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

Write a summary of the work presented in your dissertation. Introduce the topic and highlight your main contributions and results. The abstract should be comprehensible on its own, and should no contain any references. As far as possible, limit the use of jargon and abbreviations, to make the abstract readable by non-specialists in your area. Do not exceed 300 words.

Acknowledgements

Write any personal words of thanks here. Typically, this space is used to thank your supervisor for their guidance, as well as anyone else who has supported the completion of this dissertation, for example by discussing results and their interpretation or reviewing write ups. It is also usual to acknowledge any financial support received in relation to this work.

Contents

List of Figures

List of Tables

Glossary

P	Placeholder

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 "Falling Sand" graphics application will be implemented that will serve as a demo and benchmark for real-time performance of the distance field regeneration. The simulation itself will not be optimized and include a bare minimum of sand and water voxels that can fall and spread out in the world.

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).

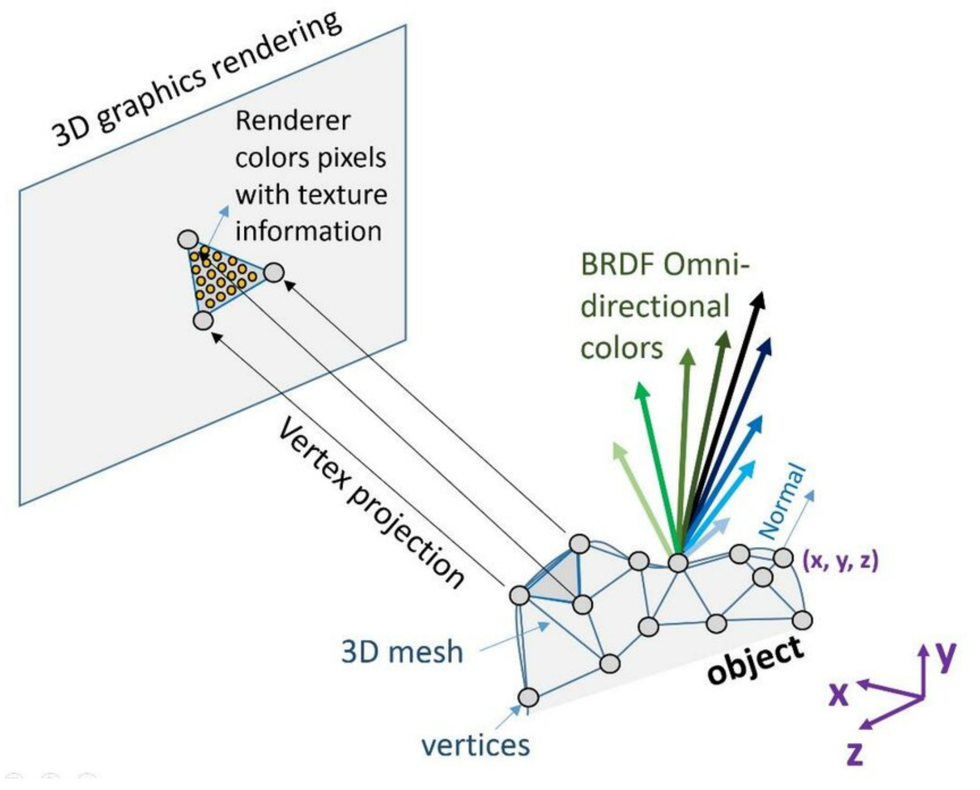


Figure 2.1: Illustration of how a collection of triangles get rasterized onto a screen as demonstrated by Lafruit et. al 2016

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 framerates (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.

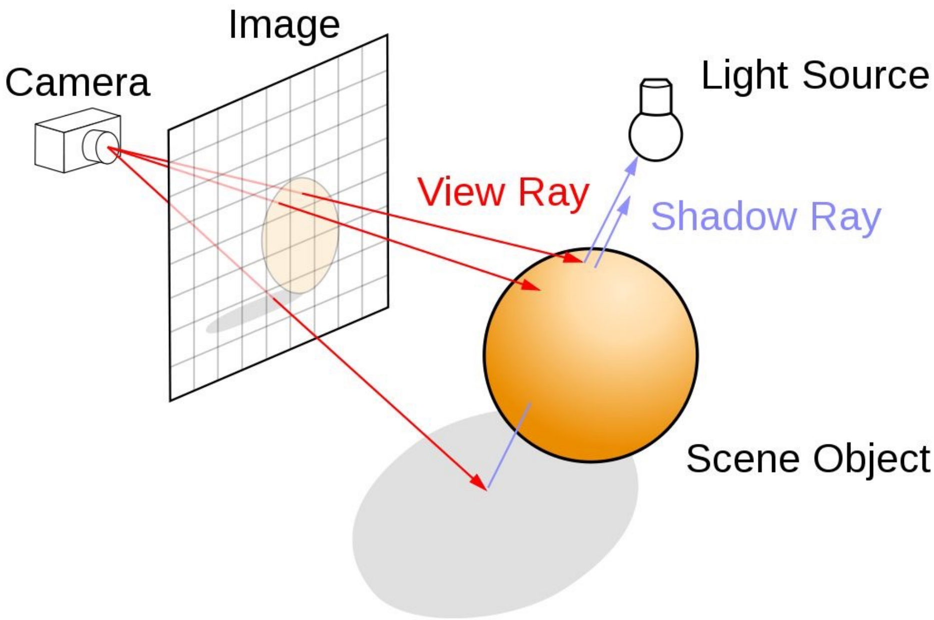


Figure 2.2: Visualisation of how ray-casting is used to rasterize a 3D world. Source: Wikipedia

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 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.

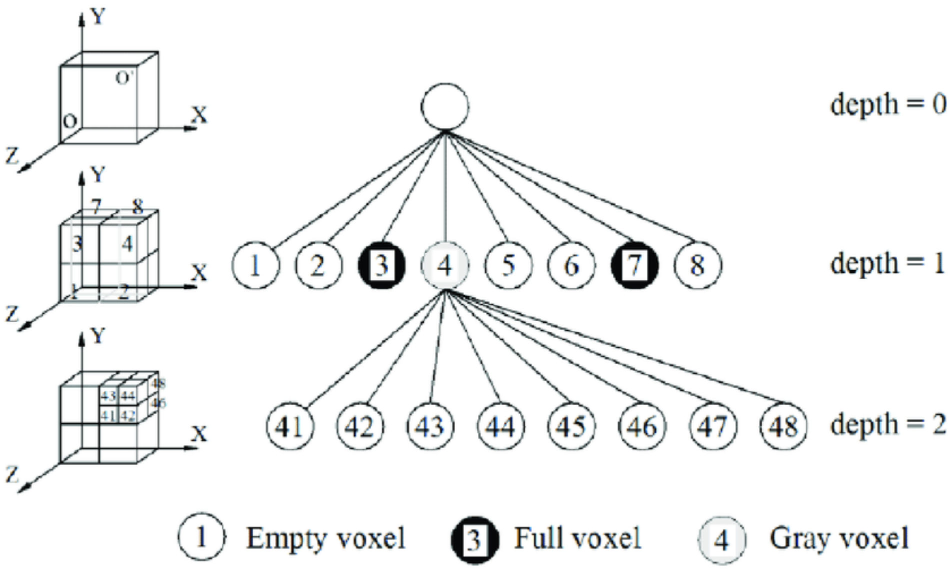


Figure 2.3: Illustration of the structure of a Sparse Voxel Octree, nodes with more details have additional subdivisions. (Truong-Hong & Laefer 2014)

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.

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

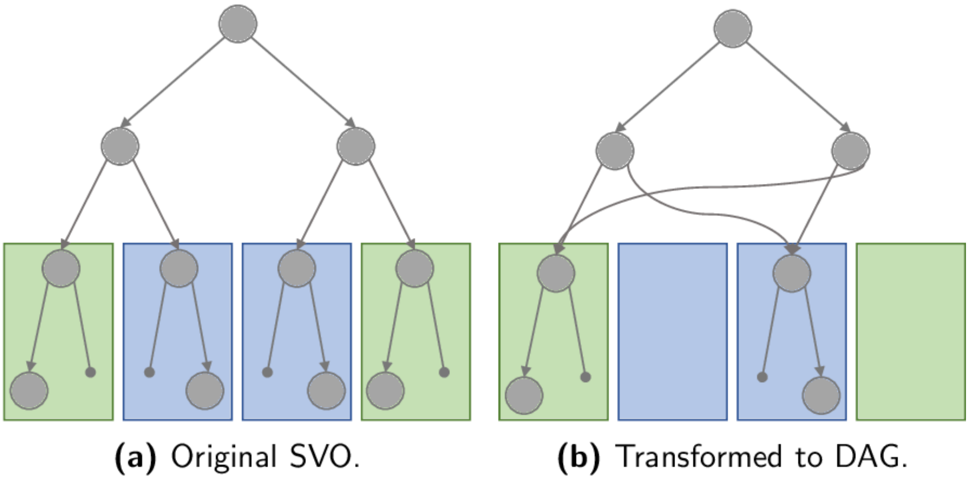


Figure 2.4: Comparison of a octree versus the compressed format of a SVDAG, illustrated as a quad tree for brevity. (Dolonius 2018)

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).

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).

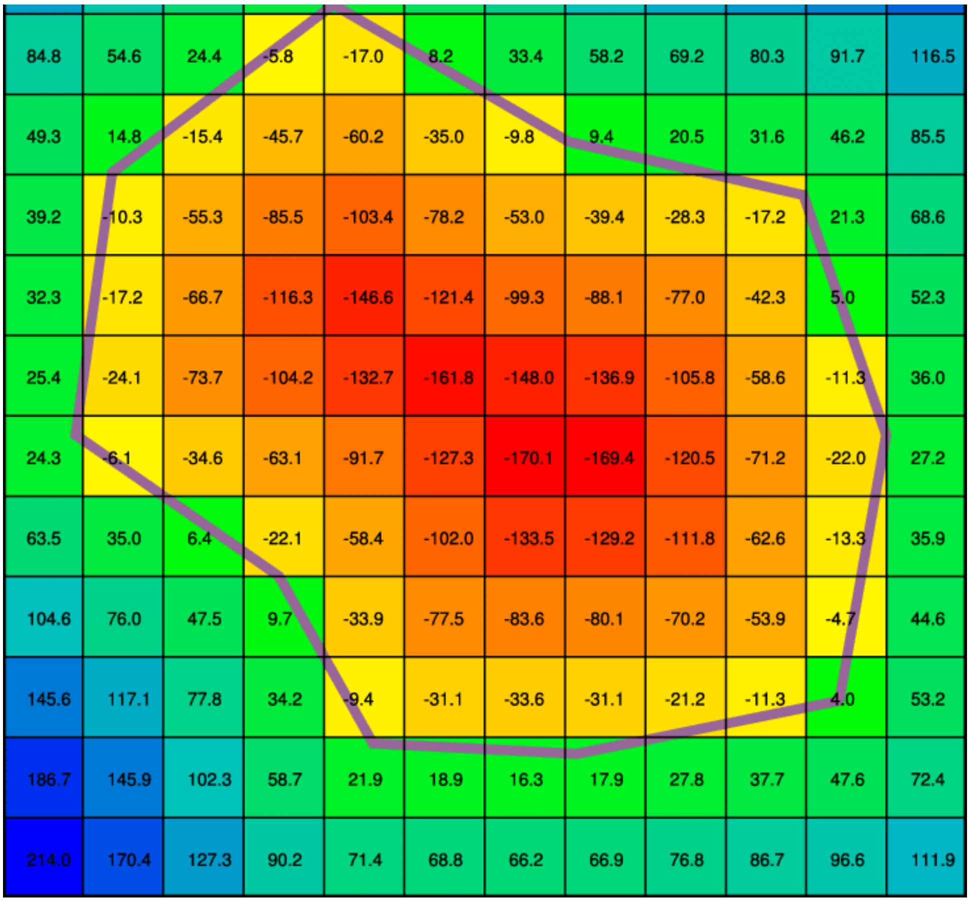


Figure 2.5: Illustration of a discrete signed distance field grid. Negatives values indicate a cell inside the shape, positive values indicate a cell outside of the shape.

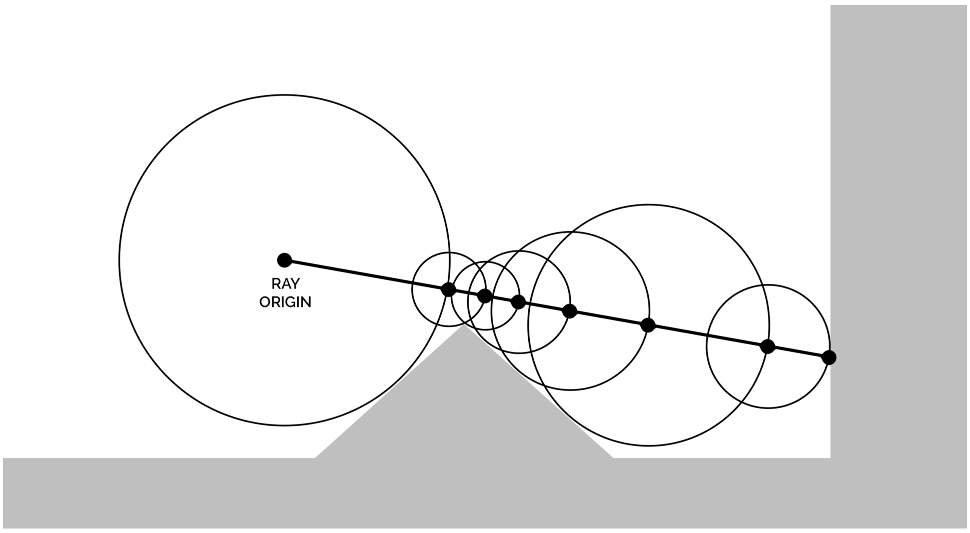


Figure 2.6: Example of a ray marching from a camera using sphere ray marching and a signed distance function. The ray will query the function for the distance to the nearest object, and advance by that amount.

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.

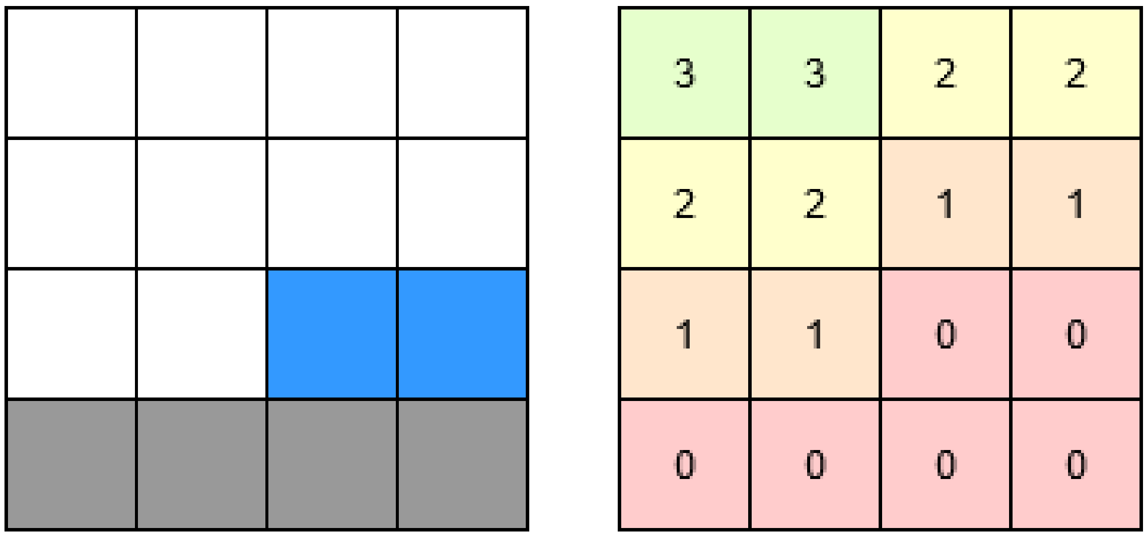
3.1 World Representation

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.



(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 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 the distance to the nearest solid voxel using the Manhattan distance, the lowest 8 bits define

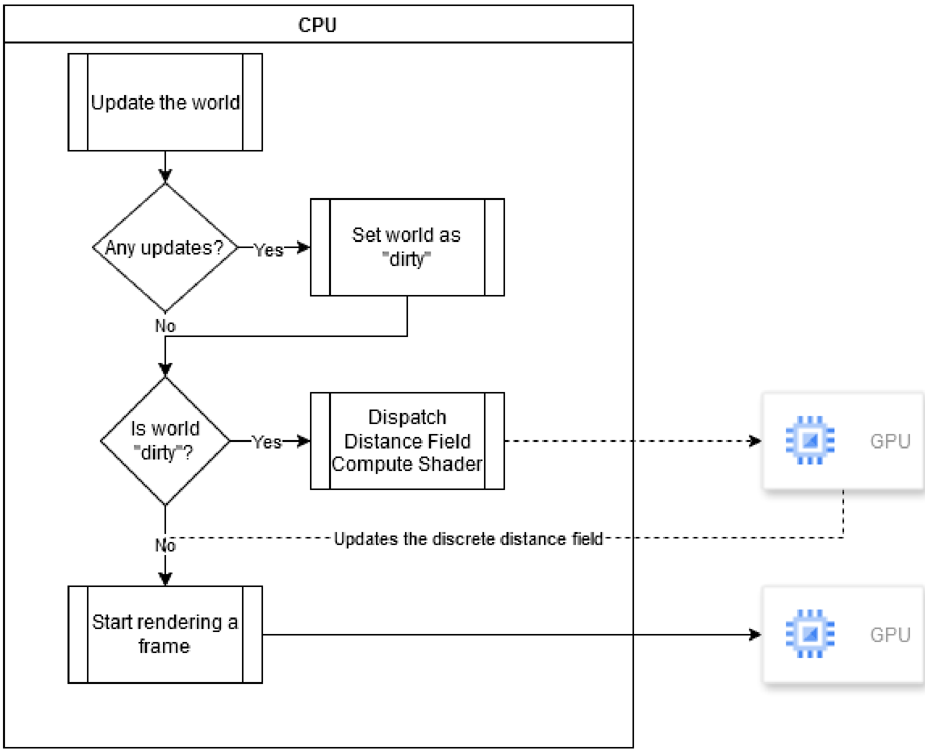


Figure 3.2: Illustration of the computational workflow required in deciding when to update the discrete distance field of a world.

the colour of the voxel in a compressed RGB332 format; voxel colours are hard-coded into the distance field as can be seen in 3.3.

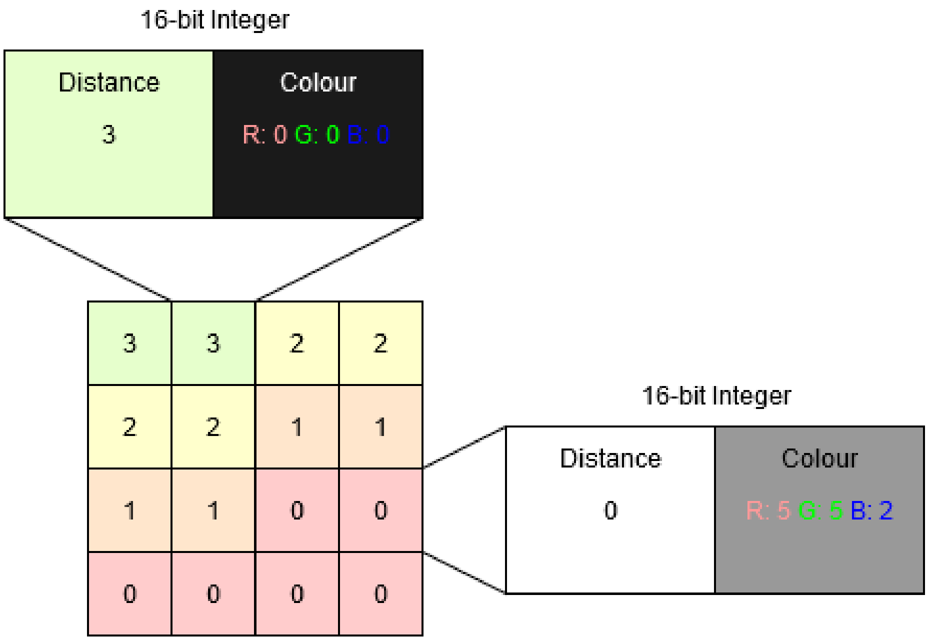


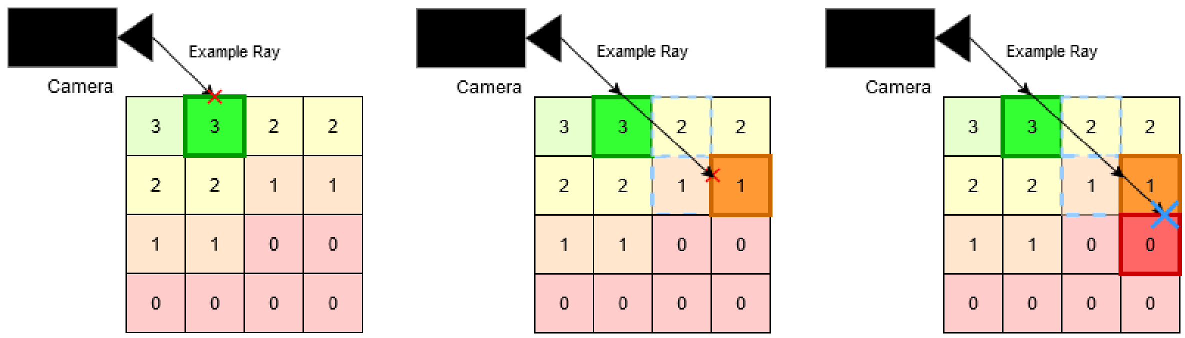
Figure 3.3: Based on the same distance field as in 3.1b, the underlying integer of an empty and solid voxel are shown.

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 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 3.5. The application will contain a voxel finite world that can be interacted with, voxels can be placed and deleted by the user at will. Voxels of different colours can be placed in the world.

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.

3.5 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 with the falling sand simulation; this should take into account the efficiency which the distance field computation is carried out and the effect it has on rendering.

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.

$$\mathrm{FPS} = \frac{1000}{\mathrm{dt_{ms}}} \tag{3.1}$$

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 simulation update and rendering 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:

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. This, 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. 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.

The results in Table 4.2 highlight how a brute-force approach is unsuitable for large dynamic

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.

worlds. A common optimization is chunking to split a large world into smaller "chunks" as described in 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

```
if x, y, z is within bounds of V then
    return V[x][y][z]
else
    return Solid voxel
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. 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.

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) is an approach 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 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 **??** 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 FMM must be executed on the distance field several times until shortest distances have calculated for all voxels, the effect of several iterations can be seen in Figure 4.1.

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 $O(8N)$.

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

Algorithm 4 Fast Marching Method, Distance Field Initialization

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
    D[x][y][z] ← 0
else
    D[x][y][z] ← N * 2
end if
```

Algorithm 5 Fast Marching Method

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

Output: Distance field grid D

`voxel` $\leftarrow$ `Get Voxel at` (x, y, z)
if `voxel is solid` **then**
    Return: D
end if
Initialize $d_{min} \leftarrow N * 2$
for neighbours n of voxel **do**
    `neighbour` $\leftarrow$ `Get Voxel at` n
    if `neighbour is solid` **then**
        $d_n \leftarrow 0$
    else
        $d_n \leftarrow$ distance value at n
    end if
    $d_n \leftarrow d_n + 1$
    $d_{min} \leftarrow \min(d_{min}, d_n)$
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

FMM 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.

Single chunk performance is observed to be excellent; however, at world sizes larger than the chunk size the performance degrades substantially, this can be observed in Figure 4.1. This is

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 FMM algorithm.

due to the way synchronization is handled by the CPU to achieve distance field convergence; there is a substantial amount of time waiting for fences to be signalled before the compute shader can be executed again. This could be mitigated by refactoring the compute shader such that synchronization between iterations is handled by the GPU reducing the need for expensive memory transfers and extensive waiting on the CPU. The Fast Marching Method is inherently sequential and so is not well suited to parallelism on the GPU, as such effort will instead be spent on implementing the Fast Iterative Method as described in section.

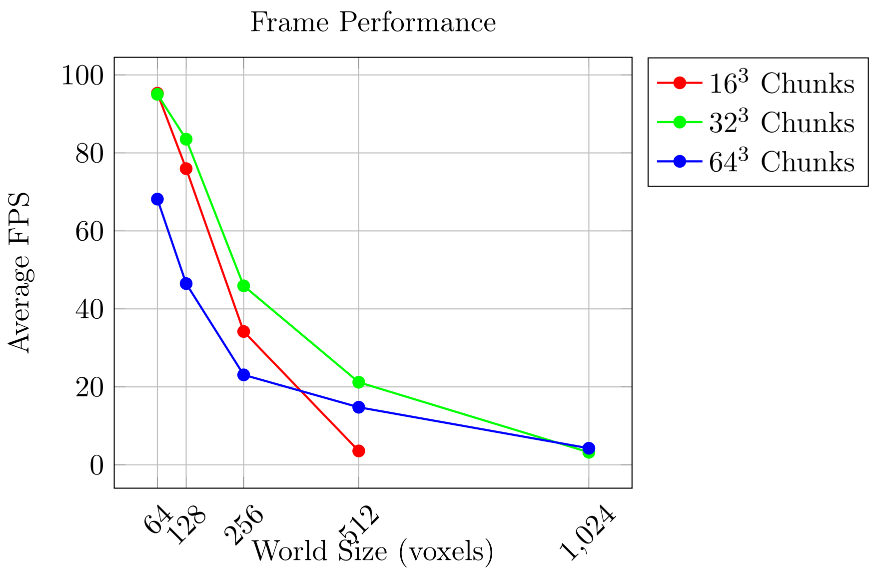
4.4 Jump Flooding Algorithm

- Well-suited for GPU
- Only provides approximations
- Quick but not exact
- Not useful for close-up distance fields, i.e. where player is close. But could be useful for seeding, or initializing distance fields with an approximation hence speeding up the algorithm run on the distance field.

4.4.1 Performance Results

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.



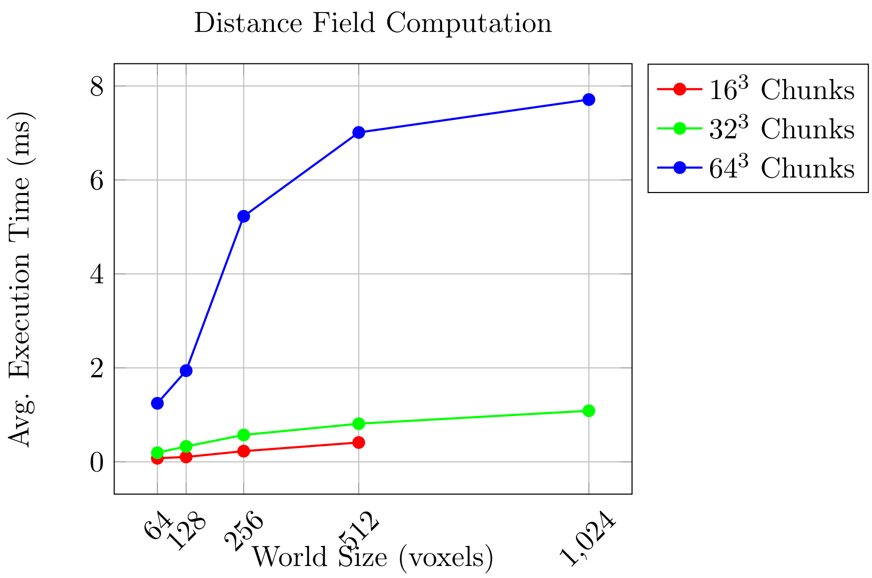
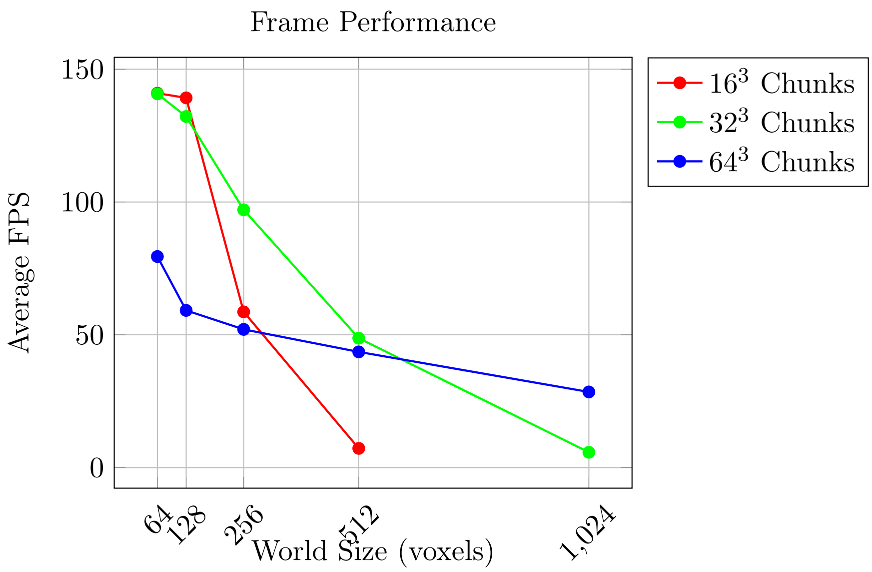


Figure 4.1: Comparison of the performance of the Fast Marching Method at different world and chunk sizes.



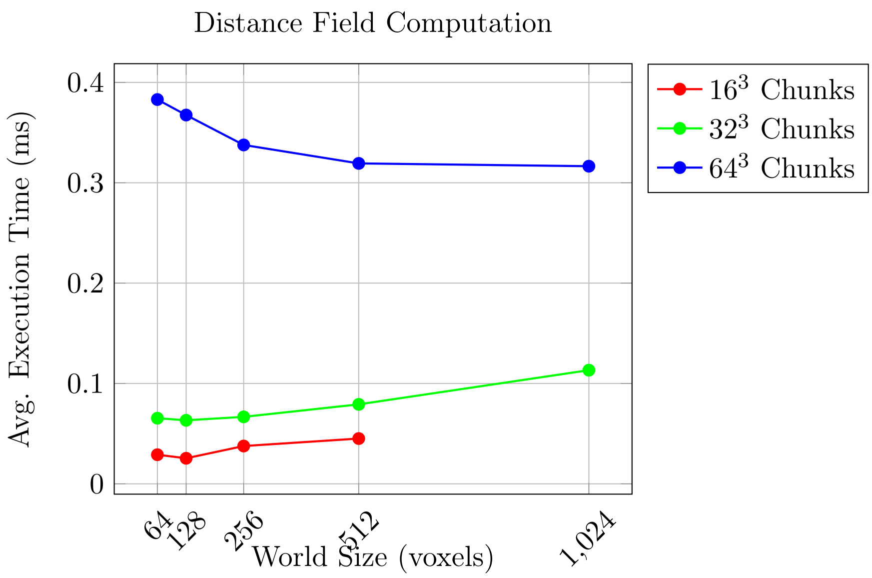


Figure 4.2: Comparison of the performance of the Jump Flooding Algorithm at different world and chunk sizes.

4.5 Coarse JFA with FIM Refinement

1. Use JFA to initialize the distance field
2. Then use FIM to converge the distance field to an exact result
3. Decrease the average number of iterations required for convergence
4. Slightly faster
5. Given the demonstration application, you can selectively choose the algorithm depending on the distance of the update to the camera. Far away updates can use JFA to quickly update the distance field, at a distance artifacts shouldn't be noticeable, while updates near the player can use just FIM to propagate the update within the chunk. Both algorithms only need to be used when a chunk's distance field is first calculated.

4.5.1 Performance Results

Divided depending on how this combination of algorithms is deployed.

4.5.1.1 Complete Recalculation using both JFA and FIM

When a chunk needs the distance field recalculated, JFA is run to initialize the distance field, and then FIM is run afterward to produce an exact result.

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 Selective Algorithm Execution

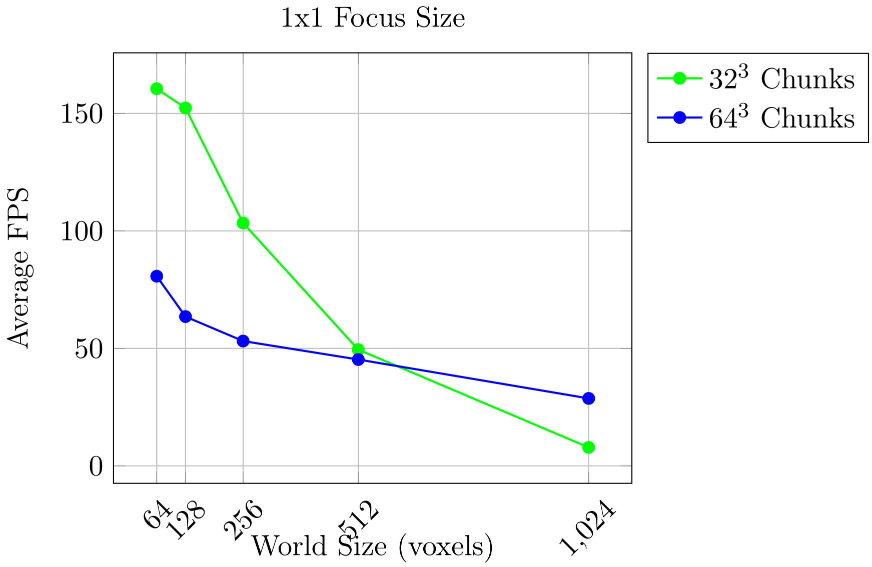
Choose what type of algorithm to execute:

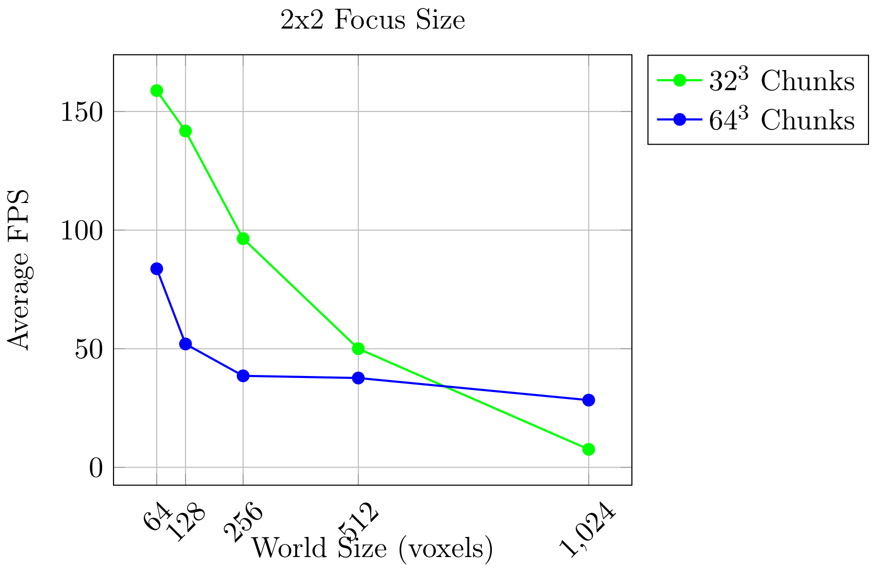
1. Hybrid - only needed for first computation. Uses JFA for initialization, and FIM for convergence.

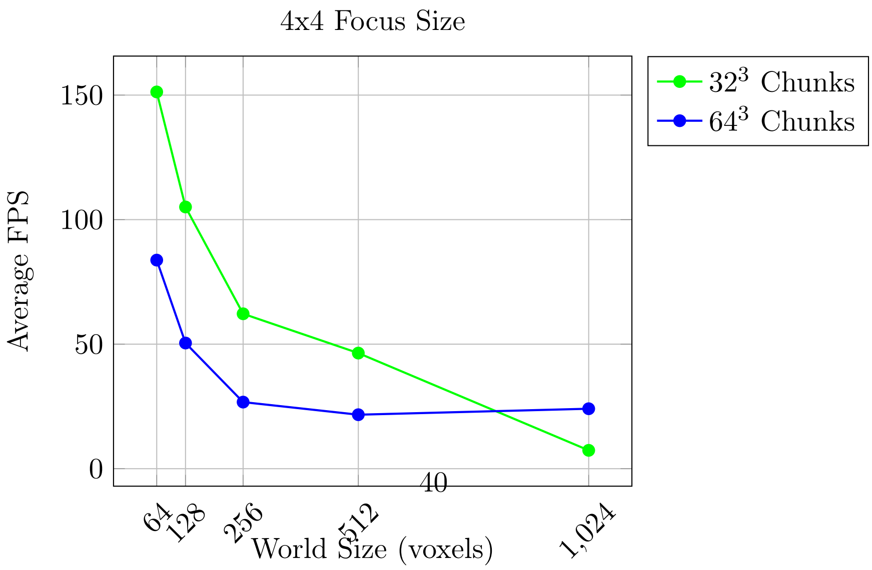
2. Pure JFA - can be used to initialize, or update, far away chunks.

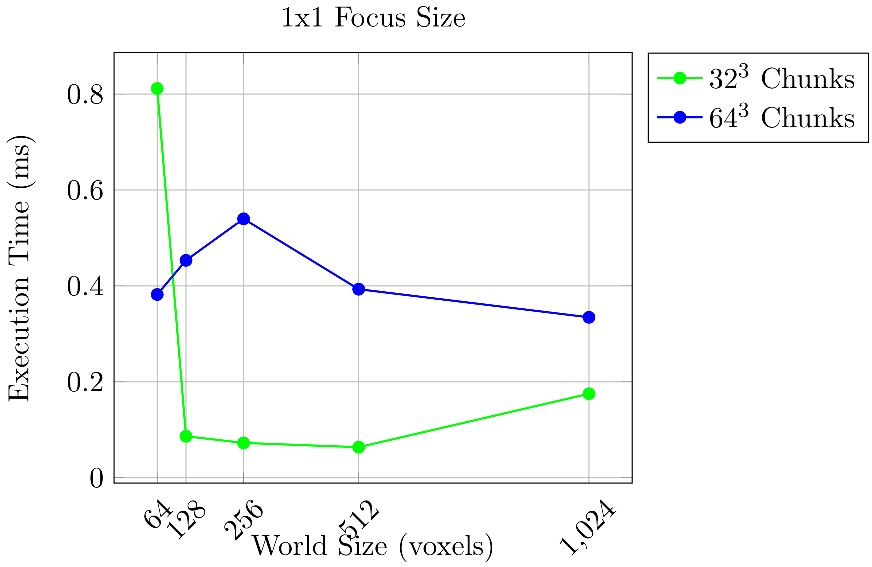
3. Pure FIM - can be used to update chunks with the most visual impact i.e. nearby chunks.

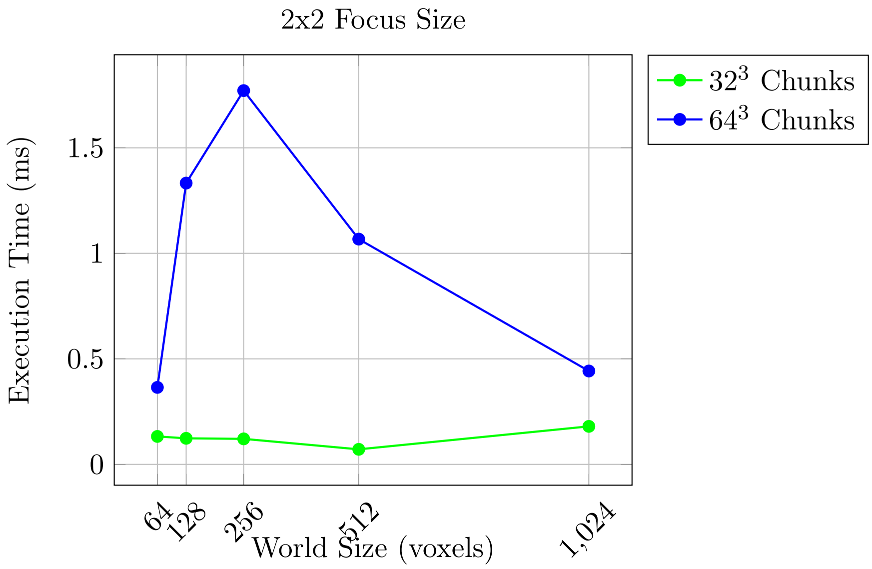
4.7 Performance Results

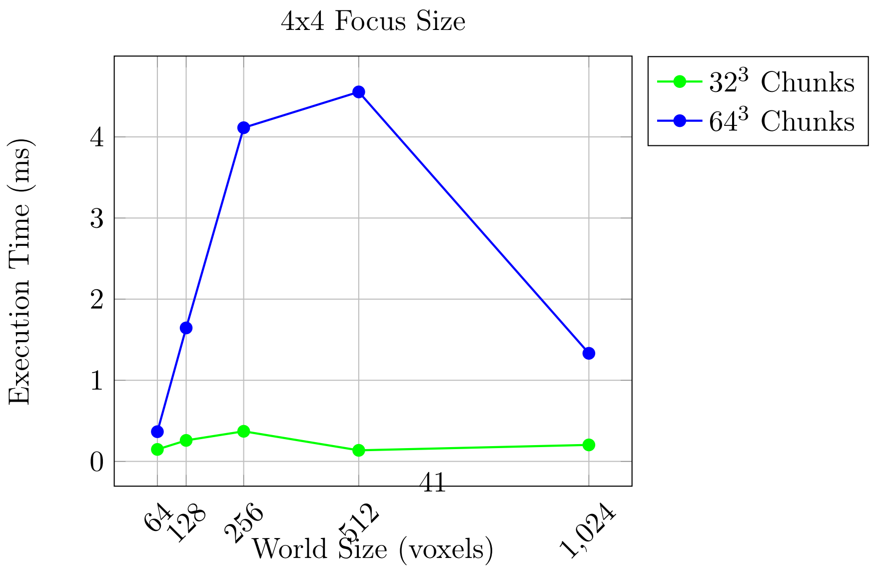

1x1 Focus Size
32^3 Chunks
64^3 Chunks
Average FPS
0
50
100
150
64
128
256
512
1,024
World Size (voxels)

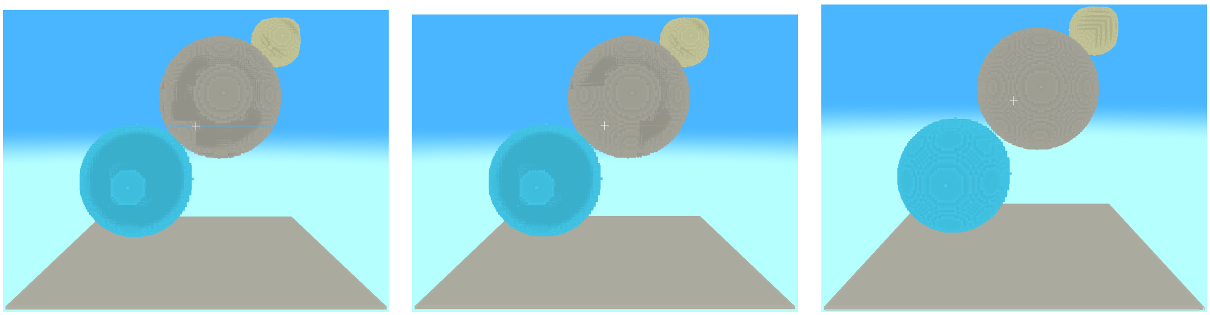

2x2 Focus Size
32^3 Chunks
64^3 Chunks
Average FPS
0
50
100
150
64
128
256
512
1,024
World Size (voxels)


4x4 Focus Size
32^3 Chunks
64^3 Chunks
Average FPS
0
50
100
150
64
128
256
512
1,024
World Size (voxels)


1x1 Focus Size
Execution Time (ms)
0
0.2
0.4
0.6
0.8
64
128
256
512
1,024
World Size (voxels)
32^3 Chunks
64^3 Chunks


2x2 Focus Size
Execution Time (ms)
0
0.5
1
1.5
64
128
256
512
1,024
World Size (voxels)
32^3 Chunks
64^3 Chunks


4x4 Focus Size
Execution Time (ms)
0
1
2
3
4
64
128
256
512
1,024
World Size (voxels)
32^3 Chunks
64^3 Chunks



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

(b) A 2x2 focus size, at the center of the world. Accurate distance field in the center but visible artifacts at the world edges.

(c) No artifacts present when the focus size encompasses the whole world.

Figure 4.5: Comparison of the rendering artifacts and distance inaccuracies introduced by utilizing JFA in regions outside the focus point.

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.

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.