FAST GPU GENERATION OF DISTANCE FIELDS FROM A VOXEL GRID

by

NICOLAS FEDOR
URN: 6683787

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2025

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: Joey Sik Chun Lam

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Nicolas Fedor
May 2025

Abstract

This dissertation investigates the feasibility of generating distance fields in real-time on modern GPUs, addressing a critical performance bottleneck in dynamic graphical applications. While distance fields offer exceptional rendering efficiency, their adoption in interactive environments has been limited by the computational cost of their generation when geometry changes.

The research employs a systematic comparative analysis of distance field generation techniques including brute force computation, spatial chunking, the Fast Iterative Method, and the Jump Flooding Algorithm. Performance evaluation criteria encompass computational throughput, memory requirements, and output quality across diverse test cases. Results demonstrate that existing single-method approaches present inherent limitations for real-time applications. In response, this dissertation proposes a novel hybrid algorithm that strategically combines the convergence characteristics of the Fast Iterative Method with the parallel efficiency of the Jump Flooding Algorithm, enhanced by an adaptive level-of-detail mechanism that optimizes computational resources.

Experimental validation confirms that the proposed approach achieves significant performance improvements over traditional methods while maintaining necessary accuracy for real-time applications. The findings establish that dynamic distance field generation is indeed viable for interactive applications, offering a more efficient alternative to other real-time volumetric representation techniques. This research contributes to the field by providing a practical, implementation-ready solution for incorporating distance fields in applications where geometry undergoes frequent modification, potentially expanding their utility in interactive simulations, procedural content generation, and advanced rendering pipelines.

The implementation code and demonstration application used in this research are available at:

https://github.com/Fedron/gpu-voxel-fields

Acknowledgements

I would like to express my sincere gratitude to my supervisor for their invaluable guidance throughout this research process. Their thoughtful reviews, strategic direction on content inclusion, and continued encouragement to aim for excellence were instrumental in shaping this dissertation.

I am deeply grateful to my parents for their unwavering support and assistance in reviewing my work. Their perspectives helped ensure the research remained comprehensible and visually effective, particularly for readers approaching the material from diverse backgrounds.

This dissertation represents not only my individual effort but also reflects the collective support and expertise of these individuals who contributed significantly to its completion.

Contents

List of Figures

List of Tables

Abbreviations

SVO	Sparse Voxel Octree
JFA	Jump Flooding Algorithm
CPU	Central Processing Unit
GPU	Graphics Processing Unit
DAG	Directed Acyclic Graphs
SVDAG	Sparse Voxel Directed Acyclic Graph
FPS	Frames Per Second
DDA	Digital Differential Analyzer
FMM	Fast Marching Method
FIM	Fast Iterative Method
JFA	Jump Flooding Algorithm

Chapter 1

Introduction

In recent years, real-time computer graphics applications have increasingly adopted distance fields as a fundamental representation for rendering and physics simulations (Jones & Satherley 2001). Distance fields, which encode the minimum distance from any point to the nearest surface, provide an elegant solution for various graphics operations including collision detection (Fuhrmann, Sobotka & Groß 2003), soft shadows (Tan, Chua, Koh & Bhojan 2022), and ambient occlusion (Wright 2015). While techniques exist for generating distance fields in real-time from a triangle mesh (Kramer 2015), techniques covering distance field generation from discrete voxel data are uncommon.

1.1 Problem Statement

Current approaches to distance field generation from voxel grids present various tradeoffs that limit their effectiveness in dynamic scenes. The Jump Flooding Algorithm (JFA) (Rong & Tan 2006, Rong & Tan 2007, Wang, Ino & Ke 2023), while efficient for parallel computation, introduces accuracy issues particularly at larger distances from surfaces and near feature edges. Scan, or prefix sum-based, approaches (Erleben & Dohlmann 2008) provide accurate results but suffer from inherent sequential dependencies that limit GPU parallelization. Wavefront propagation methods can efficiently update local regions but may struggle with concurrent updates in complex scenes (Teodoro, Pan, Kurc, Kong, Cooper & Saltz 2013).

Common optimization strategies, such as spatial partitioning into smaller chunks for localized updates (Naylor 1992), introduce their own challenges including boundary artifacts and increased

memory management overhead. While these techniques work well in isolation for specific use cases, there remains a fundamental gap in solutions that can handle arbitrary dynamic scene modifications while maintaining both accuracy and performance. This research investigates whether a novel hybrid approach—combining elements of existing techniques or developing new algorithmic patterns—could better address these challenges.

1.2 Aims and Objectives

This research aims to develop and evaluate novel GPU-based techniques for rapid distance field generation from voxel grid representations. The primary objectives are:

- To analyze and classify existing approaches to distance field generation, with particular focus on GPU-accelerated methods.
- To develop new algorithms that optimize the conversion process from voxel grids to distance fields.
- To implement and validate these algorithms on modern GPU architectures.
- To establish a comprehensive comparison framework for evaluating different distance field generation techniques.

1.2.1 Performance Metrics

The evaluation of the proposed methods will be conducted against sparse voxel octree implementations, which currently represent the state-of-the-art in many graphics applications. Key performance metrics include:

1. Computation time for initial distance field generation.
2. Memory consumption during generation and storage.
3. Update latency for localized geometric changes.
4. Scalability with increasing voxel grid resolution.
5. Accuracy of distance field values compared to analytical solutions.
6. GPU resource utilization, including memory bandwidth and compute occupancy.

1.3 Scope and Limitations

While this research addresses the core challenge of distance field generation, several related aspects fall outside its scope:

- The optimization of ray marching techniques for distance field rendering.
- The development of new compression methods for distance field storage.
- The optimization of the underlying renderer, which will include features like:
  - Memory management between the CPU and GPU.
  - Sychronization with the windowing framework.
  - Optimizing presentation of ray marching output image.

The focus remains specifically on the GPU-based generation process and its performance characteristics in dynamic scenarios. The research assumes access to modern GPU hardware and primarily targets real-time graphics applications where frequent distance field updates are required.

A sample voxel painting graphics application will be implemented that will serve as a demonstration and benchmark for real-time performance of the distance filed regeneration. For benchmarking the application will be slightly altered to allow for automatic updates to the world, and reproducibility of test results; a detailed explanation of this application can be found in Section 3.4.

Chapter 2

Literature Review

Traditional real-time rendering has predominantly relied on triangle meshes as the fundamental primitive, with modern graphics hardware specifically optimized for rasterizing triangles efficiently (Akenine-Moller, Haines & Hoffman 2019), this can be seen in Figure 2.1.

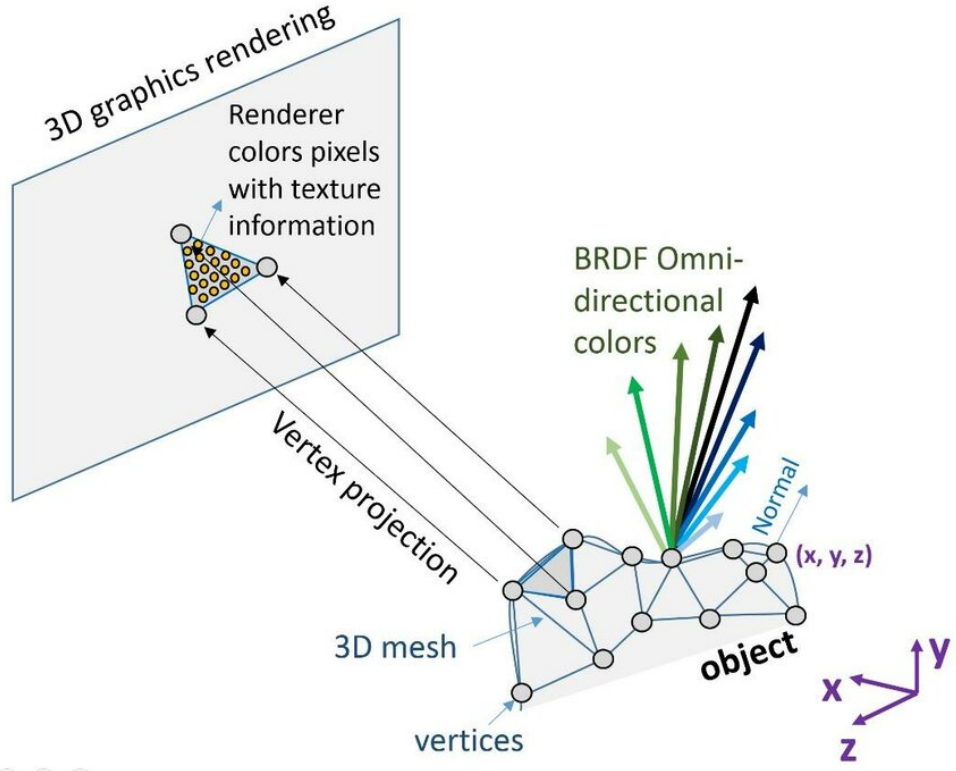


Figure 2.1: Illustration of how a collection of triangles get rasterized onto a screen as demonstrated by Lafruit et al. 2016

While this approach remains widespread, recent advances in hardware-accelerated ray tracing, particularly with NVIDIA's RTX series (2018) and AMD's RDNA2 architecture (2020), have enabled real-time ray tracing in commercial applications. Games like "Cyberpunk 2077" and "Metro Exodus Enhanced Edition" demonstrate that hybrid approaches combining rasterization and ray tracing can achieve photorealistic effects such as global illumination and accurate reflections at interactive frame rates (Keller, Viitanen, Barré-Brisebois, Schied & McGuire 2019). However, both rasterization and ray tracing face similar challenges when representing highly detailed or volumetric content, leading to the exploration of alternative representations.

Ray tracing functions by "shooting" rays out of the camera and into the world, these rays are then tested for intersections with objects in the world; rays are able to "bounce" around in the world, reflect in objects, and cast shadows. These rays are created in such a way that one, or more, rays are produced for each pixel on the screen that will allow a given ray to produce a pixel color as seen in Figure 2.2.

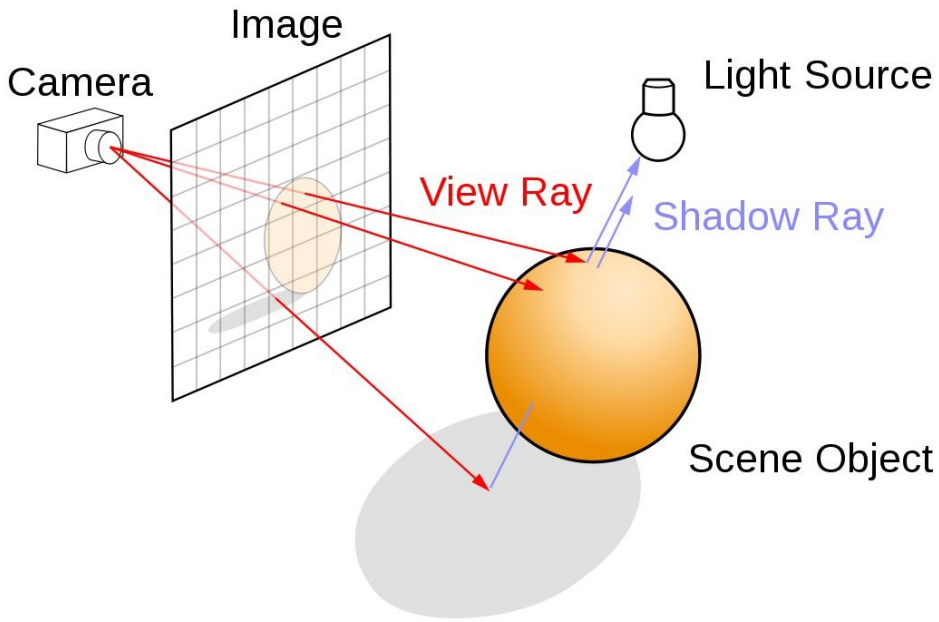


Figure 2.2: Visualization of how ray-casting is used to rasterize a 3D world. Source: Wikipedia

2.1 Voxel-Based Representations and Optimization

2.1.1 Sparse Voxel Octrees

Sparse Voxel Octrees (SVOs) have emerged as a powerful solution for managing large-scale voxel worlds (Crassin, Neyret, Lefebvre & Eisemann 2009). Crassin et al. (2009) introduced GigaVoxels, a groundbreaking approach that demonstrated efficient rendering of highly detailed voxel scenes through a hierarchical structure and streaming. Their work showed that SVOs could effectively compress empty space while maintaining quick traversal times for ray casting. Building on this foundation, Laine and Karras (2010) developed an efficient sparse voxel octree (Laine & Karras 2010) implementation that improved upon previous approaches by introducing a novel node structure and traversal algorithm. Their method significantly reduced memory requirements while maintaining high rendering performance, making it particularly suitable for static scenes with complex geometry. Figure 2.3 shows how a 3D voxel world could be represented as an octree, where the world is recursively divided into smaller and more detailed nodes; only nodes that encode information are divided into smaller nodes thus saving on memory by not explicitly storing information about empty nodes at the highest level of detail.

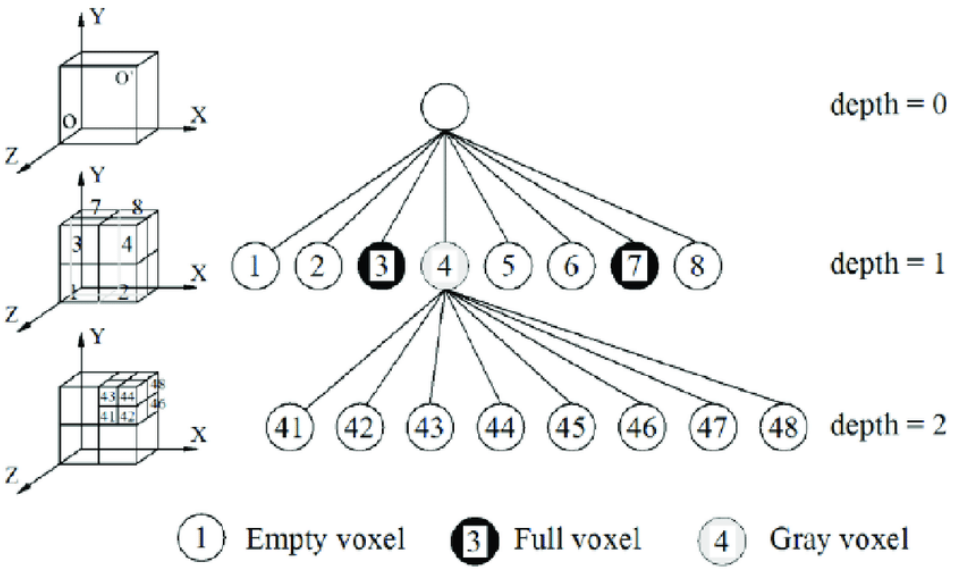


Figure 2.3: Illustration of the structure of a Sparse Voxel Octree, nodes with more details have additional subdivisions. (Truong-Hong & Laefer 2014)

2.1.2 Compression and Optimization Techniques

Several researchers have explored various compression techniques to further optimize voxel storage. Kämpe et al. (2013) introduced directed acyclic graphs (DAGs) for voxel scenes (Kämpe, Sintorn & Assarsson 2013), achieving compression ratios of up to 50:1 compared to standard SVOs while maintaining real-time rendering capabilities. This approach proved particularly effective for architectural and synthetic scenes with repeated structures as the repeated structures only needed to be encoded once with other occurrences only storing references to the encoded structure as can be seen in Figure 2.4

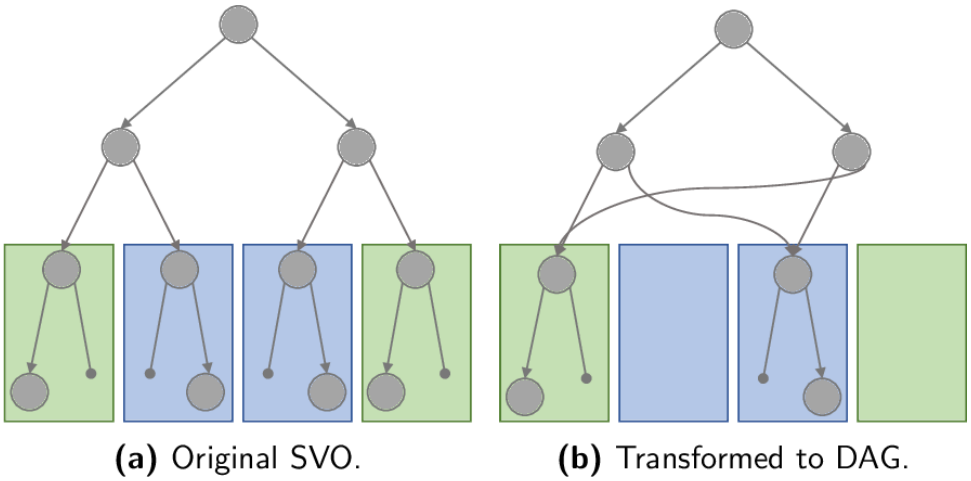


Figure 2.4: Comparison of an octree versus the compressed format of a SVDAG, illustrated as a quad tree for brevity. (Dolonius 2018)

2.1.3 Challenges in Dynamic Scenes

The primary challenge in dynamic voxel environments lies in maintaining data structures that can efficiently support modifications. Updating traditional SVOs in real-time presents significant computational overhead, as changes often require rebuilding portions of the tree structure. Pan (2021) explored a novel technique for merging SVOs and dynamically creating nodes where updates are needed (Pan 2021), while this showcases SVOs have the potential to support large dynamic scenes, the results show that real-time updates, such as in a video game application, are hard to achieve.

2.2 Distance Fields as an Alternative Representation

Distance fields have gained attention as an alternative to direct voxel storage, offering several advantages for both rendering and collision detection. Several recent works have demonstrated the advantages of using distance fields for real-time rendering of implicit surfaces (Hadji-Kyriacou & Arandjelović 2021) and function grids (Söderlund, Evans & Akenine-Möller 2022). Distance fields are a dense data structure that store the distance to the closest surface for a given point, this can be seen in Figure 2.5

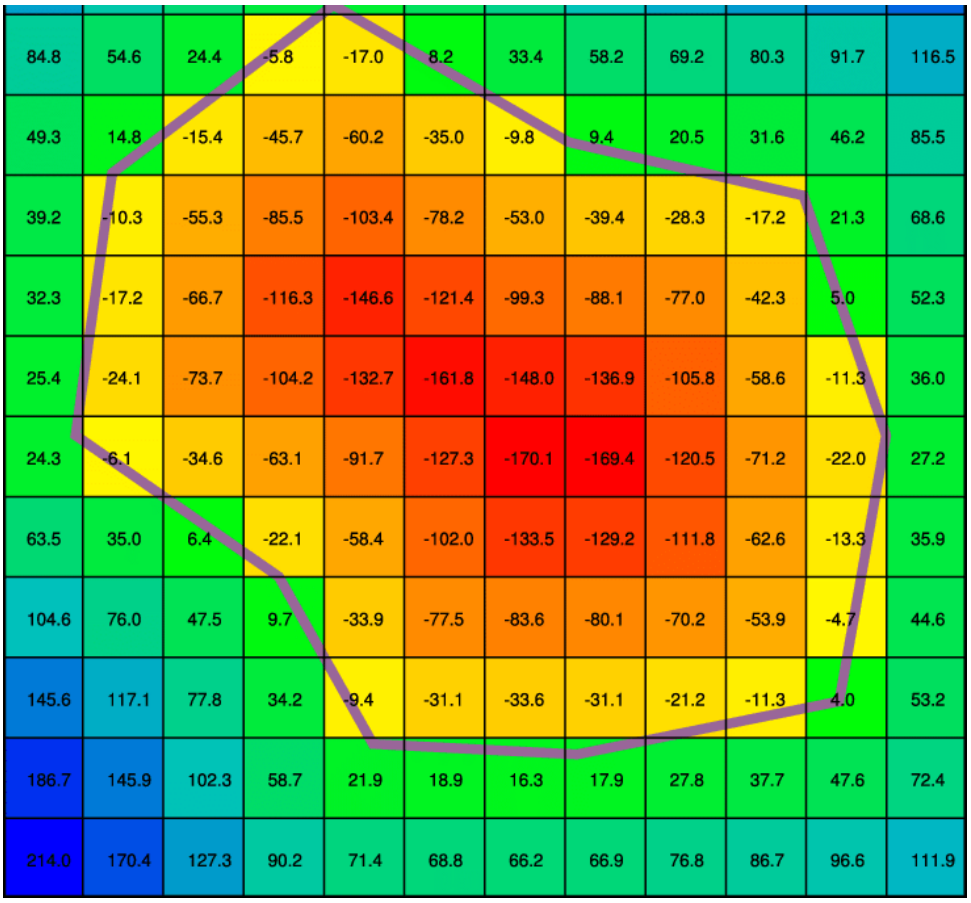


Figure 2.5: Illustration of a discrete signed distance field grid. Negatives values indicate a cell inside the shape, positive values indicate a cell outside the shape.

Ray-marching through a distance field allows you to advance the ray forward by an amount that is guaranteed not to skip over any features in the world. As seen in Figure 2.6, the ray can be advanced in variably sized steps while ensuring that the ray does not skip through a feature, something that could happen if fixed-size steps were used.

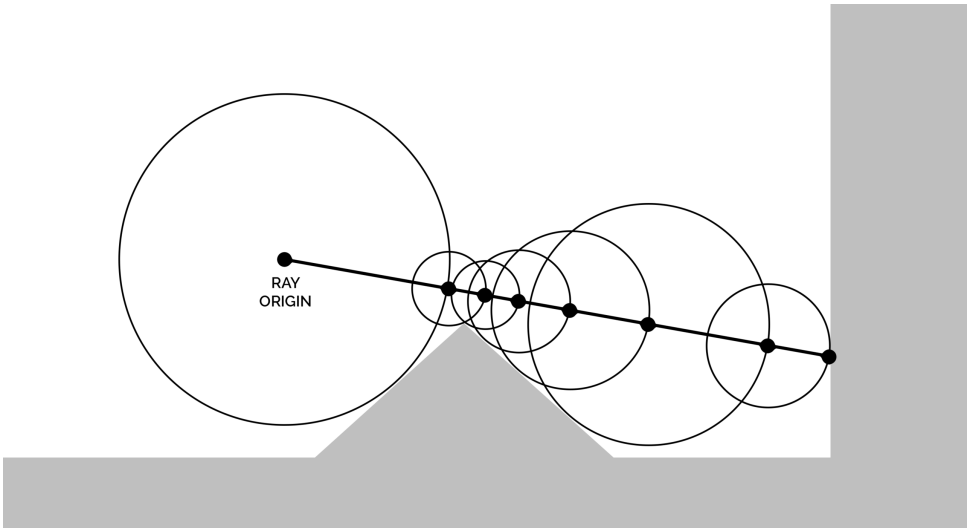


Figure 2.6: Illustration of ray marching using a distance field to advance the ray by a variable amount.

2.2.1 Real-Time Generation and Updates

The challenge of generating and updating distance fields in real-time remains an active area of research. Techniques such as Jump Flooding (Rong and Tan, 2006) provide fast approximate solutions but suffer from accuracy issues (Rong & Tan 2006, Rong & Tan 2007); improvements to Jump Flooding are being researched that allow for it to be used in dynamic contexts where recalculation of distance fields is needed (Stevenson & Navarro 2022).

Chapter 3

Methodology

The research methodology focuses on developing a dynamic distance field generation system using Vulkan, a low-level graphics API that provides precise control over GPU resources and computation. The system is designed to address the challenges of efficient distance field generation and rendering in dynamic voxel-based environments.

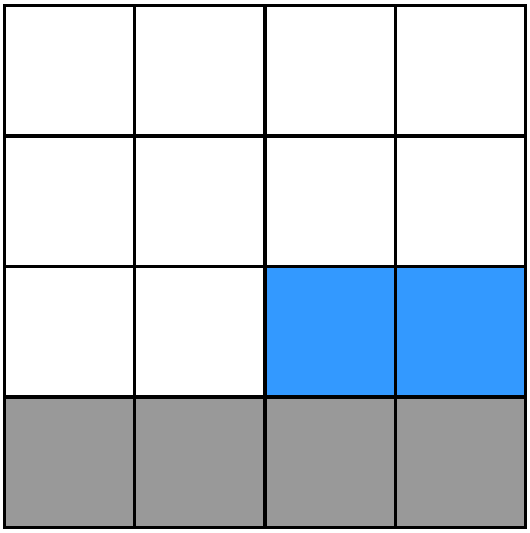
3.1 World Representation

The world is stored in a GPU buffer with host and device accessibility. This design choice prioritizes flexibility in world modification while minimizing performance overhead. Unlike traditional rendering approaches, the world buffer is not directly rendered, which mitigates potential performance penalties associated with host-visible memory. The choice of host-visible memory means that the host is able to update the world buffer as needed, and the updates will be visible to the device as well reducing complexity in staging buffers.

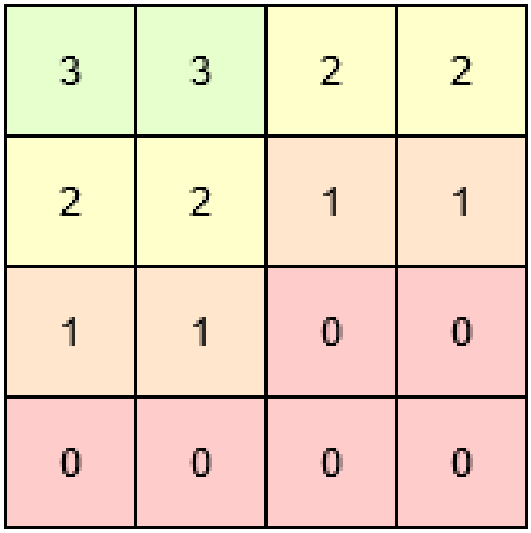
The world buffer will be stored in an uncompressed and dense format; this means each voxel in the world will be present in the world buffer with all of it's associated data. In this case a voxel will be a 32-bit unsigned integer, a value of 0 indicates an "air" voxel that should not be visible when rendered, while all other values indicate some form of solid voxel.

3.2 Distance Field Computation

The computation of a distance field, given a voxel grid, is the primary focus of this paper. To accomplish this a compute shader is implemented that will output a buffer containing the discrete distance field grid for a corresponding input voxel grid. This implementation is what will change throughout this paper as new algorithms and optimizations are added. The distance field will contain the Manhattan distance to the nearest solid voxel, this is important for accurate ray marching of the distance field 3.3. The relation between the voxel grid, and its corresponding discrete distance field can be seen in Figure 3.1.



(a) A 2D representation of the world. Empty voxels are indicated by white cells.



(b) The Manhattan discrete distance field representation of the world in 3.1a.

Figure 3.1: Illustration of the relation between a raw representation of a world, and its corresponding discrete distance field.

The computation of a distance field should not occur every frame as that would negatively impact the frame rate of an application. Instead, the CPU will update the world state to "dirty" to indicate that the distance field needs to be re-generated to reflect the newest state of the world. The workflow for this can be seen in Figure 3.2

To facilitate voxels having colors, colour information is encoded into the distance field. The distance field buffer will be a 1-dimensional unsigned integer array. The highest 8 bits define

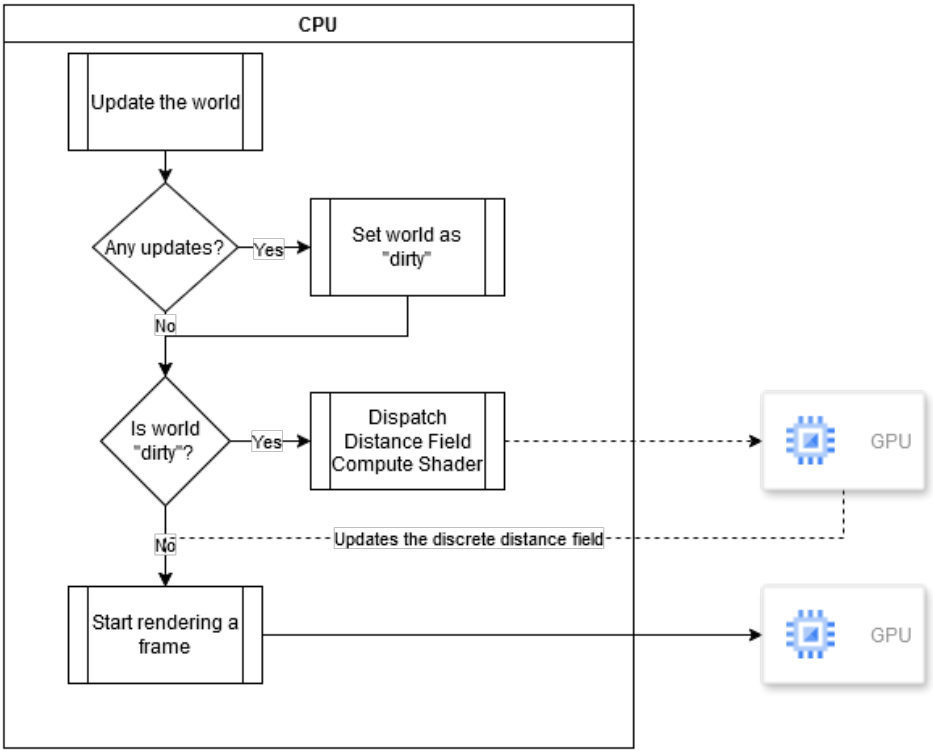


Figure 3.2: Illustration of the computational workflow required in deciding when to update the discrete distance field of a world.

the distance to the nearest solid voxel using the Manhattan distance, the lowest 8 bits define the colour of the voxel in a compressed RGB332 format; voxel colours are hard-coded into the distance field as can be seen in Figure 3.3.

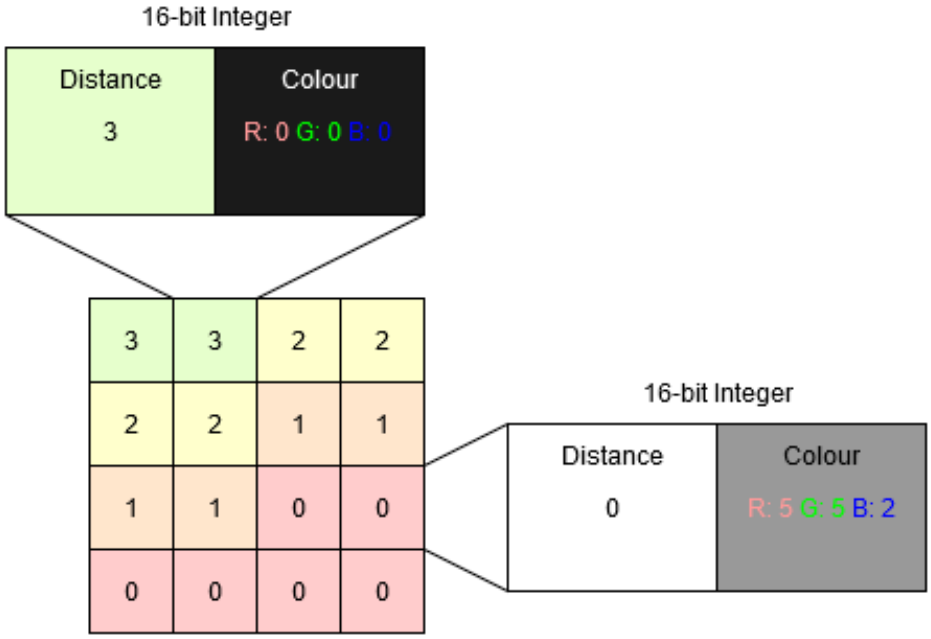


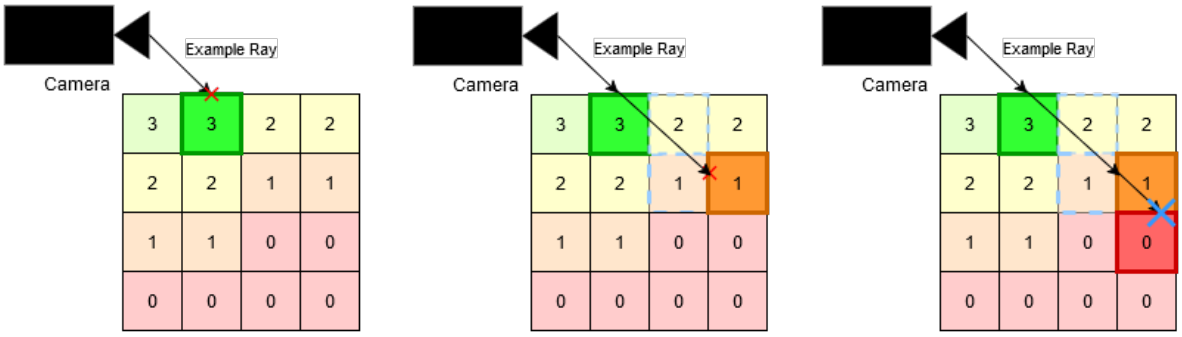
Figure 3.3: Based on the same distance field as in 3.1b, the underlying integer of an empty and solid voxel are shown.

3.3 Rendering and Ray Marching

The rendering is handled by a ray marching compute shader; the distance field is the only input to this shader. The ray marcher utilizes a digital differential analyzer to traverse through a voxel grid quickly (Amanatides et al. 1987). This approach ensures a ray traversing the distance field grid will traverse each voxel along the ray, as other methods like sphere marching could result in artifacts due to a ray "missing" a voxel, when using the Manhattan distance; the steps a ray takes through the world can be seen in Figure 3.4.

The ray marching compute shader will use a 3D perspective camera, that can move around the world. A ray can:

1. Miss the world entirely, this should result in a sky color being output at that pixel.
2. Hit the world, but not hit any solid voxels. A solid voxel is determined as a distance of 0. This will also result in a sky color being output at that pixel.
3. Hit the world, and hit a solid voxel. This will result in the color at that voxel being output to the pixel.



(a) A ray is initially shot from the camera, hitting an outermost voxel of the world; a distance of "3" is read from the distance field.

(b) The ray advances 3 voxels using DDA (Amanatides et al. 1987), before reading the distance field again. A distance of "1" is found, so the ray should continue marching through the world.

(c) The ray advances the final distance, as per the previous step, finally hitting a voxel with a distance of "0". This marks the end of the ray marching, and a colour can be read from the distance field for it to be rendered.

Figure 3.4: Example of a ray marching through a discrete distance field using DDA (Amanatides et al. 1987).

3.4 Demonstration Application

To demonstrate the using dynamic distance fields from voxel grids in action, a demonstration application will be created; this application will also be used for performance testing as defined in Section 3.5. The application will contain a voxel finite world that can be interacted with, voxels can be placed and deleted by the user.

The size of the world, and parameters of the distance field generation algorithm, can be altered to allow for the quick execution of different performance tests. SVO implementations typically see upwards of 10 voxel levels (Laine & Karras 2010) resulting in a world size of at least 1024^3, all the way to worlds of sizes 65536^3. The simulation will be run at different world sizes but will be limited by memory consumption due to the uncompressed nature of the world representation and distance field.

3.4.1 Vulkan Architecture

To support the graphical demonstration application and the compute pipeline for my algorithm, I selected the Vulkan API. Vulkan provides low-overhead, high-performance access to GPU re-

sources, offering fine-grained control over memory management, synchronization, and pipeline configuration. This level of control was essential for achieving the performance characteristics and flexibility required by the project. However, Vulkan's power comes with significant complexity: it demands meticulous management of resources, explicit synchronization, and a steep learning curve compared to higher-level graphics APIs such as OpenGL or DirectX 11.

Developing with Vulkan will require a substantial upfront investment in infrastructure code, but will ultimately enable a highly efficient and predictable execution environment tailored to the specific needs of the application. To accelerate development, the initial setup of compute pipelines and rendering was based on the *interactive fractal* example provided by the Vulkano project (Vulkano 2024).

3.5 Performance Evaluation

As part of the evaluation of the distance field generation in a dynamic environment, several performance metrics need to be gathered.

The performance test will be run multiple times with the same parameters to ensure accurate metrics are gathered; for each implementation of the distance field computation, the demonstration application will be run using differing world sizes, this may vary from implementation to implementation depending on their specific limitations. Results between implementations will also be evaluated.

To ensure a consistent test environment, the demonstration application will be modified to automatically modify the world at set intervals; this will mean that results will be reproducible when using the same seed. Voxels will be placed and deleted from the world at random positions, with random brush sizes.

3.5.1 Frame Performance Metrics

Frame performance metrics will help in determining whether the application can run in real-time in a "playable" speed. The minimum required frame rate for a game to be deemed playable is a heavily debated topic; however, player performance typically hits a plateau above 60 frame-per-second (FPS) (Claypool & Claypool 2007), with 30 FPS being a solid starting point. For this paper, a minimum target of 30 FPS will be set; this minimum FPS is what should be seen while

the world is being actively updated, it is expected that when no updates are being carried out the ray marching of the world should stay above 30 FPS consistently.

The delta time of each frame will be recorded while the application is running, and the frame rate will be calculated using the equation 3.1.

$$\text{FPS} = \frac{1000}{\text{dt}_{\text{ms}}} \tag{3.1}$$

Both the overall frame performance of the application, and during distance field computation, will be gathered.

3.5.2 Distance Field Computation Metrics

The primary metric used here will be the execution time. Assuming a 30 FPS target, the frame time is 33.33ms, this means for any given frame all application operations must take less than 33.33ms to achieve the target FPS;the upper limit for the distance field execution time is 33.33ms which would assume all other factors like ray marching and presenting a frame take no time in a frame.

3.6 Limitations and Considerations

The performance of the distance field computation, and the rendering, is highly hardware dependent; factors such as:

1. GPU specifications
2. Memory configuration
3. Driver versions

With this in mind the performance testing for this paper is carried out on a laptop with the following hardware specifications, tests will be carried out while the laptop is plugged in and all battery saving and energy efficient modes will be disabled:

CPU: AMD Ryzen 9 5900HS with Radeon Graphics @ 3.3GHz

GPU: NVIDIA GeForce RTX 3070 Laptop GPU

RAM: 16384MB @ 3200 MT/s

Chapter 4

Implementation

This chapter will focus on the implementation of various methods for generating a discrete distance field. It will start with a basic brute-force approach before delving into optimizations and other algorithms. Each implementation will include a short performance test for comparison between different implementations.

4.1 Brute-force Approach

A brute-force implementation for calculating the discrete distance field given a voxel grid is the most straight-forward to implement but will suffer from performance; especially as world sizes get larger.

To compute the distance field for a voxel grid using a brute-force approach, we consider a voxel, V at (x, y, z). The algorithm starts iterating from the origin of the world $(0, 0, 0)$ and progresses incrementally along each axis of the grid. For every voxel in the grid, the Manhattan distance is calculated to V. This exhaustive method, while producing an accurate distance field, must explore every other possible coordinate within the grid; this can be seen in Algorithm 1.

In the worst-case scenario, the algorithm must evaluate the distance for all N^3, where N is the size of one axis and the grid has uniform dimensions. Thus, the worst-case complexity is $O(N^3)$. In the best-case, when the voxel V is located near the origin a complexity of $O(1)$ can be achieved, but this is highly unlikely.

Algorithm 1 Brute Force Distance Field Calculation

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z)

Output: Distance field grid D

Initialize $D[i][j][k] \leftarrow N$ for all $i, j, k \in [0, N-1]$
for $i = 0$ to $N-1$ **do**
  for $j = 0$ to $N-1$ **do**
    for $k = 0$ to $N-1$ **do**
      if $V[i][j][k]$ is solid **then**
        $d \leftarrow |i-x| + |j-y| + |x-z|$ {Manhattan distance calculation, this will be common to all implementations.}
        if $d < D[i][j][k]$ **then**
          $D[i][j][k] \leftarrow d$ {Write only the shortest distance to the output.}
        end if
      end if
    end for
  end for
end for
Return: D

4.1.1 Performance Results

At very small world sizes, the performance of the brute-force algorithm is sufficient; however the performance gets exponentially worse the larger the world becomes, this can be seen in the FPS of the application in Table 4.1. At a world size above 256^3, the amount of work required by each warp on the GPU becomes too large resulting in the application crashing, as such the testing for this only went to a world size of 128^3.

World Size	8^3	16^3	32^3	64^3	128^3
Avg. FPS	142.33702	142.335887	91.44403	2.30128	0.04184

Table 4.1: Frame rate of the brute-force algorithm at varying world sizes with a modification every 200ms.

World Size	8^3	16^3	32^3	64^3	128^3
Avg. Time (ms)	0.13085426	0.72909933	8.415086	431.46756	25854.305
Std. Deviation (ms)	0.07906039	0.72129595	0.6034345	23.69193	43.82031
Confidence Interval (ms)	(0.12865908, 0.13304944)	(0.7251273, 0.7330714)	(8.400318, 8.429853)	(428.5129, R34.4222)	(25830.484, 25878.125)

Table 4.2: Distance field compute shader execution time using the brute-force algorithm.

The results in Table 4.2 highlight how a brute-force approach is unsuitable for large dynamic worlds. A common optimization is chunking to split a large world into smaller "chunks" as described in Section 4.2.

4.2 Splitting a World into Chunks

Partitioning, or chunking, is a common approach to divide a large problem into smaller manageable problems. In the context of voxels world, sparse voxel octrees (SVO) are an approach for dividing a large dense representation of a voxel grid into a more sparse format with data only stored where it's needed; this makes it more efficient to process and render (Laine & Karras 2010, Mileff & Dudra 2019, van Wingerden 2015).

Given that a brute-force approach has acceptable performance at a world size of 16^3, as can be seen in Table 4.2, we can divide the whole world into smaller 16^3 chunks. This will allow for updates in one chunk to be localized, as such updates will not be required to update the whole world reducing the amount of iterations required to update a distance field.

For a 512^3 sized world, we could divide it into $32,768$ chunks each with a size of 16^3. A worst-case complexity for this significantly larger world is now $O(16^3)$ compared to the $O(512^3)$ it would otherwise be without a chunking approach.

Chunks, however, present a significant problem in distance field generation as they can introduce inaccuracies between chunk borders. This can happen if we don't consider the voxels in an adjacent chunk when calculating distances, there are three potential solutions to this problem:

1. When iterating over a chunk, iterate over a size $X+2, Y+2, Z+2$ to introduce "padding". Checking neighbours that are in padding region will be treated as solid which will introduce a border in the distance field that would force rays to march into the beginning of the next chunk.

2. Include the adjacent chunks as input to the distance field compute shader. Out-of-bounds

accesses should result in checking neigbouring chunks; however, this expands the number of voxels needed to be checked and will still suffer from inaccuracies if the nearest solid voxel is not in a neighbouring chunk.

3. Combining the first approach, with multiple passes. An initial distance field calculation is computed for each chunk independently. To ensure accurate distances at chunk boundaries, another pass through the world can be done that includes neighbour information. Multiple, more global, passes will ensure distances eventually converge on the correct distance value (Gorobets 2023, Sinharoy & Szymanski 1993, Xu, Wang, Liu, Liu & He 2015).

4.2.1 Padding

The chosen approach at this point, is to introduce padding to an individual chunk when calculating the distance field. With this approach the worst-case complexity is slightly worse than without using chunking. Without chunks a 16^3 world, has a complexity $O(N^3)$, with chunks we require padding and so a chunk of the same size would have a complexity of $O((N+2)^3)$.

To account for the "padding" around a chunk, out-of-bounds accesses will be treated as a solid voxel.

Algorithm 2 Get Voxel at (x, y, z)

Input: Voxel grid V, position x, y, z

1: **if** x, y, z is within bounds of V **then**
2: **return** $V[x][y][z]$
3: **else**
4: **return** Solid voxel
5: **end if**

The algorithm for the distance field calculation remains largely unchanged except for now using Algorithm 2 to access the voxel grid V instead of direct access. The updated algorithm is now implemented as follows.

Algorithm 3 Brute force Distance Field Calculation (With chunks)

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z)

Output: Distance field grid D

Initialize $D[i][j][k] \leftarrow N$ for all $i, j, k \in [0, N-1]$
for $i = -1$ to N **do**
  for $j = -1$ to N **do**
    for $k = -1$ to N **do**
      `voxel` $\leftarrow$ `Get Voxel at` (i, j, k)
      if `voxel is solid` **then**
        $d \leftarrow |i - x| + |j - y| + |x - z|$
        if $d < D[i][j][k]$ **then**
          $D[i][j][k] \leftarrow d$
        end if
      end if
    end for
  end for
end for
Return: D

4.2.2 Performance Results

Based on the performance results of the brute-force approach, as can be seen in Table 4.2, the proceeding tests will use a chunk size of 16^3. The key improvement in using chunks is that the total world size is theoretically only limited by the memory consumption, and number of storage buffers supported by the GPU. Sufficiently large updates spanning multiple chunks will be less performant, but we can expect that a small localized update affecting only a couple of chunks will have only marginally worse performance than the previous approach.

We can see from the performance results in Table 4.3 that we are able to maintain a low average execution time when updating the distance field; however, FPS gets significantly worse at larger world sizes, this is due to the cost of managing a high amount of storage buffer objects and memory read/write operations.

World Size	32^3	64^3	128^3	256^3	512^3
Avg. FPS	141.02969	140.32344	88.75602	22.13375	3.37203
Avg. Execution Time	0.8090571	0.7992149	0.7797105	0.7632012	0.7504562
% Improvement	54.2251%	5997.63%	212032%	Inf%	Inf%

Table 4.3: Frame rate, and execution time, of the brute-force algorithm, when using a chunk size of 16^3, at varying world sizes with a modification to the world every 200ms. Percentage improvement is the improvement in frame rate compared to a comparably sized world without chunks, as demonstrated in Table 4.1.

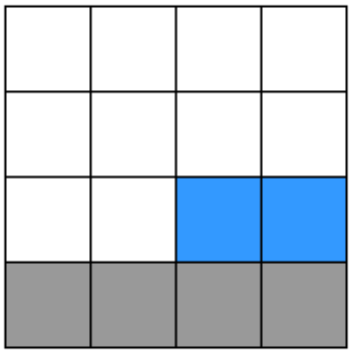
4.3 Fast Iterative Method

The fast marching method (FMM) (Sethian 1999) and fast iterative method (FIM) **??** are approaches used for solving a boundary value problem; in the case of a voxel grid distance field our boundary problem is defined as identifying the distances for each air voxel to the nearest voxel. FMM is a sequential algorithm that relies on processing scans of the grid in the correct order and complex data structures making it unsuitable for the GPU; FIM is an enhancement that doesn't have those same requirements but can achieve the same result.

FIM computes distance fields by propagating distance information outward from known boundary regions; the boundary region for a voxel grid is computed by setting all solid voxels to a distance of 0 and all air voxels to a sufficiently large number, this can be seen in Figure 4.1 and Algorithm 4.

On a GPU, each voxel can be processed in parallel by leveraging atomic minimums. When distance information is propagated in a single iteration only the minimum value is saved to the distance field. A distance can at most propagate 1 voxel away from the voxel being processed, this means that to achieve an accurate distance field FIM must be executed on the distance field several times until shortest distances have been calculated for all voxels, the effect of several iterations can be seen in Figure 4.2.

The number of iterations required for convergence is proportional to the size of a chunk - a voxel that is N units away from the chunk boundary, will require at least N units to receive the correct distance value. As such, given a chunk of size $C = 8$, the total number of voxels to process is $N = 8^3$, the worst-case computation will be $O(8N)$.



(a) A 2D representation of the world. Empty voxels are indicated by white cells.

8	8	8	8
8	8	8	8
8	8	0	0
0	0	0	0

(b) A 2D representation of a voxel grid initialized in preparation for FIM.

Figure 4.1: The corresponding initialized distance field for a voxel grid.

8	8	8	8
8	8	8	8
8	8	0	0
0	0	0	0

(a) Iteration 0.

8	8	8	8
8	8	1	1
1	1	0	0
0	0	0	0

(b) Iteration 1.

8	8	2	2
2	2	1	1
1	1	0	0
0	0	0	0

(c) Iteration 2.

3	3	2	2
2	2	1	1
1	1	0	0
0	0	0	0

(d) Iteration 3.

Figure 4.2: Convergence of distance field values after several iterations when using FIM.

Convergence can be handled on the CPU or on the GPU, the chosen approach was to implement a single iteration of the FIM algorithm, as can be seen in Algorithm 5, as a compute shader. The CPU will dispatch the compute shader until no changes have been made to the distance field, as such it can be concluded that distances have converged.

Algorithm 4 Fast Iterative Method, Distance Field Initialization

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z)

Output: Distance field grid D

1: `voxel` $\leftarrow$ `Get Voxel at` (x, y, z)

2: **if** `voxel is solid` **then**

3: $\quad D[x][y][z] \leftarrow 0$

4: **else**

5: $\quad D[x][y][z] \leftarrow N * 2$

6: **end if**

Algorithm 5 Fast Iterative Method

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z)

Output: Distance field grid D

```
voxel ← Get Voxel at (x, y, z)
if voxel is solid then
    Return: D
end if
Initialize d_min ← N * 2
for neighbours n of voxel do
    neighbour ← Get Voxel at n
    if neighbour is solid then
        d_n ← 0
    else
        d_n ← distance value at n
    end if
    d_n ← d_n + 1
    d_min ← min(d_min, d_n)
end for
if d_min < current distance of voxel then
    Update D at (x, y, z) to d_min
    Mark distance field as changed
end if
Return: D
```

4.3.1 Performance Results

FIM is expected to be significantly more efficient than the brute-force approach, as such the optimal chunk size to use needs to be determined first. We can see from Table 4.4, that we are able to compute the distance field for a chunk much faster; however at 128^3 sized chunks the variance in the number of iterations required for convergence leads to inconsistent performance.

There is a substantial increase in performance for distance field computation of single chunks; however, the cost of CPU-GPU synchronization means that once there are several chunks that

Chunk Size	8^3	16^3	32^3	64^3	128^3
Total Avg. Time (ms)	0.0408411	0.05089584	0.1566725	1.089782	17.556416
Std. Deviation (ms)	0.012230597	0.009375835	0.07214462	0.5198319	16.088636
Average Iterations for Convergence	1.0313779	1.1514729	1.4801816	2.3467271	6.194286

Table 4.4: Distance field compute shader execution time (as a total of all iterations required to achieve convergence) using the FIM algorithm.

need updating there is a significant amount of time spent on waiting for various fences to be signalled for synchronization. The time spent waiting for fences is time not spent doing useful work on the GPU nor CPU, as such we can observe a decline in performance at large world sizes with several chunks as can be seen in Figure 4.3.

4.4 Jump Flooding Algorithm

The Jump Flooding Algorithm (JFA) works by propagating information about the closest occupied voxels through a grid using a pattern of "jumps" with progressively decreasing step sizes. Unlike traditional flooding algorithms that work with neighboring cells, JFA allows information to "jump" across larger distances initially, making it highly efficient.

For a distance field from a voxel occupancy grid, this can be seen in Figure 4.4 and Algorithm 6:

1. Initialize the distance field with an initial known boundary, similar to the initialized required for FIM as seen in Figure 4.1.
2. Execute multiple passes with decreasing step sizes using powers of 2.
3. In each pass, the shortest distances are gathered from neighbouring voxels using the step size.

JFA operates in $log(N)$ passes for an NxN grid, with the step size halving in each pass - starting with a step size $N/2$ decreasing to a step size of 1 using powers of 2. The expected complexity of JFA is $O(N^3 log(N))$. JFA is a highly efficient algorithm; however, due to the nature of "jumps" and ambiguous closest point information it is possible for the true shortest distances to not be chosen when checking neighbouring voxels, this can result in a distance field that is not exact.

Jump Flooding requires multiple passes for distance information to be propagated through the

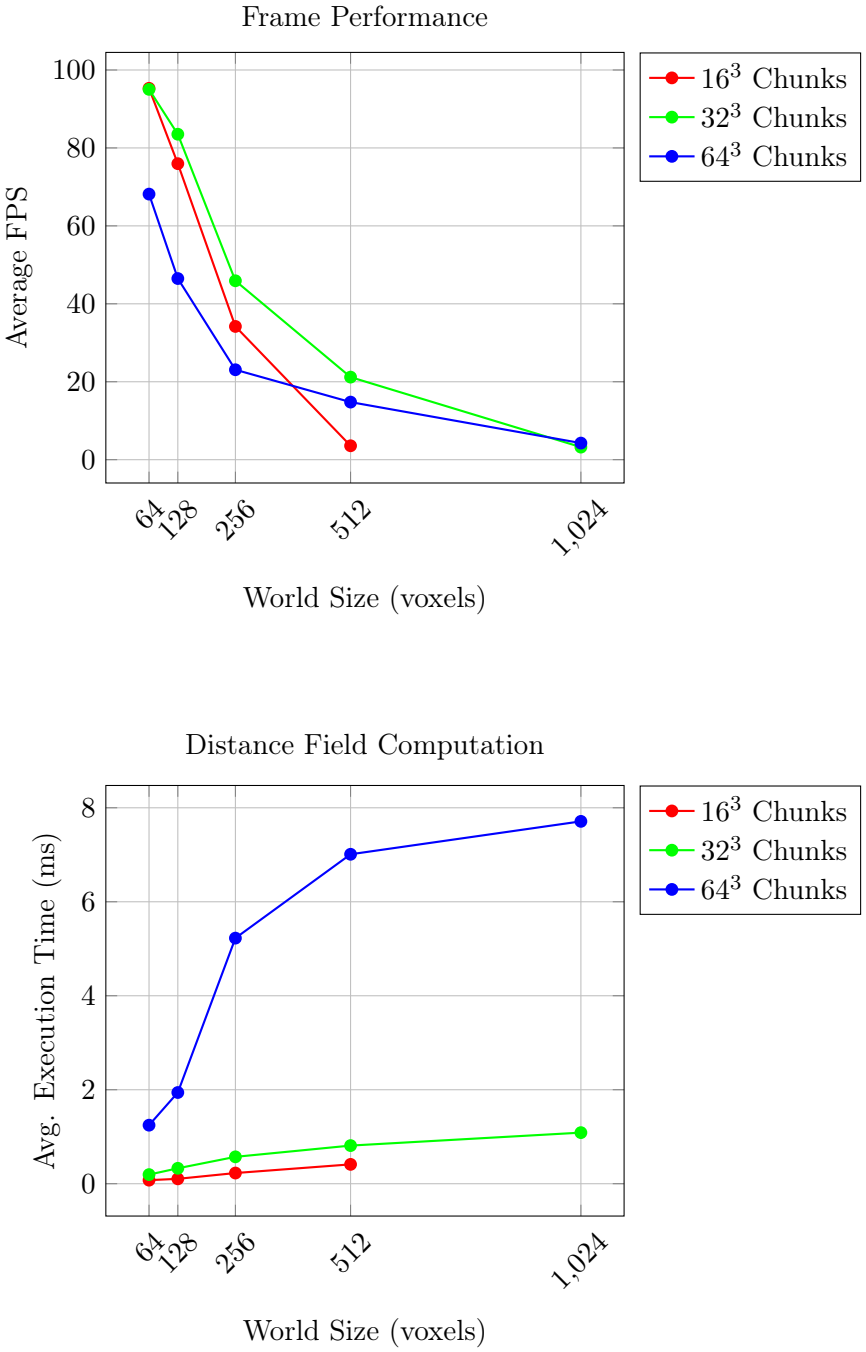
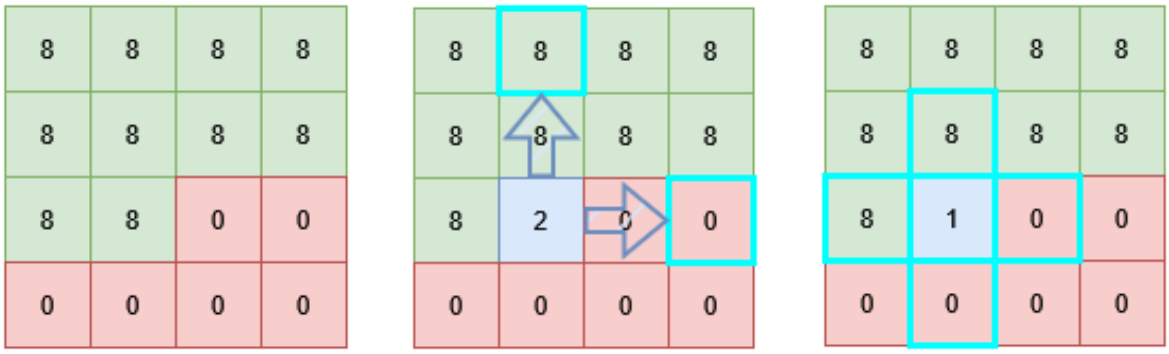


Figure 4.3: Comparison of the performance of the Fast Iterative Method at different world and chunk sizes.



(a) Initialized distance field. (b) Pass 1, Step Size 2. (c) Pass 2, Step Size 1

Figure 4.4: Illustration of checking neighbours at decreasing step sizes.

entire distance field - the chosen approach is for the compute shader to implement a single iteration of JFA, as seen in Algorithm 6. The CPU will be responsible for dispatching the compute shader at decreasing step sizes until a step size of 1 is reached. This will again introduce CPU-GPU synchronization losses but makes implementing the algorithm simpler. Similar to FIM, it is required that the distance field is initialized with large values for air voxels, and distances of 0 for solid voxels as seen in Algorithm 4.

Since information needs to be written to, and read from, the distance field at the same time it is necessary to employ a "ping-pong" buffer approach where buffer A is only used to read distance information, while new distance information is written to buffer B - these buffers are then swapped before the next iteration of JFA. This will eliminate any race conditions that may arise.

Algorithm 6 Jump Flooding Algorithm

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z), Step Size S

Output: Distance field grid D

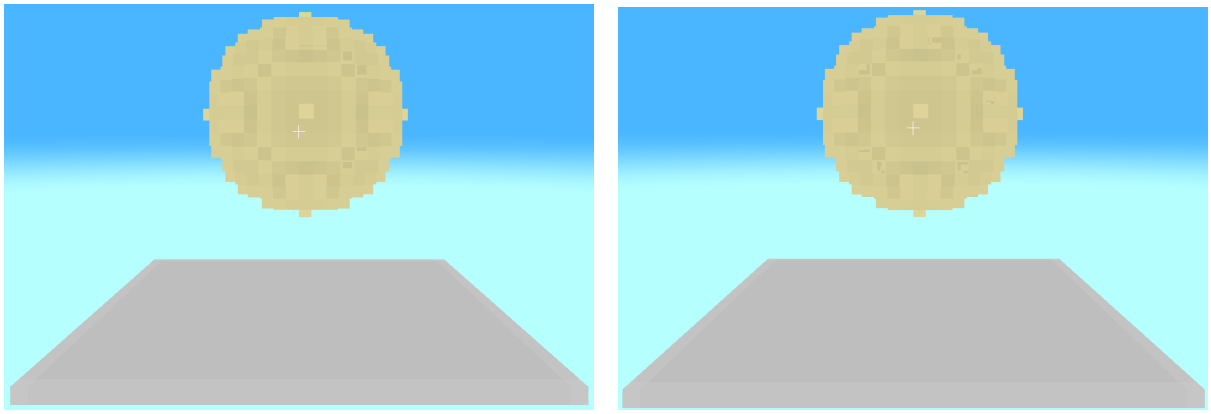
1: `voxel` ← `Get Voxel at` (x, y, z)
2: **if** `voxel is solid` **then**
3: **Return:** D
4: **end if**
5: Initialize d_{min} ← distance value of voxel at (x, y, z)
6: **for** neighbours n of voxel S steps away **do**
7: `neighbour` ← `Get Voxel at` n
8: d_n ← distance value of voxel at *neighbour*
9: $d_{min} \leftarrow \min(d_{min}, d_n + S)$
10: **end for**
11: **Return:** D

4.4.1 Performance Results

We can see from Table 4.5 and Figure 4.6 that the compute shader execution time, as a total of all dispatches required to complete JFA for a single chunk, is significantly quicker that FIM; it is feasible to now use chunks as big as 64^3. This improved performance is not entirely comparable to FIM, as JFA only produces an approximate distance field, this means that the render of the distance field will contain minor artifacts where distances don't align with the true surface which can be seen in Figure 4.5.

Chunk Size	8^3	16^3	32^3	64^3	128^3
Total Avg. Time (ms)	0.031079482	0.37563447	0.07390518	0.38933817	8.052377
Std. Deviation (ms)	0.0060166107	0.0011396826	0.014052732	0.1003621	3.9264889

Table 4.5: Distance field compute shader execution time using the JFA algorithm.



(a) Fast Iterative Method

(b) Jump Flooding Algorithm

Figure 4.5: Comparison of rendering an exact and approximate distance field.

4.5 Coarse JFA with FIM Refinement

We observed that JFA can produce an approximate distance field, with minimal artifacts, very quickly in Section 4.4. FIM is efficient at producing an accurate distance field but the distribution in the number of iterations required for convergence can have a negative impact on the performance; as described in Section 4.3, FIM works by propagating minimum distances through the distance field.

It is possible to use a combination of JFA and FIM to increase the execution speed, compared to a pure FIM implementation, and maintain the computation on an exact distance field; the implementation for this uses the following 3 steps, with each step being defined in its own compute shader and being dispatched by the CPU in series:

1. **Algorithm 4**: to initialize the distance field.
2. **Algorithm 6**: to create a more accurate "seed" that FIM can be run against to reduce the number of iterations required for FIM to converge on the correct distance values.
3. **Algorithm 5**: to converge distance values to their minimums.

4.5.1 Performance Results

Executing the distance field compute shader on single chunks of varying sizes, as seen in Table 4.6, we observe that the number of required iterations to achieve convergence is decreased compared

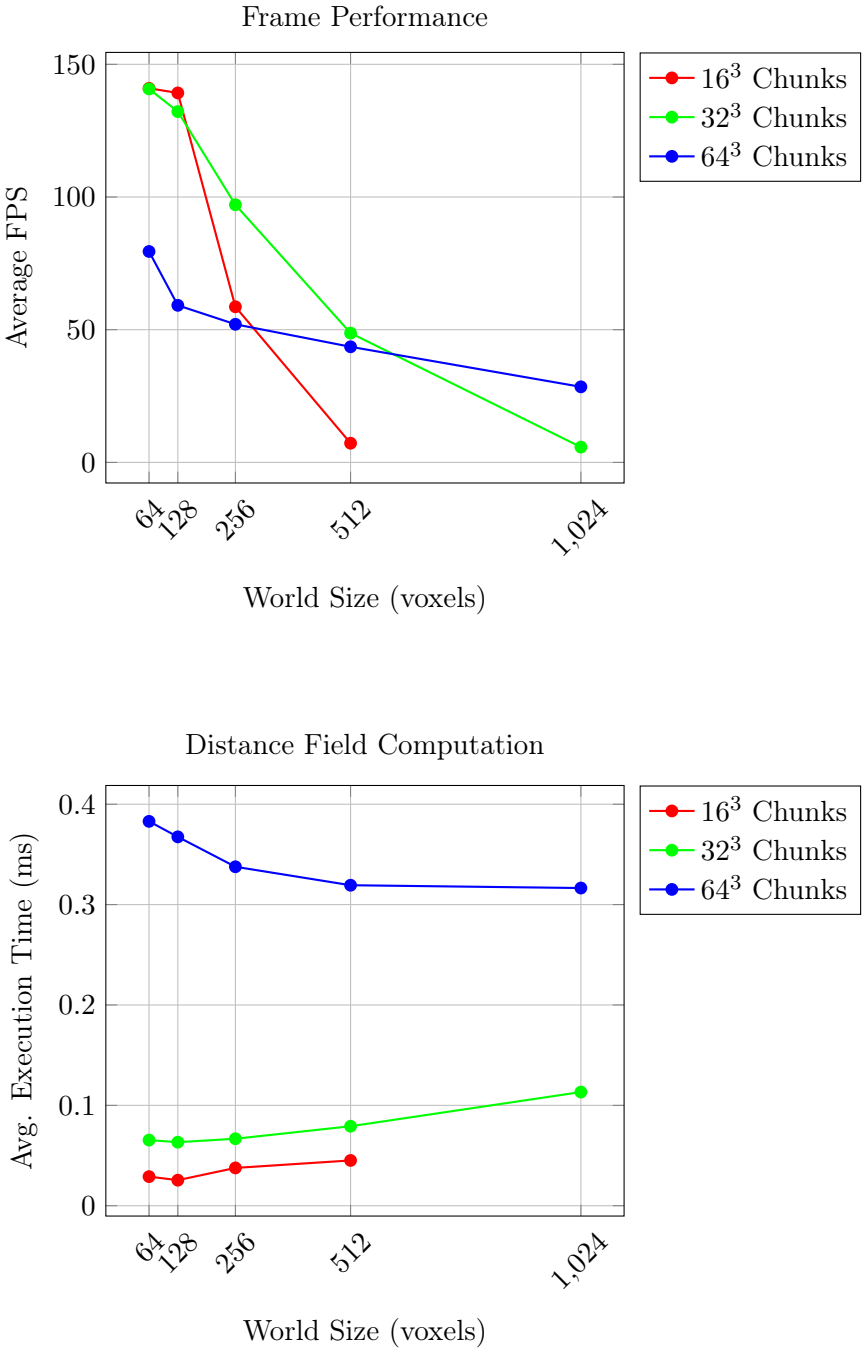


Figure 4.6: Comparison of the performance of the Jump Flooding Algorithm at different world and chunk sizes.

to running FIM on a freshly initialized distance field as seen in Section 4.3. The total execution time is also less compared to pure FIM, especially as chunk sizes get larger.

Chunk Size	8^3	16^3	32^3	64^3	128^3
Total Avg. Time (ms)	0.04326828	0.042948034	0.142226286	0.8856182	16.556087
Std. Deviation (ms)	0.014001529	0.0082087135	0.004889435	0.48032447	13.672007
Average Iterations for Convergence	0.21335079	0.41733277	0.9516885	1.2535588	4.1610336

Table 4.6: Distance field compute shader execution time using hybrid JFA and FIM approach. Compared against a pure FIM execution.

4.6 Level-of-detail with JFA and FIM

Using the previous hybrid JFA and FIM approach in Section 4.5 optimized the execution of FIM; however, we lost the advantage of excellent execution time that pure JFA results in as was visible in Figure 4.6.

We can reduce the amount of expensive work the GPU needs to do calculating exact distance fields only for the chunks where it is necessary to have precise surface information. Chunks that are determined to be of less significance can suffice with only JFA being run against them, this way they have a distance field representation that closely matches the true surface and when rendered produce a passable result at a distance.

The approach in this section is similar to level-of-detail traditionally used in video games with mesh rendering, where far away meshes are composed of fewer triangles to reduce the performance strain of that mesh as a large amount of detail is not necessary for objects that will only take up a small amount of space in the displayed image.

Level-of-detail can be implemented for our approach by executing JFA for chunks that fall outside a "focal point", and only executing FIM on chunks that are within the focal point. We can vary the amount of detail in the world overall by increasing the focus size. The execution of FIM would be carried out on chunks that have already had JFA run on them. An overall implementation of this can be seed in Algorithm 7, with the decision of what algorithm to run being decided by the CPU, and the individual algorithms being implemented as their own compute shaders executed by the GPU.

Algorithm 7 JFA and FIM with focus area

Input: Chunk voxel grid C_{voxels}, Chunk position C_{pos}, Focus area F, Focus size F_{size}

Output: Distance field grid D

1: Initialize $F_{min} \leftarrow F - F_{size}$

2: Initialize $F_{max} \leftarrow F + F_{size}$

3: Execute `Algorithm 4` on C_{voxels}

4: Execute `Algorithm 6` on C_{voxels}

5: **if** $C_{pos} >= F_{min}$ and $C_{pos} <= F_{max}$ **then**

6: Execute `Algorithm 5` on C_{voxels}

7: **end if**

4.6.1 Performance Results

The focal point for all the following test was located at the center of the world; the effect of changing the focus size was the focus. We observe from Figure 4.7 and Figure 4.8 that increasing the focus size has a significant impact on the performance; however, as can be seen from Figure 4.9 there is a significant increase in visual fidelity as the focus size increases and more chunks have an exact distance field computed using FIM.

4.7 Optimized implementation

This section will focus on applying various optimization to the Algorithm described in Section 4.6 to achieve the best possible performance.

4.7.1 JFA GPU Synchronization

Section 4.4 covers the Jump Flooding Algorithm in detail, but the chosen implementation in that section was for the compute shader to execute a single pass of the algorithm. This approach means that the CPU is required to wait on the GPU to flag that the required buffers are no longer in use before the next pass, at a smaller step size, can be dispatched by the CPU.

The CPU uses fences to wait for the GPU to signal when required resources are no longer in use by the GPU, this requires synchronization across the CPU and GPU and, in our case with

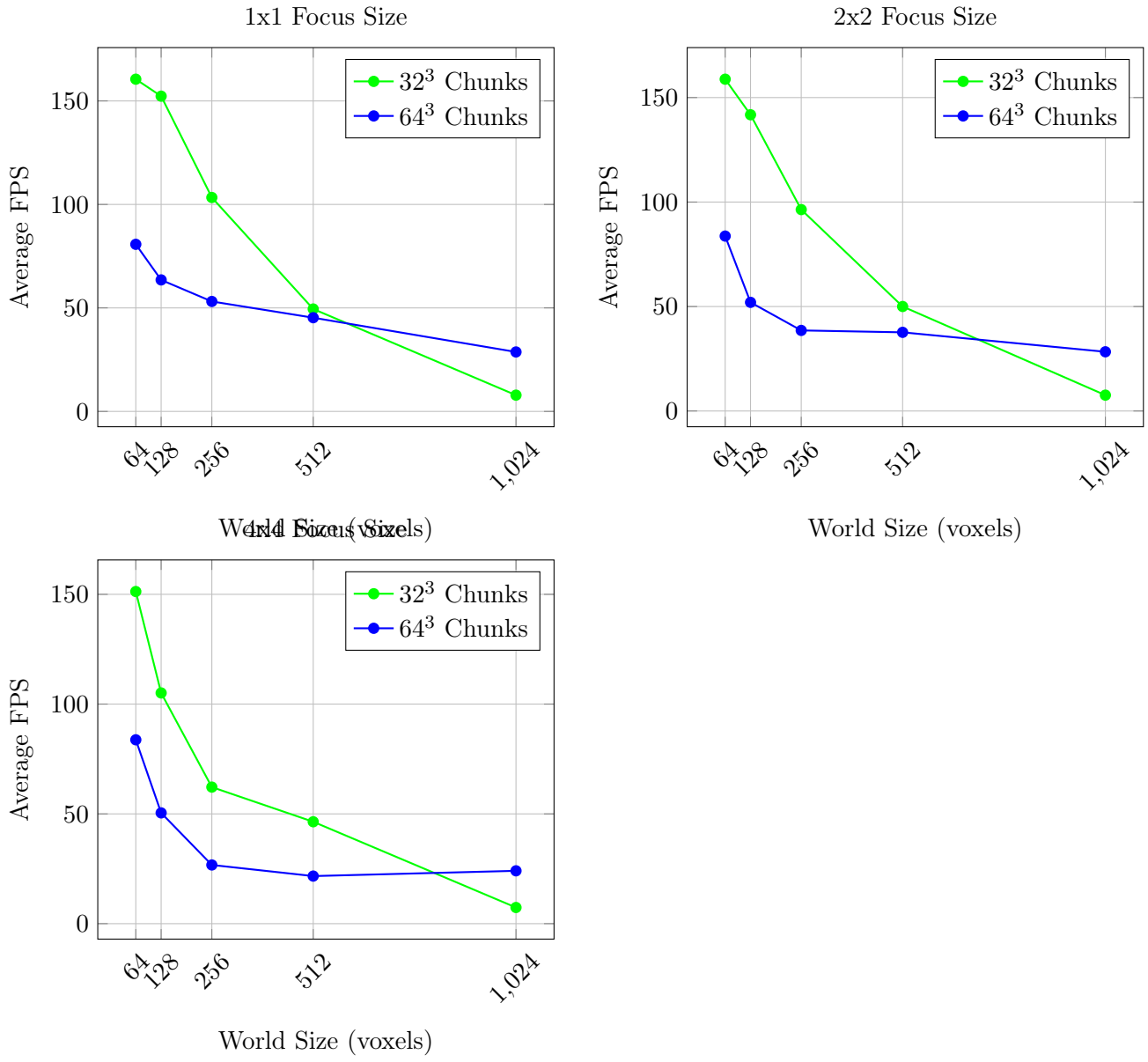


Figure 4.7: The effect of focus size on the FPS using a level-of-detail approach with JFA and FIM.

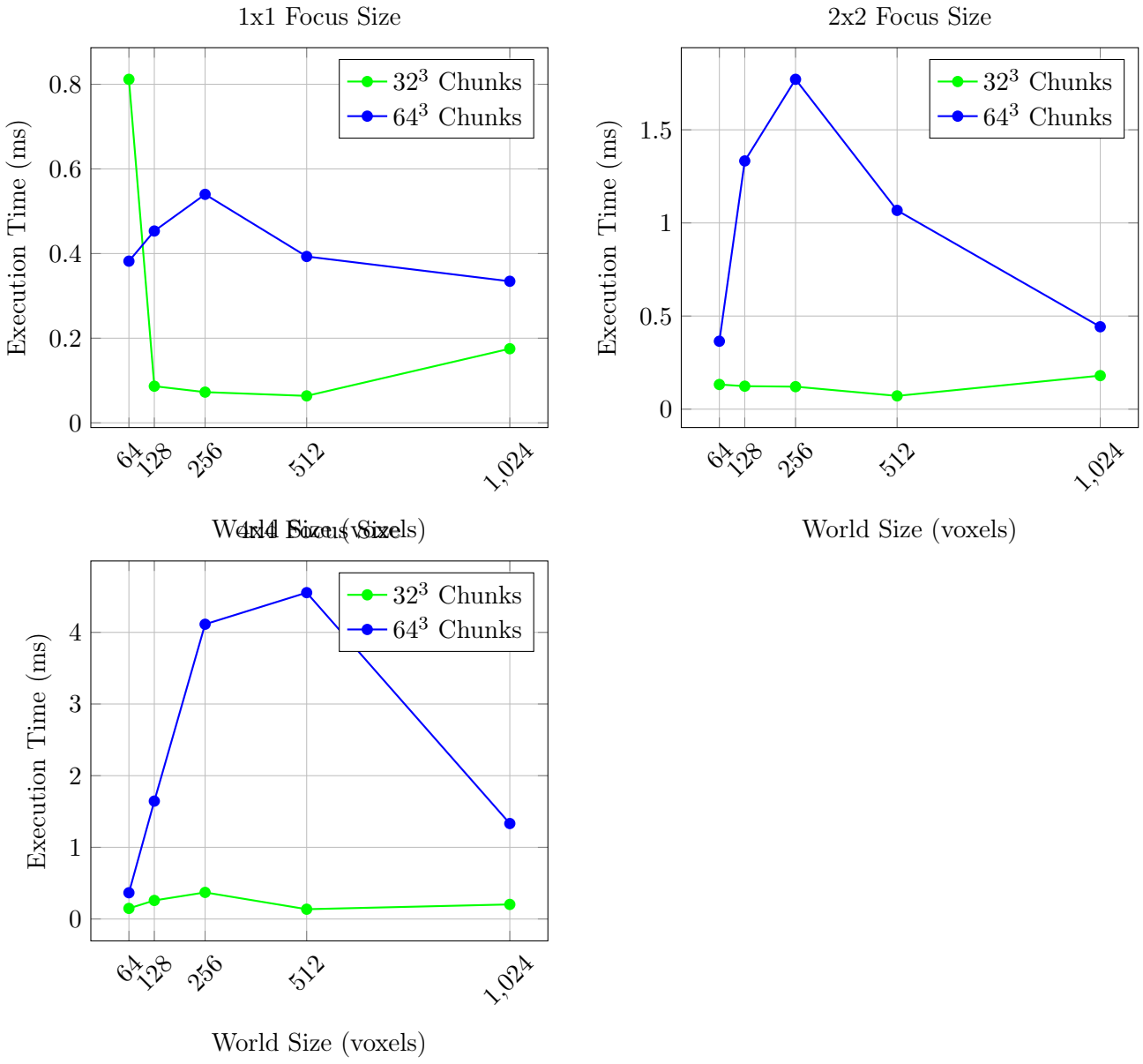
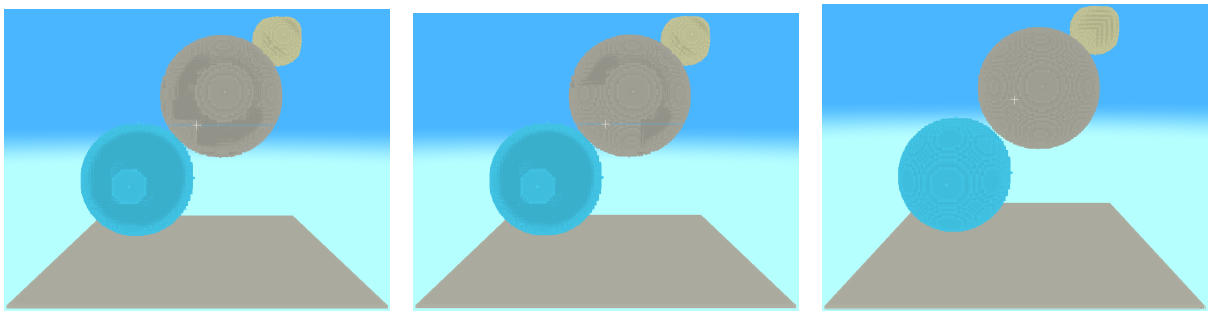


Figure 4.8: The effect of focus size on the computation time of the distance field using a level-of-detail approach with JFA and FIM.



(a) 1x1 focus size at the center of the world.

(b) A 2x2 focus size at the center of the world.

(c) Focus size encompassing the whole world.

Figure 4.9: Comparison of the rendering artifacts and distance values inaccuracies introduced by utilizing JFA in regions outside the focus point.

large buffers, is an expensive process, especially as the number of buffers and chunks increases.

An updated algorithm for JFA with the logic for iterating through decreasing step sizes can be seen in Algorithm 8; this is implemented in a single compute shader, as such the CPU now only needs to dispatch the compute shader once with an initial step size, and the required number of iterations.

Algorithm 8 Jump Flooding Algorithm with Iterations

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z), Initial Step Size S_{init}, Required Iterations I_{req}

Output: Distance field grid D

```
voxel ← Get Voxel at (x, y, z)
if voxel is solid then
    Return: D
end if
Initialize i ← 0
for i < I_req do
    Initialize S_cur ← S_init >> i
    if S_cur == 0 then
        Return: D
    end if
    Initialize d_min ← distance value of voxel at (x, y, z)
    for neighbours n of voxel S steps away do
        neighbour ← Get Voxel at n
        d_n ← distance value of voxel at neighbour
        d_min ← min(d_min, d_n + S)
    end for
    Memory barrier to synchronize all warps and threads
    i ← i + 1
end for
```

4.7.2 FIM GPU Synchronization

As described in Section 4.3, the FIM compute shader was responsible for running a single iteration of the FIM algorithm; a flag is set at the end of the compute shader to indicate whether any changes were made. The CPU is responsible for dispatching the FIM compute shader until no changes are made to the distance field. A host-visible buffer was used which contains a flag indicating whether any changes were made to the distance field, this way the GPU is able to write to it, and the CPU can read and reset it as needed; this requires further

fences and barriers to ensure CPU and GPU don't try to write to the buffer at the same time.

This can be optimized by having a device-only buffer that the GPU is entirely in control of; this buffer will contain a modified count, and the GPU can then execute another iteration of FIM if no changes were made. An optimized FIM algorithm used in the compute shader can be seen in Algorithm 9.

Algorithm 9 Fast Iterative Method with Iterations

Input: Voxel grid size N, Voxel grid V, Voxel location (x, y, z)

Output: Distance field grid D

```
voxel ← Get Voxel at (x, y, z)
if voxel is solid then
    Return: D
end if
Initialize i ← 0
Initialize i_max ← max(dimensionfromN)
Initialize changes ← 0
for i < i_max do
    Initialize d_min ← N * 2
    for neighbours n of voxel do
        neighbour ← Get Voxel at n
        if neighbour is solid then
            d_n ← 0
        else
            d_n ← distance value at n
        end if
        d_n ← d_n + 1
        d_min ← min(d_min, d_n)
    end for
    if d_min < current distance of voxel then
        Update D at (x, y, z) to d_min
        changes ← changes + 1
    end if
    Memory barrier to synchronize all warps and threads
    if changes == 0 then
        Return: D
    end if
    i ← i + 1
end for
```

4.7.3 Single monolithic compute shader

In the current implementation of level-of-detail as described in Section 4.6, each algorithm is implemented as its own compute shader that is then dispatched by the CPU as needed. This requires further CPU-GPU synchronization and the re-binding of various (potentially) large buffers; we can instead create a single monolithic shader that has all the resources needed for each algorithm bound once, and is then responsible for initializing the distance field, running JFA, and deciding whether FIM should also be run depending on the focus area. The complete algorithm for this can be seen in Algorithm 10, where there are algorithms referenced they are included within the implementation but have been excluded for brevity.

Algorithm 10 Monolithic Compute Shader

Input: Chunk voxel grid C_{voxels}, Chunk position C_{pos}, Focus area min F_{min}, Focus area max F_{max}, Voxel location (x, y, z)

Output: Distance field grid D

1: `voxel` $\leftarrow$ `Get Voxel at` (x, y, z)
2: **if** `voxel is solid` **then**
3: **Return:** D
4: **end if**
5: Execute `Algorithm 4` on C_{voxels}
6: Execute `Algorithm 8` on C_{voxels}
7: **if** $C_{pos} >= F_{min}$ and $C_{pos} <= F_{max}$ **then**
8: Execute `Algorithm 9` on C_{voxels}
9: **end if**

4.7.3.1 Performance Results

Applying all the optimizations as described in Sections sections 4.7.1 to 4.7.3, there is a noticeable improvement in both FPS and compute shader execution time when using a 4x4 focus size. Compute shader execution time tends to decrease with larger worlds as more chunks will use JFA which is a faster algorithm and so the average execution time will be less compared to worlds where the focus size means all chunks get recalculated using FIM; this can be seed in Figure 4.10 and Figure 4.11.

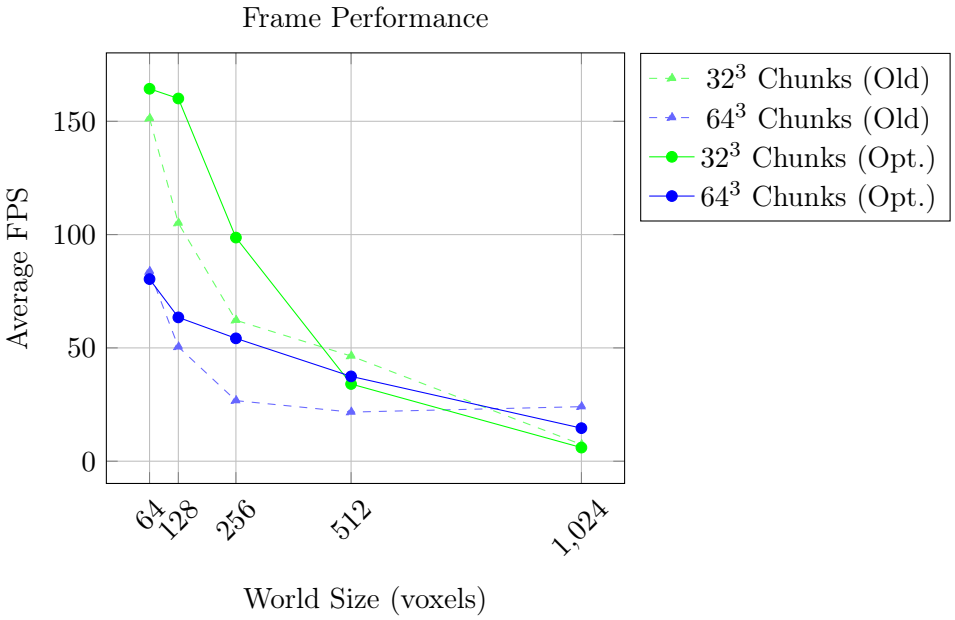


Figure 4.10: Comparison of the average FPS of the demo application using the optimized monolithic shader.

FPS decreases significantly at larger world sizes; however, this is due to the large overhead in binding a large amount of storage buffers required by the ray marcher to be able to check all voxels along a ray. Without focussing on areas such as memory consumption, and alternative representations, or different approaches for rendering distance fields the FPS will be difficult to improve at large world sizes.

4.7.4 Pure Distance Field

As described in Section 3.2, the current distance field encodes colour information as well; this approach was chosen to reduce the amount of storage buffer objects needed to be bound for the ray marcher. As shown in the optimized monolithic shader in Section 4.7.3, this methodology severely limits the FPS of the demonstration application at large world sizes. Instead of attempting to optimize the ray marcher to handle a large amount of distance fields, we can optimize the distance field to only store the distances and have the ray marcher calculate colours.

This approach is expected to make the FPS of the demonstration application worse; however, reducing the number of bit operations the distance field compute shader needs to carry out could improve the execution time.

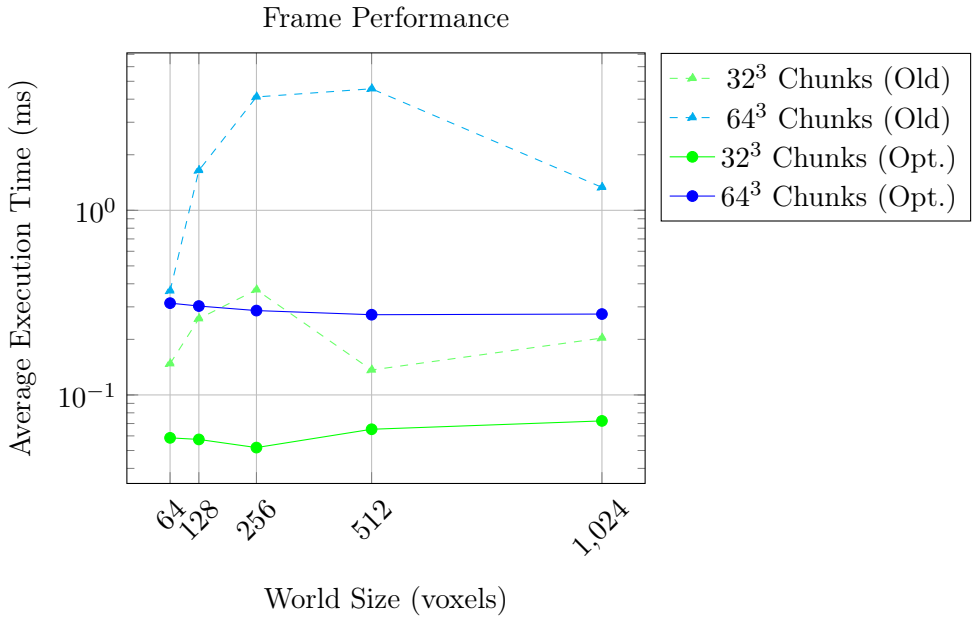


Figure 4.11: Comparison of the average execution time of the optimized monolithic shader.

4.7.4.1 Performance Results

Given the level-of-detail approach uses a focus area within which chunks are recalculated using JFA, while all others chunks use JFA as a coarse distance field calculation it is necessary to modify the demonstration application to make the distribution of the algorithm used as equal as possible.

For these performance tests, the focus is on the performance of the compute shader on single chunks. As such the world size contains 2x2 chunks, where one half of the world is in the focus area which uses FIM for accurate distance field calculation while the other half of the world uses JFA for coarse distance field calculation.

Using these modifications to the test demonstration application, we observe that the distance field execution time has increased compared to when colour information is included in the distance field - comparing Figure 4.11 and Table 4.7. We can also see that chunks up to a size of 128^3 have an execution time that would fall under the 33.33ms target; a chunk size of 256^3 also falls under 33.33ms however there would not be much time left in the frame for other tasks. The calculation of the distance field for a 128^3 chunk takes less than the delta time required to achieve 60FPS (16.67ms); in theory this would mean with a sufficiently optimized ray marcher this distance filed calculation approach could work up to 60FPS at 128^3 chunk sizes, potentially allowing for world larger than 1024^3 in voxels.

Chunk Size	16^3	32^3	64^3	128^3	256^3
Avg. Time (ms)	0.023970172	0.058316454	0.29299217	4.556905	29.263702
Std. Deviation (ms)	0.004594186	0.013979476	0.030013368	2.038477	20.3295
Confidence Interval (ms)	(0.023801085,0.024139259)	(0.058031257,0.05860165)	(0.29210484,0.2938795)	(4.457361,4.6564484)	(27.745697,30.781708)

Table 4.7: Single chunk performance of an optimized compute shader, using a pure distance field.

Chapter 5

Evaluation

This section will carry out a critical evaluation of the work undertaken in this paper, and the results of the final algorithm compared to the targets and metrics set out in the introduction and compared to other works.

5.1 Final Results

The final implementation of the compute shader had two variations:

- Monolithic compute shader with colour information encoded in the distance field.
- A pure distance field, storing only distance values.

The methodology defined that the distance field was to also encode colour information, as seen in Section 3.2, as such the final testing will be carried out using that version of the compute shader as it was found that a pure distance field places too much strain of the ray marcher and subsequently heavily affects the FPS of the application. While the ray marcher wasn't the focus of this paper, achieving a good FPS using the demonstration application was.

As the final algorithm is a combination of JFA and FIM that is selectively executed depending on whether the chunk being processed is in the focus area, to determine the single chunk performance of the algorithm a world of $2x2x2$ chunks is created with the focus area covering points $(0, 0, 0)$ to $(0, 1, 1)$; this will ensure that an even distribution of updates are JFA only and FIM only. As seen in Table 5.1, we observe that the execution time of the compute for a single chunk

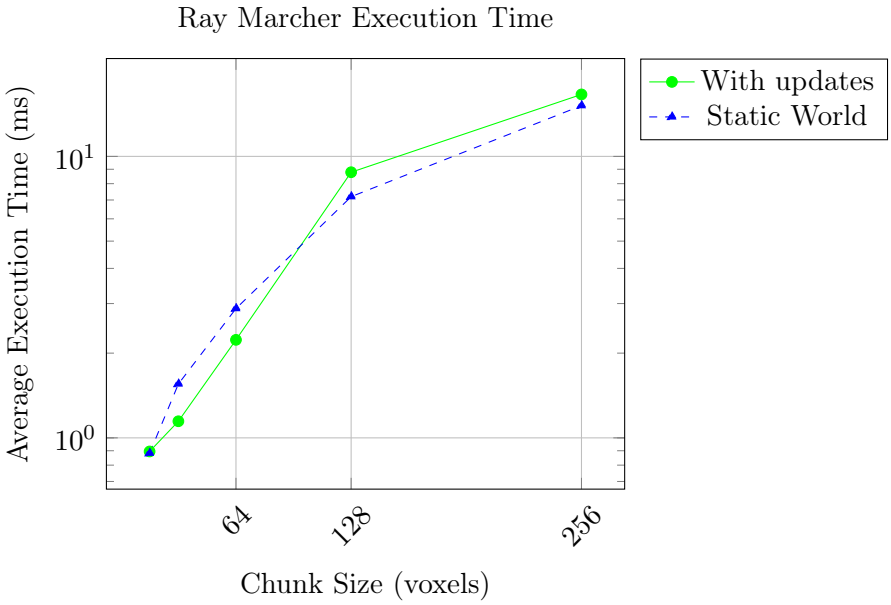


Figure 5.1: Execution time of the ray marching compute shader.

falls below the delta time for a 30 FPS application (33.33ms) for chunk sizes as high as 256^3 - in theory this achieves the aim of having a distance field compute shader that can be executed within a single frame with time left over for other tasks such as rendering at 30 FPS.

While the execution time of the compute shader remains under 33.33ms, the actual FPS of the demonstration application using 256^3 chunks is only 1.42 FPS, or only 9.47 FPS for 128^3 chunk sizes, as seen in Table 5.1; this does not meet the aim of achieving 30 FPS in the demonstration application. The ray marcher compute shader, while not the focus of this paper, does impose a significant performance penalty as seen in Figure 5.1. Combining the execution time of the distance field and ray marcher compute shader, the FPS of the demonstration application matches what was seen in Table 5.1.

Chunk Size	16^3	32^3	64^3	128^3	256^3
Avg. Time (ms)	0.022988718	0.057287574	0.29529586	4.571752	29.337261
Std. Deviation (ms)	0.00283882	0.013250402	0.036815774	1.7831647	18.889853
Avg. FPS	163.57758	164.06527	65.23792	9.47038	1.42479

Table 5.1: Single chunk performance of an optimized compute shader.

The use of a discrete distance field means every voxel in the world needs to be represented; in the final implementation each voxel is represented by an "*uint*" data type which is 4 bytes. For

a world of size 1024^3 that is a memory requirement of 4GB to represent the world; this needs to be double as the world is currently represented twice, once as a distance field, and as a raw voxel grid meaning a total of 8GB is required to represent the world. This places a hard-limit to the size of the world that can be tested using the hardware available during the testing of this paper.

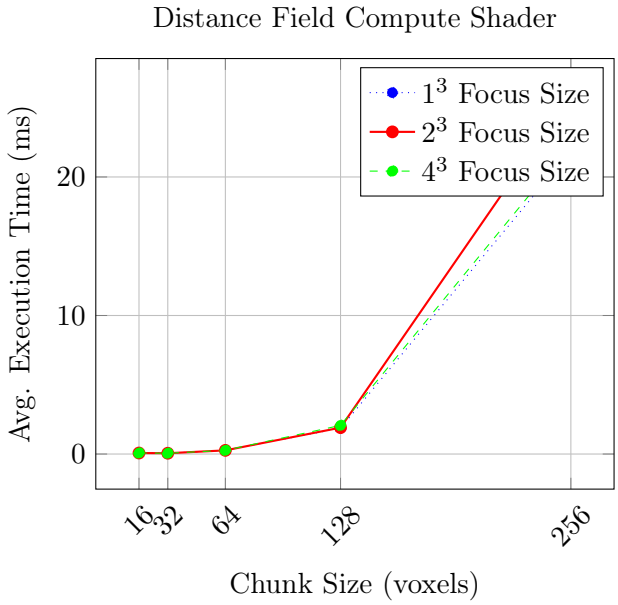
We can see in Figure 5.2 that the distance field execution time of the compute shader at a world size of 1024^3 falls below the delta time of a 30 FPS application at all chunk sizes tested, and even below 60 FPS for chunk sizes equal to or smaller than 128^3; however, the ray marcher compute shader is a significant bottleneck and results in a frame rate that does not meet the minimum target of 30 FPS, at any focus size and chunk size combination.

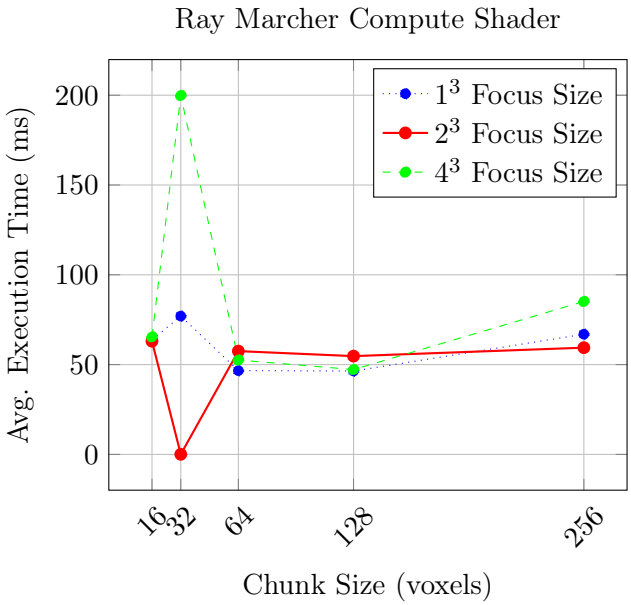
5.1.1 Aims and Objectives

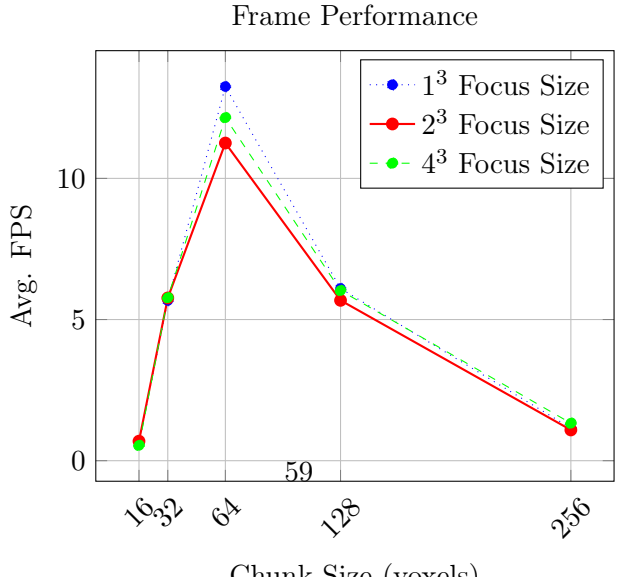
The aims and objectives, as set out in Section 1.2, were partially met.

- **Analysis and classification of existing approaches:** Existing approaches such as brute-force, chunking, fast iterative method, and jump flooding algorithm were analyzed and classified.
- **Development of a new algorithm:** A new novel algorithm was **not** developed; however, a combination of existing algorithms was developed in a way that made dynamic distance fields viable.
- **Implementation and validation:** All mentioned approaches were implemented and validated using the comparison framework in great detail.
- **Comprehensive comparison framework:** A demonstration application for testing was created with clear reproducible steps and clearly gathered performance metrics.

A key objective was to be able to achieve 30 FPS in the demonstration application and have a distance field algorithm execution time of less than 33.33ms. The algorithm execution exceeds this target by a significant amount; however, the FPS target was not met due to a significant limitation in the ray marcher.


Distance Field Compute Shader
Avg. Execution Time (ms)
0
10
20
16
32
64
128
256
Chunk Size (voxels)
1^3 Focus Size
2^3 Focus Size
4^3 Focus Size


Ray Marcher Compute Shader
Avg. Execution Time (ms)
0
50
100
150
200
16
32
64
128
256
Chunk Size (voxels)
1^3 Focus Size
2^3 Focus Size
4^3 Focus Size


Frame Performance
Avg. FPS
0
5
10
16
32
64
128
256
Chunk Size (voxels)
1^3 Focus Size
2^3 Focus Size
4^3 Focus Size

5.1.2 Limitations and Difficulties

The parallel nature of GPUs is a significant advantage as work can be more efficiently distributed; this becomes important when dealing with a large amount of data as required with a dense distance field and voxel grid. The dense representation of the voxel grid requires a large amount of memory with each individual voxel requiring 4 bytes of storage, once for the voxel grid, and once for the corresponding distance field. As such the world size is limited by the amount of available memory on the GPU.

This paper set out to show the viability of using GPU distance field for representing voxel grids in a visual application; this required the implementation of some form of rendering algorithm for displaying the results of the distance field. As described in the final results, the implemented ray marcher became a significant bottleneck in the rendering performance and as such FPS of the demonstration application is not optimal. Optimizing the ray marcher was not a focus or objective of this paper, as such the performance of the ray marcher was not improved, but the next limitation in this implementation is the ray marcher compute shader.

5.1.2.1 Algorithm Implementation

Implementing my algorithm using Vulkan's compute pipeline introduced several challenges. Initially, the two sub-algorithms were written as separate compute shaders, which made the development and testing process more modular and manageable. However, this approach required frequent synchronization and dispatch coordination on the host, leading to performance overhead. To address this, I combined both sub-algorithms into a single "monolithic" compute shader to minimize host-side synchronization and reduce pipeline barriers. While this improved runtime efficiency, it introduced significant complexity in both the shader and the host code. The monolithic shader relied on numerous tightly coupled memory buffers and descriptor sets, many of which were only relevant to specific parts of the shader but had to be created, managed, and bound regardless. This made it difficult to maintain clear ownership and lifecycle management of resources in the host code. Furthermore, in cases where only one sub-algorithm was executed, unnecessary resources were still allocated and bound, leading to some inefficiencies and complicating the overall system design.

5.1.2.2 Vulkan

Although Vulkan offered the low-level control and potential performance benefits required for my algorithm, its complexity introduced significant overhead during development. The verbosity of the API made even simple operations tedious and error-prone. While writing the compute shader itself was relatively straightforward, careful management of memory layout, alignment, and memory residency was required, often leading to subtle bugs and time-consuming debugging.

On the host side, the complexity of managing object lifetimes — particularly for resources like descriptor sets, pipelines, and buffers — significantly increased development effort. Some resources were static, others needed to be updated frequently, and distinguishing between the two added additional design complexity. In hindsight, a higher-level API such as OpenGL might have been a more pragmatic choice for the purposes of demonstration, offering quicker setup and faster iteration times. However, given the memory- and compute-intensive nature of the algorithm, it is likely that Vulkan's fine-grained control ultimately allowed for better performance and scalability than OpenGL could have provided.

5.1.3 Comparison to Related Works

Sparse Voxel Octrees are a commonly used representation for voxel grids as they are highly memory efficient but have typically not been great for dynamic worlds. The works of Pan, Yucong (Pan 2021) experimented with the development of an algorithm for handling dynamic updates to SVOs; for their testing they used a world size of 512^3 with a resolution of $1280x720$ for the ray marcher and achieved a dynamic update time of $22ms$ and a ray casting time of 10.04 in their Buddha and Dragon scenario.

The implementation in this paper does not include a voxelization feature, as such a directly comparable test cannot be carried out; however, using a similarly sized world and renderer resolution we can see the algorithm in this paper is able to out-perform a dynamic SVO as seen in Table 5.2 in execution time at various focus sizes and the ray marcher has similar performance.

	This Paper	Dynamic SVO
Focus Size 2^3		
Update Time (ms)	0.27185795	22.23
Ray Marcher (ms)	9.462824	10.04
Focus Size 4^3		
Update Time (ms)	0.27252674	22.23
Ray Marcher (ms)	9.663957	10.04
Focus Size 8^3		
Update Time (ms)	0.2706304	22.23
Ray Marcher (ms)	10.29717	10.04

Table 5.2: Comparison of a hybrid JFA and FIM distance field to a dynamic SVO.

Chapter 6

Conclusion

A final discussion on the results of the findings in this paper, and the improvements and further work that could be done.

6.1 Reflection

This paper has found an algorithm for generating distance fields from a voxel grid for use in a dynamic world; being the primary objective of this paper we can say it has been a success. Other methods for generating distance fields were also discussed and the differences between them, and the potential for performance improvements was also discussed as various improvements and optimizations were applied over the course of implementation. A deep understanding of distance fields and the algorithms used to generate them was developed, as well as a detailed guide for how those algorithms work and their limitations that could serve as a template, or starting point, for others.

A visual demonstration application, in the form of a voxel "drawing" game, was developed that allowed for a testing framework to be developed to be accurately test the performance of various algorithms. It also served as a way to identify whether the algorithms being used created an accurate distance field without artifacts. While the FPS of the demonstration application suffers at very large world sizes, it serves as a solid base from which a more complex voxel driven game could be developed using distance fields for rendering.

The rendering of distance fields, and the memory intensity of them, was not entirely understood before undertaking this work; the end result was the discovery that a naive ray marcher becomes

a significant bottleneck due to the large amount of data the GPU needs to have read access to and the cost of binding that data to the ray marching compute shader.

6.2 Further Work

As previously discussed, the distance fields, and raw voxel grid, require a large amount of memory; sparse voxel octrees are a memory efficient way to represent voxel grids but as discovered in this paper updating a distance field is more performant. It would be beneficial to identify a way to improve the memory efficiency of the distance field representation, this could include:

- A hybrid SVO and distance field approach allowing for sparse representation in areas where high fidelity is not required.
- Distance field compression, or run-length encoding. Similar to how SVOs achieve their high compression values, a distance field may be able to be compressed to reduce its memory footprint.

The ray marching of several large distance fields was also identified as a bottleneck in this paper; however, this could be remedied by improving the representation by using the suggested approaches above for improving the memory efficiency of the distance field. Alternatively, a better suited ray marching approach could be implemented instead as there are various techniques for only casting rays that have high significance that are already used in ray casting through a voxel world such as cone tracing.

Bibliography

Akenine-Moller, T., Haines, E. & Hoffman, N. (2019), *Real-time rendering*, AK Peters/crc Press.

Amanatides, J., Woo, A. et al. (1987), A fast voxel traversal algorithm for ray tracing., *in* 'Eurographics', Citeseer, pp. 3–10.

Claypool, K. T. & Claypool, M. (2007), 'On frame rate and player performance in first person shooter games', *Multimedia systems* **13**(1), 3–17.

Crassin, C., Neyret, F., Lefebvre, S. & Eisemann, E. (2009), Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering, *in* 'Proceedings of the 2009 symposium on Interactive 3D graphics and games', pp. 15–22.

Dolonius, D. (2018), *Sparse Voxel DAGs for Shadows and for Geometry with Colors*, Chalmers Tekniska Hogskola (Sweden).

Erleben, K. & Dohlmann, H. (2008), 'Signed distance fields using single-pass gpu scan conversion of tetrahedra', *Gpu Gems* **3**, 741–763.

Fuhrmann, A., Sobotka, G. & Groß, C. (2003), Distance fields for rapid collision detection in physically based modeling, *in* 'Proceedings of GraphiCon', Vol. 2003, Citeseer, pp. 58–65.

Gorobets, A. V. (2023), 'An approach to the implementation of the multigrid method with full approximation for cfd problems', *Computational Mathematics and Mathematical Physics* **63**(11), 2150–2161.

Hadji-Kyriacou, A. & Arandjelović, O. (2021), 'Raymarching distance fields with cuda', *Electronics* **10**(22), 2730.

Jones, M. W. & Satherley, R. (2001), Using distance fields for object representation and rendering, *in* 'Proc. 19th Ann. Conf. of Eurographics (UK Chapter)', London, pp. 37–44.

Kämpe, V., Sintorn, E. & Assarsson, U. (2013), 'High resolution sparse voxel dags', *ACM Transactions on Graphics (TOG)* **32**(4), 1–13.

Keller, A., Viitanen, T., Barré-Brisebois, C., Schied, C. & McGuire, M. (2019), Are we done with ray tracing?, *in* 'SIGGRAPH Courses', pp. 3–1.

Kramer, L. (2015), Real-time sparse distance fields for games, *in* 'Game Developers Conference'.

Laine, S. & Karras, T. (2010), Efficient sparse voxel octrees, *in* 'Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games', pp. 55–63.

Mileff, P. & Dudra, J. (2019), 'Simplified voxel based visualization', *Production Systems and Information Engineering* **8**, 5–18.

Naylor, B. F. (1992), Interactive solid geometry via partitioning trees, *in* 'Proc. Graphics Interface', Vol. 92, pp. 11–18.

Pan, Y. (2021), Dynamic update of sparse voxel octree based on morton code, Master's thesis, Purdue University.

Rong, G. & Tan, T.-S. (2006), Jump flooding in gpu with applications to voronoi diagram and distance transform, *in* 'Proceedings of the 2006 symposium on Interactive 3D graphics and games', pp. 109–116.

Rong, G. & Tan, T.-S. (2007), Variants of jump flooding algorithm for computing discrete voronoi diagrams, *in* '4th international symposium on voronoi diagrams in science and engineering (ISVD 2007)', IEEE, pp. 176–181.

Sethian, J. A. (1999), 'Fast marching methods', *SIAM review* **41**(2), 199–235.

Sinharoy, B. & Szymanski, B. (1993), Finding optimum wavefront of parallel computation, *in* '[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences', Vol. 2, IEEE, pp. 225–234.

Söderlund, H. H., Evans, A. & Akenine-Möller, T. (2022), 'Ray tracing of signed distance function grids', *Journal of Computer Graphics Techniques Vol* **11**(3).

Stevenson, R. & Navarro, C. A. (2022), 'Gpu voronoi diagrams for random moving seeds', *arXiv preprint arXiv:2209.00117* .

Tan, Y. W., Chua, N., Koh, C. & Bhojan, A. (2022), 'Rtsdf: Real-time signed distance fields for soft shadow approximation in games', *arXiv preprint arXiv:2210.06160* .

Teodoro, G., Pan, T., Kurc, T. M., Kong, J., Cooper, L. A. & Saltz, J. H. (2013), 'Efficient irregular wavefront propagation algorithms on hybrid cpu–gpu machines', *Parallel computing* **39**(4-5), 189–211.

Truong-Hong, L. & Laefer, D. F. (2014), 'Octree-based, automatic building facade generation from lidar data', *Computer-Aided Design* **53**, 46–61.

van Wingerden, T. (2015), Real-time ray tracing and editing of large voxel scenes, Master's thesis.

Vulkano (2024), 'Vulkano examples: Interactive fractal'. Accessed: 2025-04-27.
URL: `https://github.com/vulkano-rs/vulkano/tree/846c373daa3101b1c0287318df71b91102a01c2c/examples/interactive-fractal`

Wang, J., Ino, F. & Ke, J. (2023), Prf: A fast parallel relaxed flooding algorithm for voronoi diagram generation on gpu, *in* '2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)', IEEE, pp. 713–723.

Wright, D. (2015), Dynamic occlusion with signed distance fields, *in* 'ACM SIGGRAPH', Vol. 3.

Xu, C., Wang, T. Y., Liu, Y.-J., Liu, L. & He, Y. (2015), 'Fast wavefront propagation (fwp) for computing exact geodesic distances on meshes', *IEEE transactions on visualization and computer graphics* **21**(7), 822–834.