunk size 16). Detailed statistics are given in Appendix 1.

In smaller scenes, such as *House* and *Villa*, both algorithms exhibit similar average execution times. However, the chunked algorithm demonstrates a significantly higher standard deviation and a wider min–max range (refer to Appendix 1).

It is evident that the chunked algorithm processes a substantially larger number of voxels compared to the sequential algorithm. This behavior arises because the chunking approach tends to overestimate the spatial volume, particularly in smaller scenes like *House*, which has a volume of 10^3 units at a chunk size of 16.

As the scenes increase in size and complexity, the chunked algorithm begins to outperform its sequential counterpart. Although the chunked algorithm consistently processes more voxels, the associated overhead becomes relatively less significant. With increasing complexity, the gap between the number of voxels processed by the chunked algorithm and the sequential version narrows.

Intermediate Resolution (Voxel Size 0.5): Benchmark results for voxel size 0.5 follow the same trend observed at voxel sizes 1.0 and 0.25. For brevity, these results are omitted here and are available in Appendix 2.

Voxel Size 0.25: Table 8.3 presents execution times at a higher resolution, with voxel size reduced to 0.25.

Alg.	Scene	Time	Speedup	Voxels	Iteration
Seq	House	114.1	1.998	51 440	–
Chu	House	57.10	1.998	196 600	8
Seq	Villa	4 463	12.27	475 500	–
Chu	Villa	363.6	12.27	770 000	25
Seq	Region	1 435 000	71.86	25 540 000	–
Chu	Region	19 970	71.86	32 060 000	24
Seq	Mapper's Peak	2 343 000	54.26	30 960 000	–
Chu	Mapper's Peak	43 180	54.26	38 400 000	26

Table 8.3: Mean SFS creation times (ms) and corresponding speedups for different scenes, along with processed voxel counts and algorithm iterations (voxel size 0.25, chunk size 16). Detailed statistics are given in Appendix 4.

At this scale, the performance benefits of the chunked algorithm become increasingly pronounced. For example, in the *Mapper's Peak* scene, where the chunked algorithm was already more than 2× faster at voxel size 1.0, the speedup increases dramatically with finer resolution. In larger scenes such as *Region*, the chunked version achieves a speedup exceeding 71× over the sequential baseline.

Algorithm	Scene	Scale 1.0→0.5	Scale 0.5→0.25
Seq	House	3.220	5.950
Chu	House	1.630	5.650
Seq	Villa	6.250	27.50
Chu	Villa	3.040	3.800
Seq	Region	25.40	32.80
Chu	Region	5.560	6.700
Seq	Mapper's Peak	23.70	32.30
Chu	Mapper's Peak	5.190	6.130

Table 8.4: Scaling factors in mean execution time for halving voxel size (chunk size 16).

Table 8.4 presents the scaling factors for the mean execution time when halving the voxel size for both algorithms. In each step, the workload increases by a factor of 2 in each dimension, resulting in a total factor of 8 (i.e., 2^3). The results indicate a notable rise in execution time, with sequential runs exhibiting a much more aggressive scaling behavior than the chunked parallel algorithm.

For smaller scenes, such as *House*, the scaling is moderate. However, for larger and more complex scenes like *Region* and *Mapper's Peak*, the growth factors become extreme. Specifically, the execution time for the sequential algorithm grows by over 32x when the voxel size is reduced from 0.5 to 0.25, while the chunked algorithm shows a significantly more controlled scaling, staying below 7x for both *Region* and *Mapper's Peak*.

8.2.2 Scalability Stress Test

Encouraged by the promising results of the previous benchmarks, we conducted an additional stress test to evaluate the scalability limits of the chunked algorithm. For this purpose, we selected the *Region* scene and executed the chunked algorithm with a voxel size of 0.125.

The results are summarized as follows: the average execution time was under 3 minutes with 159.3 s. Over the course of 44 iterations, the algorithm generated a total of 56,053 chunks, corresponding to 229,593,088 voxels.

Unfortunately, additional tests with smaller voxel sizes could not be conducted due to memory constraints. The benchmark run reached a peak memory usage of 11.2 GB. Halving the voxel size to 0.0625 would increase the voxel count again by a factor of 8, which would exceed the available 32 GB of system RAM.

8.2.3 Finding the Optimal Chunk Size

The performance of the chunked algorithm depends heavily on the chosen chunk size. While the theoretical minimum is 2 (not recommended for practical reasons), there is no defined upper limit. To maintain storage and alignment efficiency, chunk sizes that are powers of two are preferred. In many of our tests, a chunk size of 16 provided a good balance, but the optimal size varies depending on several factors, most notably the dimensions and complexity of the scene.

As previously discussed in the case of the *House* scene (dimensions 10^3), a chunk size of 16 can cover the entire volume in the best case, or divide it into eight separate chunks in the worst case. This introduces considerable overhead and is therefore discouraged for small-scale scenes.

To investigate the relationship between chunk size and performance in a more realistic setting, we conducted a series of benchmarks using the *Region* scene. The results are shown in Table 8.5.

Chunk Size	Voxel Size	Time	Voxels	OVR	Iterations
4	1.0	1 026	492 800	1.241	26
8	1.0	476.7	632 320	1.592	14
16	1.0	536.0	974 848	2.455	10
32	1.0	534.6	1 048 576	2.640	6
64	1.0	545.0	1 048 576	2.640	5
128	1.0	2 123	8 388 608	21.12	5
8	0.5	3 196	3 952 640	1.262	27
16	0.5	2 979	5 062 656	1.616	15
32	0.5	3 792	7 667 712	2.448	10
64	0.5	4 030	8 388 608	2.678	6

Table 8.5: Execution times (ms) for the scene *Region* with variations in chunk size. Processed voxel counts and overhead volume ratios (OVR) are shown alongside execution time statistics. Detailed timing breakdowns are provided in Appendix Table 5

Results Table 8.5 shows that for a voxel size of 1.0, the fastest execution is achieved with a chunk size of 8. Larger chunk sizes of 16, 32, and 64 perform similarly, while a chunk size of 128 results in a significant performance drop (approximately 4× slower).

For a voxel size of 0.5, the optimal chunk size is 16, with chunk size 8 close behind and 32 trailing slightly. The differences are less dramatic in this configuration, indicating more stable performance across chunk sizes at finer voxel resolutions.

The Overhead Volume Ratio (OVR) We introduce a metric called the Overhead Volume Ratio (OVR), which captures how much additional volume the chunked algorithm processes compared to the sequential algorithm. The chunked algorithm may process unnecessary voxels due to the alignment of fixed-size chunks, particularly when the chunks do not tightly fit the region of interest.

Formally, let C be the number of chunks, s be the chunk size, N be the number of voxels processed by the sequential algorithm. Then the overhead volume ratio is defined as:

$$\text{OVR} = \frac{C \cdot s^3}{N}$$

A lower OVR indicates better efficiency, meaning the chunked algorithm is doing less redundant work. Notable jumps in OVR, such as between chunk sizes 8 and 16 at voxel size 1.0, often correlate with drops in performance.

Interestingly, at chunk sizes 32 and 64 and voxel size 1.0, number of processed voxels does not change because of the scene's spatial structure. The OVR remains constant and the processing times stay within a tight margin. However, chunk size 128 introduces substantial overhead, which aligns with the significant slowdown observed.

Practical Choice For most remaining benchmarks, a chunk size of 16 was chosen as it consistently provided good performance across a wide range of scenes, particularly the more complex ones. Chunk size 32 also performed well in many cases and may be preferred when aiming to reduce parallel job overhead on systems with limited threading efficiency.

8.2.4 Chunk Creation Time

To evaluate the overhead introduced by per-voxel physics collision checks—used to determine whether a voxel is free or occupied by colliders or delimiters—we measured the time spent solely on chunk creation and compared it to the total execution time of the algorithm.

Table 8.6 presents a breakdown of chunk creation time and the total execution time in ms of the chunked algorithm across various voxel sizes within the *Region* scene.

Workload	Voxel Size	Mean	Median	Min	Max	Std. Dev.	Ratio
Chunks	1.0	499.8	492.6	485.9	646.1	34.53	0.932
Total	1.0	536.0	528.7	522.0	681.9	34.45	1.000
Chunks	0.5	2 744	2 739	2 720	2 834	23.77	0.921
Total	0.5	2 979	2 968	2 941	3 110	41.70	1.000
Chunks	0.25	18 130	18 100	17 980	18 460	122.1	0.908
Total	0.25	19 970	19 930	19 750	20 390	165.9	1.000

Table 8.6: Breakdown of chunk creation time (Chunks) versus total execution time (Total) in ms across different voxel sizes in the Region scene, including the relative contribution of chunk creation (Ratio).

Results As shown in Table 8.6, the time spent on chunk creation constitutes a substantial portion of the total execution time, being consistently over 90%. Since chunk creation consists primarily of lightweight data structure allocations and a single physics collision check per voxel, we conclude that the dominant computational cost is due to these collision checks. This is further supported by the differences in scene parameters. *Mapper's Peak* has similar dimensions, anchor and delimiter counts as *Region*, still is about twice as expensive in runtime execution as detailed in Table 8.3. The main difference between these scenes is the much higher mesh collider amount in *Mapper's Peak* along with higher vertex and triangles counts (see Table 8.1).

8.2.5 Multithreaded Performance

Following the theoretical Work–Span analysis, it is important to evaluate the actual performance of the chunked algorithm in a real-world multithreaded environment. To do this, we conducted benchmarks in Unity by varying the `JobWorkerCount` setting.

By setting `JobWorkerCount` to 0, we effectively forced the chunked algorithm to run entirely on a single thread. This allowed us to directly compare single-threaded and multithreaded performance. The benchmarks were conducted in the *Region* scene, using a fixed chunk size of 16 and varying the voxel size. Table 8.7 reports the execution times for the chunked parallel algorithm executed sequentially (denoted Sequential) and parallel (denoted Parallel).

Type	Voxel Size	Mean	Speedup	Mean (Core)	Speedup (Core)
Sequential	1.0	628.0	1.000	128.0	1.000
Parallel	1.0	536.0	1.172	36.00	3.550
Sequential	0.5	3 679	1.000	935.0	1.000
Parallel	0.5	2 979	1.235	235.0	3.979
Sequential	0.25	25 560	1.000	7 430	1.000
Parallel	0.25	19 970	1.280	1 840	4.038

Table 8.7: Execution times (ms) of the chunked parallel algorithm. Mean is total sequential execution, Speedup is overall acceleration, Mean (Core) measures the portion of the algorithm without chunk creation, and Speedup (Core) its acceleration.

Results Table 8.7 reveals a modest overall speedup. For example, at a voxel size of 1.0, the speedup is only 1.172×, increasing slightly to 1.28× at a voxel size of 0.25. At first glance, this performance is underwhelming for a parallel algorithm and suggests inefficient multithreading.

However, a deeper look at the core portion of the algorithm paints a more favorable picture. When measuring only the time spent in the core section (i.e. excluding physics checks), the observed speedup is consistently better. Specifically, the core achieves a speedup of approximately 3.55× for a voxel size of 1.0, and just over 4× at a voxel size of 0.25.

8.2.6 Impact of Burst Compilation

In addition to comparing sequential and parallel execution, it is also valuable to examine the impact of Burst compilation on performance. Unlike multithreading, Burst does not introduce parallelism but instead improves the efficiency of compiled code through low-level optimizations such as vectorization and instruction reordering.

Because the effects of Burst are heavily influenced by hardware-specific factors—such as CPU architecture, cache size, and memory access patterns—its performance impact likely varies across systems. Nonetheless, Table 8.8 shows the execution times with Burst enabled and disabled across multiple voxel sizes for the *Region* scene.

Type	Voxel Size	Time	Time Burst	Speedup
No Burst	1.0	575.0	74.80	2.066
Burst	1.0	536.0	36.20	2.066
No Burst	0.5	3 246	502.0	2.136
Burst	0.5	2 979	235.0	2.136
No Burst	0.25	21 920	3 790	2.060
Burst	0.25	19 970	1 840	2.060

Table 8.8: Comparison of mean execution time (ms) with Burst compilation enabled and disabled. *Time* refers to the mean total execution time, while *Time Burst* indicates the mean execution time of the Burst-compiled sections only.

Results When comparing the total execution times, the improvements from Burst compilation appear modest. However, this is again due to the dominant cost of physics queries, which are handled outside of the Burst-compiled code and remain unaffected by these optimizations.

To obtain a more accurate assessment, we isolate the portion of the algorithm affected by Burst and compare its execution time with and without Burst enabled. This comparison reveals a consistent speedup of approximately 2×, independent of the voxel size.

8.3 Runtime Queries

The runtime query system allows retrieving the anchor corresponding to a specific world position during application runtime. To evaluate its performance, we conducted benchmarks with 1000 queries per batch and 100 batches per benchmark scene, using random samples of predefined positions.

We compared three variations of the query algorithm:

- **Sequential** — the sequential implementation.
- **Chunked-Preloaded** — the chunked implementation with all chunks preloaded in RAM.
- **Chunked-OnDemand** — the chunked implementation where chunks are not preloaded, but loaded on demand via Unity Addressables.

A chunk size of 16 was used in all tests. Benchmarks were conducted across all scenes. For Chunked-OnDemand, we limited testing to 10 batches due to the high overhead between runs required to manually unload Addressables and clear memory.

Challenges in Measuring On-Demand Loading Working with Unity Addressables introduces limitations when attempting to control chunk loading and unloading during runtime. While Unity provides APIs for this purpose, they do not offer fine-grained or reliable control over garbage collection or unloading delays. As a result, the performance of Chunked-OnDemand is not directly comparable to the other methods. The high standard deviation observed is attributed to inconsistent unload behavior and the inefficiency of Addressables in releasing memory, which are known design limitations.

Voxel Size	Algorithm	Mean	File Size	Voxels
1.0	Sequential	5.475	8.695	490 300
1.0	Chunked-Preloaded	1.008	13.01	892 900
1.0	Chunked-OnDemand	2 854	13.01	892 900
0.5	Sequential	45.24	39.13	3 891 000
0.5	Chunked-Preloaded	1.001	7 561	5 661 000
0.5	Chunked-OnDemand	4 862	7 561	5 661 000
0.25	Sequential	561.0	171.5	30 960 000
0.25	Chunked-Preloaded	1.129	470.6	38 400 000
0.25	Chunked-OnDemand	10 220	470.6	38 400 000

Table 8.9: Realtime query execution time in microseconds in the Scene Mapper's Peak. Comparison between Sequential algorithm and Chunked Algorithm with preloaded chunks in memory and on-demand loading. File sizes in MB.

Results Table 8.9 shows runtime query performance in the *Mapper's Peak* scene at varying voxel sizes. Results for other scenes are provided in the appendix (see Appendix 3). All measurements are in microseconds.

The Sequential approach exhibits query times that scale linearly with scene complexity. As a result, response times increase substantially for high-resolution or large-scale scenes.

In contrast, the Chunked-Preloaded implementation offers consistently low query times of approximately 1 microsecond per query, regardless of scene complexity or voxel size. For small scenes, this results in a speedup of 5×–10× compared to the sequential method. For larger scenes, such as Mapper's Peak with a voxel size of 0.25, the speedup exceeds 500×.

The main trade-off associated with the Chunked-Preloaded approach is its increased memory usage and larger file sizes, as all chunks must be loaded into RAM. The maximum memory footprint for across all scenes of all loaded chunks remains under 500 MB.

The Chunked-OnDemand strategy, on the other hand, performs significantly worse. Similar to the sequential approach, its query times scale linearly with scene complexity but are also several orders of magnitude slower due to file I/O overhead.

9 Discussion

9.1 Revisiting the Research Questions

The central question of this thesis asked: *How can chunking and parallelization improve the Space Foundation System's efficiency during both editor time and runtime?*

The chunked parallel algorithm outperforms the sequential version by a significant margin. As the complexity of the Space Foundation System (SFS) increases, the speedup between the sequential and chunked parallel implementations becomes more pronounced, with the chunked parallel approach consistently being faster.

Parallelization plays a key role in improving the efficiency of the SFS. The scaling factors for SFS creation in the chunked parallel algorithm remain in the single digits, and consistently below the factor of increase in workload. This is a significant improvement compared to the sequential implementation, where the scaling is much more aggressive.

Furthermore, chunking and the changes in architectural design not only lead to performance optimizations in SFS creation times but also enable parallel execution of what would otherwise be a sequentially dependent process. In runtime queries, the chunking approach reduces query complexity, making it primarily dependent on the chunk size, which remains small and constant. In contrast, the sequential algorithm's complexity scales linearly with both world size and voxel size. These findings directly support the main objective, showing that the SFS can efficiently scale to larger and higher-resolution environments.

The sub-questions posed in Chapter 1 were likewise answered:

- The performance comparison between sequential and chunked versions showed consistent and significant gains across almost all tested scenarios.
- The theoretical complexity analysis was validated experimentally, though showing that parallelization and other factors can have a big impact on real time behavior.
- Unity's optimization features, the C# Job system and the Burst Compiler were confirmed as effective in reducing execution times, although they impose architectural constraints.
- Substantial changes to data layout, serialization, and algorithmic structure were necessary to realize the parallel version.

9.2 Interpretation of Results

The evaluation highlights two key insights. First, the reference implementation, while conceptually simple and correct, becomes a severe bottleneck as scene sizes increase. Runtime grows disproportionately with scene size and voxel resolution, making the sequential variant impractical beyond small test cases. In contrast, the chunked algorithm

demonstrates a fundamental shift in scalability: by bounding locality, it reduces exploration costs and achieves effectively constant-time runtime queries when relevant chunks are memory-resident. This transition is not merely a speedup but a qualitative improvement in how the algorithm scales with problem size.

The chunked algorithm's ability to handle volumes upward of 229 million voxels with efficiency underscores its suitability for production-scale applications, particularly in high-resolution or large game levels. Designers benefit not only from faster generation but also from dramatically reduced query times—on the order of microseconds for preloaded chunks—enabling responsive feedback cycles in the editor. Despite slightly higher memory usage and file sizes, these remain within practical bounds for modern systems.

At very small scene sizes, the chunked algorithm exhibits higher standard deviation due to scheduling overhead and operating system variability. While this overhead is visible in small scenes, it becomes negligible in large-scale settings where chunking provides clear, consistent advantages.

9.3 Comparison to Related Work

The presented algorithm distinguishes itself from prior voxel flood-fill and BFS methods by explicitly targeting parallel execution within Unity's CPU-based Job System. Unlike GPU-accelerated Voronoi or distance-transform techniques, which often sacrifice determinism, omit obstacles, or require specialized hardware, this work preserves exact obstacle-aware semantics while remaining portable across standard CPU architectures.

The results align with theoretical models of parallel BFS: speedup scales with available cores until memory bandwidth or synchronization overheads dominate. The observed fourfold speedup within the parallel section confirms the effectiveness of the parallelization strategy, though the total runtime impact is diminished by surrounding sequential physics checks. This highlights Unity's Job System as a viable platform for high-performance spatial algorithms, while also revealing its current limitations.

9.4 Scalability and Performance Characteristics

Overall, the parallel algorithm scales far more favorably than the sequential one, keeping growth factors consistently below the factors of increase in workload. In contrast, the sequential variant suffers from near exponential runtime growth. At voxel size 0.125, further scaling is computationally feasible, but memory usage emerges as the limiting factor. Since most chunks remain untouched in any given iteration, a dynamic loading and unloading system could mitigate memory pressure, though such functionality lies outside of the scope of this thesis.

A key performance factor is the Overhead Volume Ratio (OVR), which measures the redundant volume explored due to chunk boundaries. Smaller chunks reduce redundant processing but may incur parallel scheduling overhead, particularly at sizes such as 8 voxels, where workloads per thread are minimal. Larger chunks amortize scheduling costs but increase redundant processing. The data suggests that smaller chunk sizes can remain effective as long as physics-based collision detection dominates runtime and outweighs the costs of voxel exploration.

Physics overlap queries remain the single largest bottleneck, often exceeding 90% of total execution time. This severely constrains the benefits of parallelization: while the parallel flood-fill section shows a clear 4× speedup, the global runtime impact is marginal because physics checks remain single-threaded. Consequently, further improvements will likely depend more on reducing physics overhead—through parallelization, approximation, or caching—than on additional refinements to chunking alone.

Burst compilation proved to be quite impactful with consistently doubling the performance of affected sections, particularly those involving voxel exploration and data structure updates. Although Burst imposes restrictions on the architecture and only allows a subset of C#, its integration proved critical in ensuring computational efficiency.

9.5 Limitations and Trade-offs

The chunked algorithm introduces complexity in implementation and maintenance, making debugging and integration more demanding than with the sequential version. Memory consumption grows with chunk metadata, and extreme chunk sizes lead to either scheduling inefficiencies or increased redundant processing. Moreover, the solution is closely tied to Unity's parallel architecture; porting to engines such as Unreal would require substantial effort.

Runtime queries also reveal important trade-offs. With preloaded chunks, query times are constant and extremely fast, independent of voxel size or scene scale. However, on-demand loading significantly degrades performance, with queries scaling linearly rather than remaining constant. While this is partially mitigated in practice by repeated access keeping chunks in memory, the current file format prioritizes usability over minimal storage overhead, leaving room for future optimization.

9.6 Outlook

The results demonstrate that parallel chunked flood-fill is a scalable, production-ready solution for voxel-based spatial processing in Unity. Its scalability enables application to large and complex levels where sequential methods fail. However, meaningful further gains will require addressing the dominant cost of physics overlap queries. Once this bottleneck is reduced, factors such as chunk size and Overhead Volume Ratio will play a more prominent role in overall performance. Additionally, dynamic chunk loading strategies could extend scalability by alleviating memory constraints. Together, these directions define a clear path for advancing the algorithm beyond its current capabilities.

10 Conclusion

This thesis investigated how chunking and parallelization can improve the efficiency of the Space Foundation System (SFS), a high-level abstraction that represents environments as graphs of semantically meaningful locations. The sequential baseline, a frontier-based, obstacle-aware voxel flood fill, was found to scale poorly as scene sizes and resolutions increased, becoming a bottleneck for both editor-time generation and runtime queries.

To address this, the work makes two central contributions. First, it introduces a chunked parallel algorithm that partitions the voxel domain into independent regions, resolves chunk borders deterministically, and preserves semantic correctness relative to the sequential baseline. Second, it delivers a complete Unity implementation using the C# Job System and Burst Compiler, with compact voxel layouts, performance-conscious serialization, and compatibility with the existing SFS API. Together, these contributions demonstrate not only the conceptual scalability of chunking, but also its practical feasibility in a production environment.

The evaluation highlights a qualitative shift in system behavior. In the SFS generation for complex scenes, the chunked parallel algorithm consistently outperformed the sequential version, achieving substantial speedups that scale with scene size and resolution. In runtime queries, performance improvements are even more significant: instead of growing with total scene size, queries become primarily dependent on chunk size, yielding effectively constant-time responses when relevant chunks are memory-resident. This scalability enables the SFS to handle volumes exceeding hundreds of millions of voxels, making it viable for complex, high-resolution environments.

At the same time, the work reveals important trade-offs and limitations. Implementation complexity increases due to border handling and chunk synchronization, and memory overhead grows with chunk metadata. More critically, single-threaded physics overlap queries dominate total runtime in many scenarios, constraining overall scalability despite clear algorithmic improvements. These limitations point to concrete directions for future research, such as parallelizing or approximating physics checks and introducing dynamic chunk loading strategies to alleviate memory pressure.

In summary, this thesis demonstrates that chunking and parallelization are an important step in transforming the SFS from a conceptually elegant but limited approach into a scalable, production-ready tool. It shows that location graph creation times can be accelerated, runtime queries reduced to near-constant cost, and large-scale environments made practical. The central takeaway is the qualitative change in scaling behavior, which establishes a strong foundation for further optimization and broader adoption in real-world game development workflows.

11 Future Work

This chapter outlines targeted directions to further increase the performance and scalability of the chunked parallel algorithm for the Space Foundation System (SFS) presented in this thesis.

11.1 Reducing Physics-Query Cost

Creation-time benchmarks identified physics overlap queries as the dominant computational cost in many scenes. Although the current implementation already utilizes non-allocating overlap queries and caches overlap results, the overhead remains significant in complex or densely populated environments.

To mitigate this bottleneck within the constraints of the existing system, several viable strategies can be considered. A particularly promising direction is the introduction of a hierarchical spatial index—such as an SVO-like octree at the chunk level, or a bounding volume hierarchy (BVH) over colliders. These structures enable top-down overlap testing: large spatial volumes are tested first, and finer-grained tests are performed only where the broad-phase detects potential intersections.

This hierarchical approach dramatically reduces the number of fine-grained physics queries in scenes with sparse geometry, while preserving conservativeness to ensure no true overlaps are missed. Sparse Voxel Octrees (SVOs) and related data structures are well-established in the literature as efficient mechanisms for hierarchical culling and volumetric queries [40, 41].

11.2 Runtime Storage Format and Identifier Policy

For efficient runtime usage, the per-chunk voxel and border representations should be optimized for both load/unload throughput and compactness. In particular, the border voxel data of anchors should be stored using a custom binary format. This format encodes per-voxel information using compact bitfields, applies delta encoding where beneficial, and tightly packs indices to reduce both on-disk size and decompression overhead.

By contrast, the location graph—which encodes the high-level structure of anchor connectivity—may remain a human-readable ScriptableObject. Its size is negligible compared to the bulk border voxel data, and preserving readability in this component facilitates debugging, versioning, and integration with tooling.

Additionally, replacing string-based identifiers with integer-based ones further improves both storage efficiency and computational performance. To ensure compatibility with the existing API, this thesis adopts a dual-representation approach: bidirectional mappings between string and integer identifiers are introduced. This allows for consistent API use while enabling high performance and memory efficiency during SFS creation.

11.3 Memory and Creation-Time Scalability

Our scalability benchmarks revealed that a primary bottleneck lies in memory consumption, rather than execution time. To address this limitation, a practical next step is to implement a streaming or eviction policy for chunks during SFS creation. Specifically, once a chunk and all of its neighboring chunks—those that could influence its final state—have been finalized, the chunk can be serialized to disk and removed from memory.

A least-recently-used (LRU) container, or a cost-aware variant thereof, offers a straightforward and effective heuristic for bounding peak memory usage. This approach allows the system to maintain only the most recently relevant chunks in memory, which is particularly beneficial for ensuring fast reconciliation along chunk borders without compromising overall scalability.

11.4 GPU and Heterogeneous Compute

Although GPUs offer significant parallel compute capacity, the flood-fill algorithm used for SFS generation poses challenges for direct GPU mapping. Its correctness depends on deterministic, seed-based propagation semantics, which align more naturally with sequential or tightly synchronized execution.

However, the chunked architecture presents opportunities for heterogeneous compute. Chunk-local processing enables partial offloading to GPU kernels—either by processing multiple chunks in parallel or accelerating intra-chunk voxel operations—provided that a deterministic reconciliation step handles cross-chunk borders.

Adapting the algorithm to GPU constraints is non-trivial. GPUs favor data-parallel, lock-free kernels and exhibit different memory and branching behavior than CPUs. Preserving determinism across hardware boundaries will likely require a hybrid CPU/GPU approach and careful coordination between stages.

In summary, GPU acceleration holds promise for improving throughput, but realizing this potential demands careful architectural adaptation to reconcile the algorithm's inherently non-parallel characteristics with the strengths of GPU execution.

List of Figures

List of Tables

Bibliography

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[2] Ronald C Arkin. Path planning for a vision-based autonomous robot. In *Mobile robots I*, volume 727, pages 240–250. SPIE, 1987.

[3] Daniel Arribas-Bel and Martin Fleischmann. Spatial signatures-understanding (urban) spaces through form and function. *Habitat International*, 128:102641, 2022.

[4] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM computing surveys (CSUR)*, 23(3):345–405, 1991.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[7] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[8] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.

[9] M Akmal Butt and Petros Maragos. Optimum design of chamfer distance transforms. *IEEE Transactions on Image Processing*, 7(10):1477–1484, 1998.

[10] Carlos Couder-Castañeda, Mauricio Orozco-del Castillo, Diego Padilla-Perez, and Isaac Medina. A parallel texture-based region-growing algorithm implemented in openmp. *Scientific Reports*, 15(1):5563, 2025.

[11] Mark De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Prasad M Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F Naughton. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 259–270, 1998.

[14] Sebastian Dorn, Nicola Wolpert, and Elmar Schömer. Voxel-based general voronoi diagram for complex data with application on motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 137–143. IEEE, 2020.

[15] Daniel Dyrda and Claudio Belloni. Space foundation system: An approach to spatial problems in games. In *2024 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2024.

[16] Godot Engine. Godot documentation: Area2d and area3d nodes. `https://docs.`

godotengine.org/, 2025.

[17] Xuhui Fan, Bin Li, and Scott Sisson. The binary space partitioning-tree process. In *International Conference on Artificial Intelligence and Statistics*, pages 1859–1867. PMLR, 2018.

[18] Pedro F Felzenszwalb and Daniel P Huttenlocher. E ciently computing a good segmentation. In *DARPA Image Understanding Workshop*. Citeseer, 1998.

[19] Steven Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the second annual symposium on Computational geometry*, pages 313–322, 1986.

[20] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[21] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, 1980.

[22] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*, pages 121–131, 2019.

[23] Epic Games. Unreal engine documentation: Gameplay tags. https://docs.unrealengine.com/, 2025.

[24] Epic Games. Unreal engine documentation: Smart objects system. https://docs.unrealengine.com/, 2025.

[25] Epic Games. Unreal engine documentation: Trigger volumes and collision. https://docs.unrealengine.com/, 2025.

[26] A. Gelineau and M. Leblanc. Dna: Ubisoft's data analytics platform for game telemetry. In *GDC 2019*, 2019.

[27] Articy Software GmbH. Articy:draft integration with unreal engine. https://www.articy.com/, 2025.

[28] Articy Software GmbH. Showcase: Tropico 6. https://www.articy.com/en/showcase/tropico-6/, 2025.

[29] Rhys Goldstein, Kean Walmsley, Nigel Morris, and Alexander Tessier. Algorithms for voxel-based architectural space analysis. In *2023 Annual Modeling and Simulation Conference (ANNSIM)*, pages 508–519. IEEE, 2023.

[30] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.

[31] Roland Gruber, Stefan Gerth, Joelle Claußen, Norbert Wörlein, Norman Uhlmann, and Thomas Wittenberg. Exploring flood filling networks for instance segmentation of xxl-volumetric and bulk material ct data. *Journal of Nondestructive Evaluation*, 40(1):1, 2021.

[32] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[33] Hsien-Hsi Hsieh and Wen-Kai Tai. A simple gpu-based approach for 3d voronoi diagram construction and visualization. *Simulation modelling practice and theory*, 13(8):681–692, 2005.

[34] Intel Corporation. Improve performance with vectorization. https://www.

intel.com/content/www/us/en/developer/articles/technical/improve-performance-with-vectorization.html, 2021. Accessed: 2025-08-09.

[35] Intel Corporation. Intel® core™ processor family, 2024. Accessed: 2025-08-14.

[36] Michał Januszewski, Jörgen Kornfeld, Peter H Li, Art Pope, Tim Blakely, Larry Lindsey, Jeremy Maitin-Shepard, Mike Tyka, Winfried Denk, and Viren Jain. High-precision automated reconstruction of neurons with flood-filling networks. *Nature methods*, 15(8):605–610, 2018.

[37] Michał Januszewski, Jeremy Maitin-Shepard, Peter Li, Jörgen Kornfeld, Winfried Denk, and Viren Jain. Flood-filling networks. *arXiv preprint arXiv:1611.00421*, 2016.

[38] Mincheol Kim, Chanyang Seo, Taehoon Ahn, and Hee-Kap Ahn. Farthest-point voronoi diagrams in the presence of rectangular obstacles. *Algorithmica*, 85(8):2214–2237, 2023.

[39] Secret Lab. Yarn spinner: Dialogue system for unity. https://yarnspinner.dev/, 2025.

[40] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 55–63, 2010.

[41] Samuli Laine and Tero Karras. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*, 2(6):1–30, 2010.

[42] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*, pages 347–353. 1998.

[43] Inkle Ltd. Ink: Narrative scripting language and inky editor. https://www.inklestudios.com/ink/, 2025.

[44] Kevin Lynch. *The image of the city*. MIT press, 1964.

[45] Sean Patrick Mauch. *Efficient algorithms for solving static Hamilton-Jacobi equations*. California Institute of Technology, 2003.

[46] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[47] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[48] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[49] Seongjin Park, Jeongjin Lee, Hyunna Lee, Juneseuk Shin, Jinwook Seo, Kyoung Ho Lee, Yeong-Gil Shin, and Bohyoung Kim. Parallelized seeded region growing using cuda. *Computational and mathematical methods in medicine*, 2014(1):856453, 2014.

[50] Hayder Radha, Martin Vetterli, and Riccardo Leonardi. Image compression using binary space partitioning trees. *IEEE transactions on image processing*, 5(12):1610–1624, 1996.

[51] Gabriel Rivera and Chau-Wen Tseng. Locality optimizations for multi-level caches. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 2–es, 1999.

[52] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, 2006.

[53] Guodong Rong and Tiow-Seng Tan. Variants of jump flooding algorithm for computing discrete voronoi diagrams. In *4th international symposium on voronoi diagrams in science and engineering (ISVD 2007)*, pages 176–181. IEEE, 2007.

[54] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. MIT Press, 2004.

[55] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC press, 2008.

[56] Markus Schütz, Katharina Krösl, and Michael Wimmer. Real-time continuous level of detail rendering of point clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 103–110. IEEE, 2019.

[57] Noor Shaker, Julian Togelius, and Mark J Nelson. Procedural content generation in games. 2016.

[58] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. Cache conscious algorithms for relational query processing. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1994.

[59] Yossi Shiloach and Uzi Vishkin. An o (n2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[60] DE Singh, DB Heras, and FF Rivera. Parallel seeded region growing algorithm. *VIII Simposium Nacional de Reconocimiento de Formas y Análisis de Imágenes, Bilbao, Spain*, 1999.

[61] Avneesh Sud, Naga Govindaraju, Russell Gayle, and Dinesh Manocha. Interactive 3d distance field computation using linear factorization. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 117–124, 2006.

[62] Avneesh Sud, Naga Govindaraju, and Dinesh Manocha. Interactive computation of discrete generalized voronoi diagrams using range culling. In *Proc. International Symposium on Voronoi Diagrams in Science and Engineering*, 2005.

[63] Kelvin Sung and Peter Shirley. Ray tracing with the bsp tree. In *Graphics Gems III (IBM Version)*, pages 271–274. Elsevier, 1992.

[64] Unity Technologies. Unity manual: Collider and trigger events. `https://docs.unity.com/`, 2025.

[65] Unity Technologies. Burst. `https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/optimization-overview.html`, n.d. Accessed: 2025-07-05.

[66] Unity Technologies. Data-oriented technology stack (dots). `https://unity.com/dots`, n.d. Accessed: 2025-07-05.

[67] Unity Technologies. Entity component system (ecs). `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html`, n.d. Accessed: 2025-07-05.

[68] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4):449–

460, 2014.

[69] William C Thibault and Bruce F Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 153–162, 1987.

[70] Unity Technologies. C# job system overview. `https://docs.unity3d.com/2018.3/Documentation/Manual/JobSystemOverview.html`, 2018. Accessed: 2025-08-09.

[71] Unity Technologies. Write multithreaded code with the job system. `https://docs.unity3d.com/Manual/job-system.html`, 2025. Accessed: 2025-08-09.

[72] Wouter Van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*, pages 91–100, 2016.

[73] Marek Vinkler, Jiri Bittner, and Vlastimil Havran. Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of High Performance Graphics*, pages 1–8. 2017.

[74] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

[75] Fan Yang, You Li, Mingliang Che, Shihua Wang, Yingli Wang, Jiyi Zhang, Xinliang Cao, and Chi Zhang. The polygonal 3d layout reconstruction of an indoor environment via voxel-based room segmentation and space partition. *ISPRS International Journal of Geo-Information*, 11(10):530, 2022.

Appendix

1 Execution Time Comparison

Alg.	Scene	Mean	Median	Min	Max	Std. Dev.
Seq	House	5.948	5.020	4.925	17.75	2.833
Chu	House	6.191	3.981	3.918	46.46	9.482
Seq	Villa	25.94	25.03	23.88	34.48	2.485
Chu	Villa	31.57	29.36	28.93	70.54	9.188
Seq	Region	1 724	1 716	1 691	1 861	36.41
Chu	Region	536.0	528.7	522.0	681.9	34.45
Seq	Mapper's Peak	3 062	3 050	3 009	3 153	45.15
Chu	Mapper's Peak	1 356	1 355	1 350	1 368	4.448

Table 1: Detailed SFS creation times (ms) for different scenes (voxel size 1.0, chunk size 16).

Alg.	Scene	Time	Speedup	Voxels	Iteration
Seq	House	19.17	1.896	6 682	–
Chu	House	10.11	1.896	32 770	6
Seq	Villa	162.1	1.692	58 260	–
Chu	Villa	95.81	1.692	221 200	14
Seq	Region	43 720	14.67	3 132 000	–
Chu	Region	2 979	14.67	5 063 000	15
Seq	Mapper's Peak	72 510	10.31	3 891 000	–
Chu	Mapper's Peak	7 044	10.31	5 661 000	26

Table 2: Mean SFS creation times (ms) and corresponding speedups for different scenes, along with processed voxel counts and algorithm iterations (voxel size 0.5, chunk size 16).

Alg.	Scene	Mean	Median	Min	Max	Std. Dev.
Seq	House	19.17	18.63	18.38	25.26	1.568
Chu	House	10.11	10.02	9.552	11.47	0.470 0
Seq	Villa	162.1	161.9	157.5	168.2	3.037
Chu	Villa	95.81	95.66	93.87	99.59	1.559
Seq	Region	43 720	43 650	43 490	44 440	242.4
Chu	Region	2 979	2 968	2 941	3 110	41.70
Seq	Mapper's Peak	72 510	72 510	72 390	72 740	432.5
Chu	Mapper's Peak	7 044	7 035	6 990	7 208	42.90

Table 3: Detailed SFS creation times (ms) for different scenes (voxel size 0.5, chunk size 16).

Alg.	Scene	Mean	Median	Min	Max	Std. Dev.
Seq	House	114.1	111.1	108.6	140.7	7.352
Chu	House	57.10	57.25	55.32	59.08	1.011
Seq	Villa	4 463	4 362	4 273	5 015	194.9
Chu	Villa	363.6	363.4	359.3	371.3	2.954
Seq	Region	1 435 000	1 435 000	1 427 000	1 452 000	13 580
Chu	Region	19 970	19 930	19 750	20 390	165.9
Seq	Mapper's Peak	2 343 000	2 343 000	2 325 000	2 361 000	25 580
Chu	Mapper's Peak	43 180	43 030	42 770	44 570	499.0

Table 4: Detailed SFS creation times (ms) for different scenes (voxel size 0.25, chunk size 16).

2 Scalability Stress Test

Chunk Size	Voxel Size	Mean	Median	Min	Max	Std. Dev.
4	1.0	1 026	1 006	939.0	1 118	82.74
8	1.0	476.7	472.5	466.5	545.5	17.23
16	1.0	536.0	528.7	522.0	681.9	34.45
32	1.0	534.6	530.9	525.8	581.5	12.84
64	1.0	545.0	545.3	536.0	556.2	5.480
128	1.0	2 123	2 121	2 099	2 185	20.27
8	0.5	3 196	3 169	3 105	3 417	81.85
16	0.5	2 979	2 968	2 941	3 110	41.70
32	0.5	3 792	3 799	3 740	3 860	38.67
64	0.5	4 030	3 962	3 906	4 768	191.1

Table 5: Detailed execution time statistics (in milliseconds) for the scene Region with variations in chunk size.

3 Runtime Queries

Voxel Size	Algorithm	Mean	File Size	Voxels
1.0	Sequential	12.26	0.033 60	679.0
1.0	Chunked-Preloaded	0.940 7	0.179 9	16.38
1.0	Chunked-OnDemand	496.4	0.179 9	16.38
0.5	Sequential	12.93	0.247 3	6.682
0.5	Chunked-Preloaded	0.801 1	0.454 5	32.77
0.5	Chunked-OnDemand	880.9	0.454 5	32.77
0.25	Sequential	15.92	1.217	51.44
0.25	Chunked-Preloaded	0.797 0	2.520	196.6
0.25	Chunked-OnDemand	1 640	2.520	196.6

Table 6: Realtime query execution time in microseconds for the *House* scene. Comparison between Sequential algorithm and Chunked Algorithm with preloaded chunks in memory and on-demand loading. File sizes in MB.

Voxel Size	Algorithm	Mean	File Size	Voxels
1.0	Sequential	0.306 9	0.448 5	9.290
1.0	Chunked-Preloaded	1.011	0.908 5	73.73
1.0	Chunked-OnDemand	869.7	0.908 5	73.73
0.5	Sequential	1.177	2.044	58.26
0.5	Chunked-Preloaded	0.969 2	3.203	221.2
0.5	Chunked-OnDemand	1 238	3.203	221.2
0.25	Sequential	5.081	9.307	475.5
0.25	Chunked-Preloaded	0.984 9	12.19	770.0
0.25	Chunked-OnDemand	1 251	12.19	770.0

Table 7: Realtime query execution time in microseconds for the *Villa* scene. Comparison between Sequential algorithm and Chunked Algorithm with preloaded chunks in memory and on-demand loading. File sizes in MB.

Voxel Size	Algorithm	Mean	File Size	Voxels
1.0	Sequential	10.78	7.992	397 100
1.0	Chunked-Preloaded	1.084	13.37	974 800
1.0	Chunked-OnDemand	3 881	13.37	974 800
0.5	Sequential	85.19	34.68	3 132 000
0.5	Chunked-Preloaded	1.089	67.85	5 063 000
0.5	Chunked-OnDemand	5 572	67.85	5 063 000
0.25	Sequential	657.2	154.8	25 540 000
0.25	Chunked-Preloaded	1.058	402.5	32 060 000
0.25	Chunked-OnDemand	26 530	402.5	32 060 000

Table 8: Realtime query execution time in microseconds for the *Region* scene. Comparison between Sequential algorithm and Chunked Algorithm with preloaded chunks in memory and on-demand loading. File sizes in MB.