Volumetric data structures for real-time ray tracing

Utrecht University
Master's Thesis in Game and Media Technology

In collaboration with **Traverse Research**

Author: Rosalie de Winther (1326236)
Company supervisor: Jacco Bikker
Primary supervisor: Alex Telea
Secondary supervisor: Peter Vangorp
September 29, 2023

Abstract

Volumetric effects such as clouds, explosions, smoke, and fog are important scene elements for computer games. While these can be efficiently handled in a rasterizer, path tracers typically struggle to render them efficiently. In this thesis we provide the required prior knowledge to think about the different trade-offs of volume data structures, and we provide a specialized implementation for compressed density data. The resulting data structure reduces the voxel data memory footprint by a factor of 8 to 16 times over 16-bit floating point values. This is achieved by utilizing a novel method of storing density data in block-compressed textures and deduplicating homogeneous nodes. These methods allow us to store animation sequences in game-ready asset sizes and render them at real-time frame rates.

Acknowledgments

Firstly I would like to thank Traverse Research for accommodating my research and providing incredible guidance. Secondly, I want to thank Jacco Bikker for inviting me to help organize High Performance Graphics 2023, and countless discussions about voxels, rendering and GPUs. Tertiary, I would like to thank Alex Telea for his very critical look at all the documents that I made during the past year and for always being able to help with any questions regarding my thesis, it was very helpful and taught me a lot. Fourthly, I want to thank Emilio Laiso for all the collaboration on the implementation side, and for integrating our two systems to allow for (in my opinion) very pretty renders. Furthermore, I would like to thank Manon Oomen for her support during my thesis and the nitty-gritty detailed explanations of GPU performance. And of course, I want to thank Devon Dissel for always being there to cheer me up and dealing with my long days away from home.

I also want to thank some communities that have helped me through the process of writing this thesis either on an emotional or technical level. Namely, The Revenant Rebellion for providing some distractions when stress was high. The VoxelGameDev.com community for providing technical help and staying up to date with the latest developments in volume data structure research. The study association Sticky for providing a comfortable space near the University to study and work. And lastly the Utrecht Skate Parade for keeping me fit throughout it all.

Contents

1 Introduction

This thesis aims to be an introduction to volume rendering data structures as well as introduce two compression steps of volume density data. The focus is not so much on shrinking our geometry, but more so on shrinking the actual density data. This is required for our desire to implement full three-dimensional volume animation models in the in-house renderer at Traverse Research. A basic understanding of the Graphics processing unit (GPU) is expected. Along with knowledge of the basic terminology used when talking about ray tracing such as acceleration structures, primitives, rays, and the rendering equation.

The structure of this thesis is as follows. In Chapter 2 we start with an introduction into rendering, volume rendering specifically and discuss some sparse volume data structures along with general compression techniques. In Chapter 3.1 we go over the requirements set up by Traverse Research which were the main guidelines/constraints for this research. After which Chapter 4 follows where we cover the main techniques behind our implementation, which should convey the main idea behind the researched techniques. The following chapter, Chapter 5, describes how our methods were implemented and used in Traverse's custom engine. This implementation led to multiple tests and benchmarks which are described in Chapter 6. And lastly we conclude this research in Chapter 7 where we answer the research questions and show some final renders made using our methods.

1.1 Traverse Research

Traverse Research is a company that specialized in state-of-the-art research in graphics, and more specifically ray tracing. Traverse does research for multiple hardware vendors including AMD and ARM, for whom they develop ray tracing algorithms, optimizations, and workloads.

1.1.1 Breda

Traverse's current in-house hybrid rendering framework "Breda" includes many features that are expected from a modern renderer. However, volume rendering is still largely unexplored. Breda can compile for either a Vulkan or DirectX 12 backend to make use of the latest ray tracing features. Along with that, some advanced rendering techniques are used. Most notably bindless rendering[Bouma, 2022] and render graphs[Oomen, 2022]. The former is a different way of interfacing with data on the GPU, while the latter simplifies GPU synchronization while optimizing dispatch ordering based on read and write dependencies of different pieces of data on the GPU. All of this is written in Rust and HLSL.

1.1.2 Volumes

In parallel to this research there is another project going on that focuses on shading as explained in Section 2.2. The combination of both projects allows for physically based shading of large animated volumetric effects in real-time. The exact placement of this research inside the Breda framework can be seen in Figure 1.

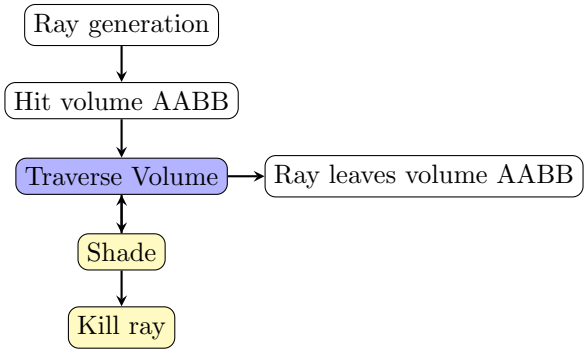


Figure 1: Flow diagram highlighting where this research fits inside the Breda framework. The white boxes indicate parts of the pipeline that already exist inside the Breda framework. The blue box shows what this research will focus on, and the yellow boxes highlight what parts were developed in parallel to this research. As can be seen, when rays bounce inside a volume, they can repeatedly be shaded and continue traversal until they either leave the volume or are killed because they won't contribute any light to the scene.

2 Related work

This chapter goes over all prior knowledge that is required to properly follow the rest of this thesis. We quickly go through some fundamentals of rendering, and quickly move to volume rendering specific topics. We conclude this Chapter with the gaps in current research and where this research fits in.

2.1 Rendering in games

Computers have existed for some time now, and since the start, they have been used to render things on a screen. Whether this is text, graphs, photos, or virtual worlds. When rendering virtual worlds, for example, games or movies, optimized algorithms have to be used to keep up with the growing complexity and increasing fidelity that is expected of new titles. One of these algorithms that has been used in practically all games is rasterization, which quickly projects triangles onto the screen. A different algorithm, with different benefits and drawbacks, is ray tracing, which tries to emulate the physical properties of light as it exists in the real world. Usually, we render scenes that consist of a set of models, for example for every tree, rock or character in the world. These models are made up of triangles which store details such as material, normal and position.

2.1.1 Rasterization

Screen space rendering techniques using the graphics processing unit's (GPU) hardware rasterizer are frequently referred to as rasterization. This technique has been the industry standard for at least the past 20 years. It is a pipeline of operations that calculates which triangles are visible (see Figure 2). Because graphics hardware has been optimized for this specific pipeline, it is fast and thus, where possible, the rasterization pipeline can and should be exploited where possible. However, there are issues. For example, reflections cannot show off-screen objects without sophisticated (and often slow) tricks. This is an inherent issue with the rasterization because the fragment shader stage does not know anything about off-screen and obscured triangles. Along with that, rasterization performance tends to scale linearly with the number of triangles. Therefore, it quickly becomes impractical to use for scenes/assets with incredibly high triangle counts, as required by modern render pipelines.

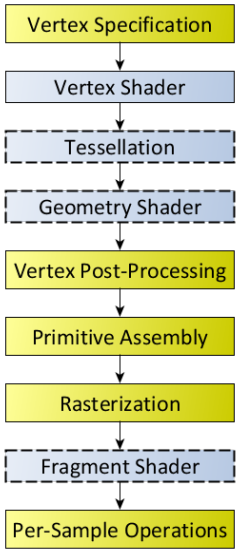


Figure 2: An overview of the rasterization pipeline, where blue stages are programmable and yellow stages are dedicated stages built into GPU hardware. We start with a bunch of vertices and apply a vertex shader. This allows the triangles to be transformed for whatever reason, whether it's because the camera was moved or an object has changed for example. Then we tessellate the vertices into triangles. After which we apply an optional geometry shader that can generate or remove triangles. Then our triangles are projected onto an image, and we end up with information like normal, texture coordinate, and depth. This information can then be used to calculate the final color of a certain set of pixels. [khronos, 2022]

2.1.2 Ray and path tracing

Ray tracing is an entirely different technique that allows us to create arbitrary rays and find what primitives they intersect. Which has the benefit of no longer being bound to a certain view matrix, like with rasterization. This immediately opens up the door for techniques that were traditionally not possible. For example, off-screen reflections can now be rendered by shooting new rays from the reflection point [Whitted, 1979] (see Figure 4). We also don't have to process every triangle individually, instead, we can shoot rays for every pixel in the screen, and find the intersecting triangles quickly by using an acceleration structure.

Although ray tracing has been theorized by many to be the replacement of classic rasterization methods, we have only recently seen it being applied to games after NVIDIA released its RTX graphics cards. Additionally, we see that all AAA games currently still depend on the rasterizer and only use ray tracing to enhance their graphics either by adding realistic (1) diffuse indirect light, which is often mislabeled as global illumination, (2) shadows, (3) reflections, (4) ambient occlusion or a combination of these four[Burnes, 2018].

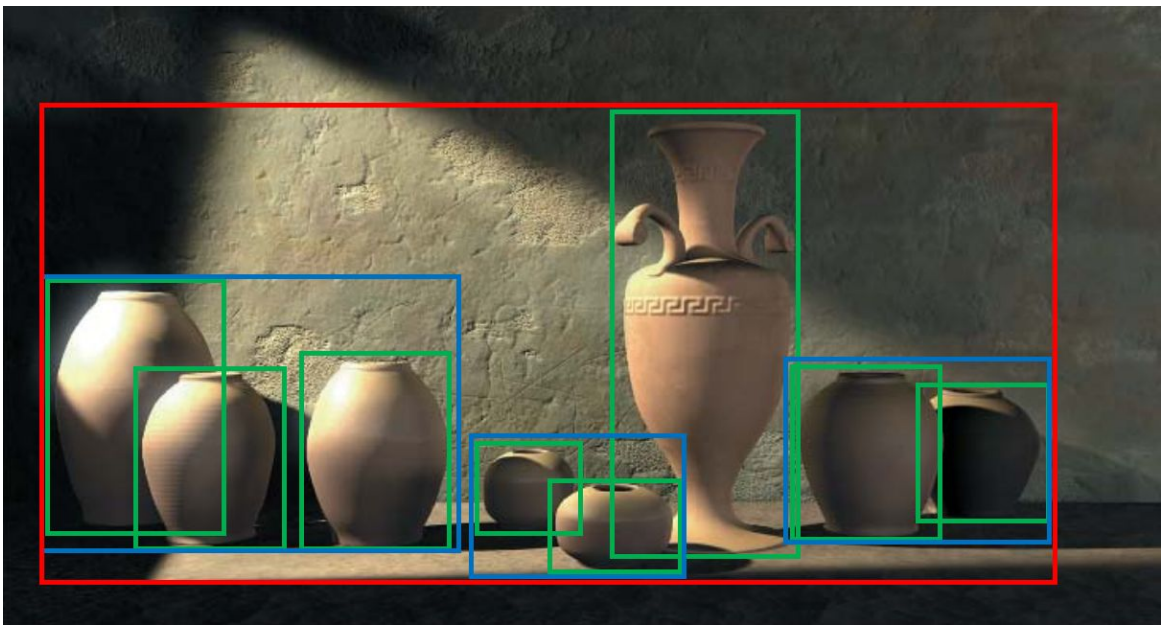


Figure 3: A bounding volume hierarchy where the red bounding box encapsulates all objects. Then a step lower in the hierarchy, we see the blue bounding boxes which encapsulate groups of objects. And the leaves, indicated by the green bounding boxes, contain the primitives. These structures are stored in a tree-like structure and can, theoretically, have infinite depth or width. Finding the intersection between a ray and the bounding volume hierarchy can be done in $O(\log n)$ time on average, where n is the number of primitives. [Bikker, 2022]

Path tracing is a specific algorithm that uses ray tracing to calculate physically accurate images by trying to solve the rendering equation [Kajiya, 1986]. It does this by using recursive numerical integration of the rendering equation for every pixel. This results in a realistic image once converged. However, for the longest time, this has been too slow to run in real-time on commodity hardware. Fortunately, there have been some major advances in (1) sampling techniques as described in [Lin et al., 2022] and (2) filtering techniques as described in [Yang et al., 2020]. Along with the advent of better GPU hardware, it might make real-time path tracing a reality for upcoming AAA games.

Generally, path tracing involves multiple steps to achieve real-time performance [Laine et al., 2013], (1) ray generation, (2) ray extension, (3) shading and (4) shadow ray extension. This pipeline is executed for every pixel on the screen, so for a standard 1080p monitor that would be $1920 * 1080 = 2.073.600$ rays. Step 1 is executed once at the start, and then the latter three steps are done repeatedly to simulate realistic light transport through the scene. The main task of steps 2 and 4 is to find intersections with the scene geometry. This can be done by individually performing intersection tests with every triangle in the scene with a time complexity of $O(n)$, where n is the number of triangles. However, this linear scaling is prohibitive when the complexity of models increases. So to fix this problem, bounding volume hierarchies (BVH as seen in Figure 3) are used to accelerate these intersection tests. This is done by creating a traversable tree with triangles at the leaf nodes, which reduces the average traversal time to $O(\log n)$. To allow instanced rendering without drastically increasing the triangle count of the scene or increasing build times, a top-level acceleration structure (TLAS) is created over a bottom-level acceleration structure (BLAS). Here the TLAS contains pointers to multiple BLAS's which contain the actual model data [Koch et al., 2020].

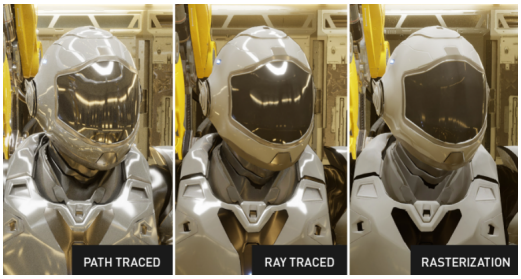


Figure 4: Here we see the difference between a path traced, a ray-traced and a rasterized render. As can be seen, the rasterized scene does not contain accurate reflections and has trouble with specular highlights from the many lights. The ray-traced render does show accurate specular reflections. And when we look at the path traced scene we also see glossy reflections and accurate indirect illumination. [NVIDIA, 2022]

2.2 Path traced volume rendering

In this project volume data represents clouds, explosions, fire and sand/dust storms. All of these effects consist of small particles which we can model using densities. This means that we are not just interested in finding the discretized boundary of a volume, but also in the contents of the volume. We may have varying densities

or even different materials within one volume. For example, a flame that emits light surrounded by smoke, where the outer edge of this smoke is so sparse that it is almost completely transparent. When shading these kinds of materials there are three main things which can happen: (1) The ray is absorbed, meaning that no light transport will be rendered. When this happens, the light energy is converted into another energy form like heat. (2) The ray is traversed through (refracted) or bounces off (reflected) the particle. This will change the direction of the ray and potentially change the ray payload, for example when a blue particle is hit, the ray is modified to only return the blue color if a light is hit. (3) The ray hits an emissive particle. Meaning that light transport will be rendered, and the ray will not continue.

2.2.1 Data structure

Ray tracing these kinds of volumes is different from normal ray tracing, as they require completely different methods. Normally there is a set of surfaces that define a scene. To render these scenes, BVHs are used to find an intersection between a ray and the scene as described in Section 2.1.2. For volume rendering, we don't have these concrete surfaces but instead have volumes with implicit particles, like dust or water droplets from a cloud. To render these volumes we somehow have to find where we hit such a particle and evaluate its shading function. We do not simulate these particles individually but describe volumes of the scene using densities, which are discretized into small volumetric pixels (voxels).

2.2.2 Delta tracking

When traversing such a volume we calculate how far we can traverse through these densities before we are likely to hit a particle [Kajiya and Von Herzen, 1984]. Then we take the step, evaluate the shading function and calculate how long our next step can be. This process is called ray marching [Quilez, 2008]. For homogeneous volumes (with a uniform density) these step sizes will always be the same, but for heterogeneous volumes, they might not. In the heterogeneous case, we calculate the longest possible step we could take through the most dense voxel in our scene and use that to make sure we do not overshoot and miss any dense voxels. This method is called delta tracking [Kutz et al., 2017] and quickly becomes expansive when a single voxel has a high density. A solution would be to take steps proportional to the highest local density, for example, the highest density of the 8^3 nearest voxels. This way we can take step sizes that are proportional to the densities which we are currently traversing, instead of crippling the performance for every ray because of a single voxel.

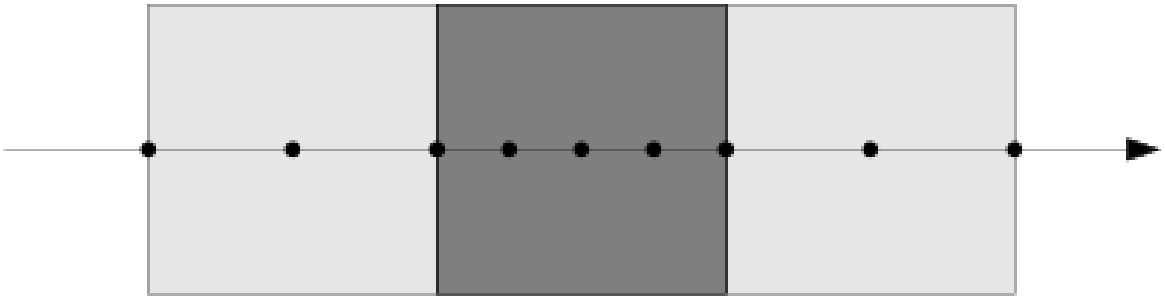


Figure 5: A ray traversing through a heterogeneous volume, where dots represent the location of shading evaluations. As can be seen, the more dense the volume (darker) the shorter the steps are.

2.3 Voxel data structures

Data structures for volume data have seen multiple forms trying to optimize ray marching performance, memory footprint, update speed and simplicity. This research specifically targets heterogeneous volumes which often consist of gradients where outer voxels can have small non-zero values, instead of having opaque volume boundaries. We will briefly go over some major volume data structures, below.

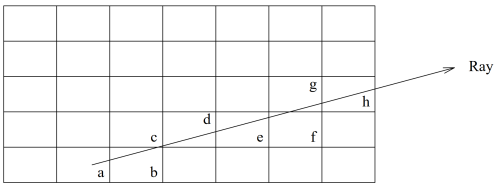


Figure 6: A ray being shot through a dense grid and marking every traversed voxel with a letter. [Amanatides et al., 1987]

2.3.1 Dense 3D grid

The most basic structure (see Figure 6). Here, every value in the 3D array corresponds to a voxel. Indexing into this structure is fast and editing is trivial. However, there is no compression or space skipping. Another problem with flat arrays is their memory requirement as it has cubic scaling with the resolution. However, small-scale simulation software still often relies

on this data structure as it is easy to implement, and in many cases, simulations have to update each voxel anyway.

2.3.2 Efficient sparse voxel octrees (ESVO)

A solution to the memory problems was proposed by NVIDIA, by introducing sparsity to homogeneous volume data [Laine and Karras, 2010]. This is an efficient implementation of an octree for the GPU. Furthermore, this tree only stores data for heterogeneous volumes making emtpy regions small in memory. This memory reduction also yields a speedup when traversing the volume, as larger steps can be taken through homogeneous areas (see Figure 7). Along with that, the structure will more easily fit in the different caches of the GPU. However, due to the nature of tree structures, the data access pattern is less coherent which negatively impacts ray marching performance. The paper also did not take any form of animation into account. So this data structure only really works on static scenes [Lin, 2021].

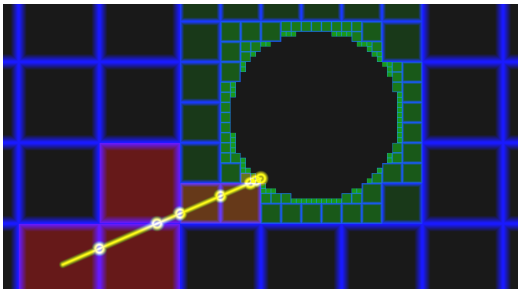
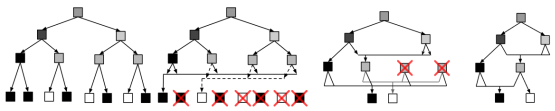


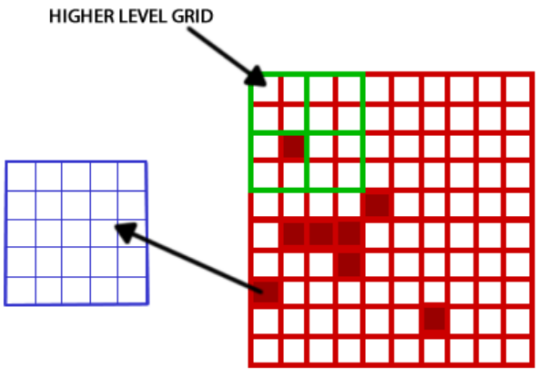
Figure 7: Hierarchical digital differential analyzer (HDDA) traversal through a sparse voxel octree. The yellow line is our ray. The brighter green a cell is, the deeper our nodes in the octree are. We can see that nodes that are further away from the circle, can be stepped through with big steps, but as we approach the circle, our steps become smaller. [abje, 2017]

2.3.3 Sparse voxel directed acyclic graphs

A follow-up paper was released further improving the memory footprint optimization [Kämpe et al., 2013]. Here, the nodes of the octree can be reused for identical volumes, so one leaf can be the child of multiple nodes that share the same data (see Figure 8a). The result is a significant reduction in memory usage over ESVO. Afterward, additional papers were written to, (1) further optimize the memory footprint by creating a lossy data structure which will slightly alter the volume data in an attempt to find more similar volumes [van der Laan et al., 2020] (LSVDAG). And (2) allow for real-time editing of this data structure [Careil et al., 2020] (HashDAG).



(a) Removal of identical nodes in all levels of a binary tree. In the SVDAG paper, the same technique is applied to octrees. [Kämpe et al., 2013]



(b) Here we see the brick map (in red), which is dense. Whenever an area in this brick map is not homogeneous (indicated by the filled-in squares), we point to a brick (in blue). [van Wingerden, 2015]

2.3.4 Brick map

Later another paper was released which tried to maintain part of the memory benefits from octrees while also improving ray marching performance [van Wingerden, 2015]. This was done by keeping the idea of a hierarchical structure, but limiting the tree depth to 2 layers (a top-level grid, the brick map and the bottom-level bricks, as can be seen in Figure 8b), which improves cache utilization. An additional benefit of reducing the depth of a data structure is the update time. When running a fluid simulation, which has to touch every leaf node in a tree, a shallower tree will take fewer steps when going down to the bottom of the tree to modify each voxel. Similar to the flat 3D array, the brick map structure is fast when it comes to update speeds, as both have shallow tree structures. Another paper explores implicit brick maps [Nießner et al., 2013]. These are implicit because they don't encode their location in a grid, but hash a world position and use that hash to index into the brick buffer directly. This allows for an unbounded index space instead of having to create a grid inside a

certain axis-aligned bounding box (AABB). However, nothing has been documented about them in the context of ray tracing.

2.3.5 VDB

In the movie industry, **VDB** [Museth, 2013] has been the standard for volumetric rendering for a long time. This is a hierarchical B+tree structure with 3 layers (see Figure 9), which optimizes for both query speed and memory size. It does so by making clever use of bit masks (see Section 2.4.2), logical bitwise operations and inverted tree traversal. The last of these optimizations means that we don't have to traverse down the tree every time we query a point. But instead use an accessor object, we assume that consecutive points will be close to each other, and remember where we are in the tree by storing pointers to these locations in our accessor. Then the next time we query a point we start at the bottom of the tree and have $O(1)$ access time if we retrieve a point that resides in the same bottom layer.

The VDB structure has mostly been used on the CPU. There have been implementations that port VDB to the GPU, but these do not support dynamic topology [Hoetzlein, 2016] [Museth, 2021]. They would allow individual voxels to be modified, but can't place new voxels in arbitrary locations. This limitation makes simulation on the GPU impractical while maintaining the sparsity which makes the structure effective. The reason why this dynamic topology has never been implemented likely has to do with an inherent race condition when trying to modify any tree structure in parallel.

Research has also been conducted on encoding voxels or even internal layers of the VDB structure in neural networks [Kim et al., 2022]. These neural encodings can store a lot of detail in limited memory, but take minutes to train. Which, again, makes dynamic updates impossible.

One useful result of the standardization of VDB technology is the data format. Many tools can interface directly with the VDB file format [Attrach, 2022], and many artistic tools can export to the VDB file format. These features make it easy to download specific effects and test in a specific renderer.

2.4 Attribute separation

All modern processors are built with cache hierarchies. These caches are usually a few megabytes and consist of many 128-byte cache lines. Which contain all recent memory accesses so, they can be reused quickly when the memory is fetched again. When accessing any memory, an entire cache line (of 128 bytes) is loaded. This means the memory access cost of fetching a single floating point value (4 bytes) is the same as fetching 32 (128/4) floating point values. Abusing this is key to writing fast CPU and GPU code. Therefore, it is crucial to have as little unused data within each cache line fetch as possible. This can be achieved by separating the voxel data from the geometry structure [Dado et al., 2016].

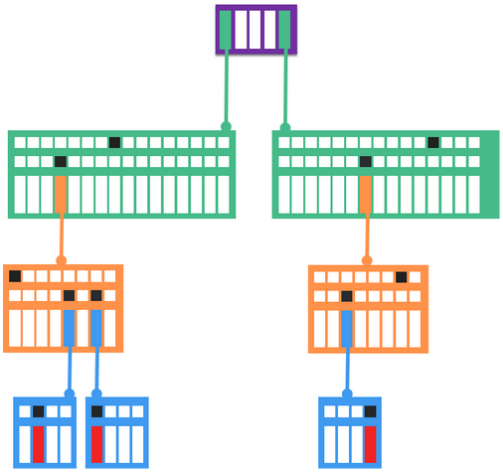


Figure 9: The four-level tree structure of the VDB data structure. In purple, we see the top node which has an arbitrary size and contains a hashmap, indexed using spatial hashing, which points to the first level of internal nodes. The two levels of internal nodes (in green and orange) both contain three rows of values. The top row is a bit mask indicating which values are "active" (this is application-specific), the middle row is a bit mask indicating that a child node exists, and the bottom row contains a pointer to the child node. Then in blue, we have a top row which is a bit mask indicating if a voxel is empty, and the bottom row containing the actual voxel data. [Museth, 2013]

2.4.1 Fast volume traversal

There are two main branches of volume traversal. The first is by sampling the volume along a ray with certain step sizes. This is what delta tracking does (see Section 2.2.2). Another method of traversing the volume is a more exact technique called digital differential analyzer (DDA) [Amanatides et al., 1987]. This analytically calculates how far along the ray we can step to arrive at the next voxel. Using this technique we also know exactly how much density we accumulated along our steps. This has benefits for the shading part of our render, but we won't go into that in this research. DDA has been extended to allow hierarchical volume data structures to be traversed, this is called a hierarchical digital differential analyzer (HDDA)[Laine and Karras, 2010].

Recently, NVIDIA has done research about using signed distance fields (SDF) to speed up traversal[Söderlund et al., 2022]. The general idea behind these distance fields is that the maximum step length a ray could take in any direction (without hitting anything) is pre-calculated, and used to skip over voxels that are known to be empty (see Figure 10). This method could

be applied to most of the data structures described above, and
might specifically be promising when combined with flat structures which don't implicitly store distance values by using bigger nodes in homogeneous volumes. However, calculating these SDFs is one additional step in the pipeline. Although this can be done quickly even for large volumes (less than 1 ms for 1024^3 voxels [Cao et al., 2010]), it is unclear if the speedup during tracing outweighs the construction cost.

2.4.2 Bit masks

When taking steps along a ray, there is a high chance that we will often access voxels that are close to each other, especially when using DDA, and we have to traverse all consecutive voxels. To reduce the bandwidth requirement of these methods we can use bit masks to indicate whether voxels are empty [van Wingerden, 2015][Museth, 2013]. When using 32-bit floating point values as densities, for example, we can traverse all voxels that have a density of 0 using the bit mask. Respectively, we can have a $16 \times 8 \times 8$ region of voxels in a single cache line, making our traversal significantly less bandwidth-heavy.

Method	Sampling Performance	DDA Performance	Memory Footprint	Update Speed	Simplicity
Dense Grid	++	−−	−−	++	++
ESVO	−−	+	+	−−	−
SVDAG	−−	+	++	−−	−−
Brick map	+	++	−	+	+
VDB	+	++	+	−	−

Table 1: An overview of the different data structures. The pluses and minuses provide some intuition into what every structure is optimized for. A very broad summary of the different metrics: The shallower the tree, the faster our sampling. Deeper trees have better DDA performance (see Section 2.4.1 for a very brief description), but we get optimal performance when our nodes approximate the size of our caches and cache lines. Deeper trees improve memory footprint and deduplication improves it even further. The shallower our tree, the faster and easier our updates. And simplicity depends on the method.

2.5 Texture compression

Although it might sound odd, we could learn a thing or two from texture compression techniques. They are similar to our voxel data in the sense that both contain floating point arrays. The only two differences are that textures usually store color data and are two-dimensional, while our volume data contains densities and is three-dimensional. These floating point arrays that textures use have pretty much always been too big for computers, at least when no compression is applied. Usually, photos are compressed using a lossy technique like JPEG, and logos or other digital assets without too many gradients are compressed using PNG which is not lossy. These methods work fine when we care about compression ratios, but they do not when we want to do billions of texture fetches per second on the GPU. For this purpose block compression is usually being used[Reed, 2012]. Block compression has different formats for different use cases. For example, BC4 is suitable for grayscale textures, and BC5 is optimal for normals (as seen in Table 2). These formats all encode 4×4 pixels into a certain number of bytes, which can be used for 2D textures but also technically work in three dimensions. However, we do get spatial dependencies in two out of three axes then, these are the dimensions in which the 4×4 chunk is arranged.

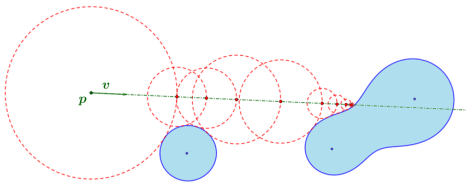


Figure 10: Marching through a signed distance field enables steps that are exactly as long, as the closest heterogeneity is far away. Here we see the ray in green, and the step sizes are indicated by the red circles. [Bálint and Kiglics, 2021]

2.6 Gaps in current research

There are of course still many issues with volume data structures, so below are some of the most notable issues with the current state-of-the-art, and how we can push the state-of-the-art forward in a meaningful way. Many methods have either optimized volume data structures for memory usage [Laine and Karras, 2010][Kämpe et al., 2013], ray tracing performance [van Wingerden, 2015] [Söderlund et al., 2022] [Museth, 2013] or simulation times (flat 3D array). However, none of these techniques currently can do all three of these things while keeping all data and computation on the GPU. Below are two issues that this research tries to address.

Type	Data Rate	Palette Size	Use For
BC1	RGB + optional 1-bit alpha	0.5 byte/px	Color maps Cutout color maps (1-bit alpha) Normal maps, if memory is tight
BC2	RGB + 4-bit alpha	1 byte/px	n/a
BC3	RGBA	1 byte/px	Color maps with full alpha Packing color and mono maps together
BC4	Grayscale	0.5 byte/px	Height maps Gloss maps Font atlases Any grayscale image
BC5	2 × grayscale	1 byte/px	Tangent-space normal maps
BC6	RGB, floating-point	1 byte/px	HDR images
BC7	RGB or RGBA	1 byte/px	High-quality color maps Color maps with full alpha

Table 2: Block compression formats and use cases [Reed, 2012].

2.6.1 Large assets

Video game assets have been growing in size to the point where games can be hundreds of gigabytes. Adding another asset type, volumes would further increase this. So although there has been a lot of research done in compressing geometry (the volume shape without its per voxel attributes) to the point where entire scenes can be voxelized into game-ready asset sizes[van Wingerden, 2015][Museth, 2013]. We also need our actual density data to be small, which has been done by [Dado et al., 2016], but this was applied to normals and colors, not specifically densities which only have one floating point value per voxel instead of three.

2.6.2 Animation delta compression

Another thing that has not been explored extensively is applying delta compression on our volumes. This has been hinted at by [Careil et al., 2020], where they suggest encoding animation frames as different branches of the tree. Resulting in a structure that can simply swap some pointers at certain points in the tree when changing to the next frame. However, there are two issues with this method: (1) The deep oct-tree structure is not optimized for tracing, and when flattening the structure we rapidly reduce the effectiveness of deduplicating our nodes. (2) This does not change the attributes of the lower-level nodes, so when our attributes change (a density increases or decreases) we would have to change all parent nodes as these should no longer point to old data. Effectively requiring a new tree for every animation frame.

3 Research questions

In this chapter we list the requirements set by Traverse Research. After which we formulate two main questions that we want answered.

3.1 Requirements

This research focuses on a specific set of requirements having to do with volume data structures. We want to support volumetric animations in the Breda renderer developed by Traverse Research (as explained in Section 1.1). This means that we want to support path tracing volumetric effects, with optional emission using assets that are proportional to standard game asset sizes. Below are the most important requirements our data structure should adhere to.

3.1.1 Asset size

Asset sizes in games have been growing for a while, with the latest titles shipping 4k textures. To keep our assets in line with standard game assets we should not be going above 100 MB per volume animation. These sizes of course heavily depend on the number of animation frames and the resolution of the volume, but sized much larger than 100's of MB become detramental to artists work flow in the engine. This requires both the volume geometry and attributes to be significantly compressed.

3.1.2 Sampling speed

Our sampling access times should be fast enough to enable path-traced volumetrics on high-end current-generation hardware. This means that our algorithm should be optimized for a typical path tracing access pattern.

3.1.3 Animation playback

Animation playback is a mandatory feature for our renderer. And if we are running high frame rate animations these should not take up too much time of our frame budget. So complex expansive delta update schemes are not an option for us.

3.1.4 Lossy compression

Pretty much all effective compression schemes for floating point data are lossy in some form. The difficult task is making this lossiness as hard as possible to notice. So when we compress our volume data it is essential that we keep its high and low frequency features intact.

3.1.5 Level of detail

For certain parts of our rendering process, we need to use the absolute highest level of detail, for example for our primary rays. However, for our secondary or shadow rays, we do not have to care as much about the exact volume boundaries. This will allow us to use a lower level of detail model to achieve almost the same results, which in turn allows us to improve performance.

3.2 Research question: Optimal data structure

What data structures, or combination of structures, are optimal for ray tracing, memory, simulation and animation, and can these structures be converted into each other? We have already seen that different techniques are optimized for different metrics. So if we know the optimal method for each of our requirements, we can work towards combining their strengths without having to create one structure that can do it all (which evidently does not exist, yet).

3.3 Research question: Performance bottlenecks

Is volume traversal memory or compute bound? This question should provide insight into future optimization techniques. Insight into questions like "If L1 cache size is increased by X amount, will this improve volume traversal speed?" or "If the clock speed of a GPU is increased, what will the impact on our volume traversal be?" will be acquired.

4 Approach

In this Chapter, we describe how we went about the creation of our data structure and what the different considerations were. This includes the high-level ideas that resulted in our data structure. We also include a section explaining the required steps for running simulations inside the VDB data structure, unfortunately, this was not implemented. We then continue with the animation setup and the compression scheme used for our voxel data.

4.1 VDB data structure

The data structure of choice is VDB. This is a compromise between ray tracing performance and compression, and can be extended or reshaped (internal layers could be removed, added or changed in size) if desired. However, the top-level node, which can be indefinitely large, was not used. This removes the need to perform the expansive hash table step and also removes the need for our accessor types. Additionally, this restricts our index space to a certain region. In our implementation, we chose the standard 5-4-3 VDB layout resulting in an index space of $1 << (5 + 4 + 3) = 4096$ cubed. More than large enough for most VDB files. This has a few implications: an empty volume uses $((1 << 5)^3 + ((1 << 5 + 4)^3))/8 = 16$MB because we always allocate 1 top-level node along with one 2nd level node for every bit in the top level node's bit mask. There are also always 3 indirections before we can access our voxel data. Our large higher-level nodes will rarely contain duplicates (as required by SVDAG, described in Section 2.3.3).

4.2 Simulation

As part of implementing a dynamic VDB data structure, there was some work done regarding a GPU-modifiable version. Unfortunately, due to time constraints, this was never fully implemented or tested. However, there were some findings about how such a system could be implemented.

Modifying a sparse volume data structure on the GPU has many inherent issues. One of those issues arises because the GPU is a massively parallel system that requires cautious handling of synchronization. When implementing a VDB-like structure with 3 indirections (top-level node, 2 internal layers, and the voxel data) we have to make sure that all parent nodes are present. One way of doing this would be to check if the parent exists, if not, we create it. We can create the parent node by keeping track of an atomic counter which is used for indexing into a large buffer in which we allocate our parent. However, what if we allocate two voxels with the same parent, then we will allocate twice and have the parents parent point to one of the allocated nodes. This is a race condition.

Let's say we tried to use that scheme, but we use one of the essential parts of the VDB data structure, the bit mask. So we use an atomic bitwise or operation on the bit mask which sets the correct bit to true. This operation will also return the original output, which for one thread will be false, and for all other threads that did the atomic operation, will be true. So now we have a single thread that can allocate the node and set the parent's parent pointer right? Well no, the GPU can, at any time, swap out the currently running warp. So one warp can set the bit, and be responsible for setting the parent's parent pointer, but it is being swapped out. This would result in other threads assuming that the pointer is already set, and them accessing garbage data.

So the only real way to handle this situation is to set the bits and pointers without reading either of these values within the same dispatch. This necessitates the separation of the initialization of the different layers of the tree into different dispatches. However, this does raise a new problem, how do we know which top-level nodes to allocate and which bits to set? We could iterate over our entire dataset for every dispatch to find which nodes have to be allocated, but this might be wasteful if our dataset is large.

To conclude, if an editable VDB data structure is desired, it can be done. However, it will necessitate the use of at least 3 dispatches which might all need to access the entire dataset (as show in Figure 11). If the dataset is not too big this might be fine because of the GPU's high bandwidth.

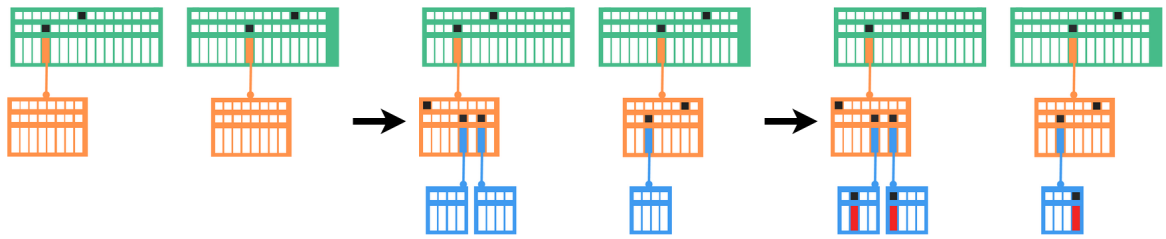


Figure 11: Construction of the VDB data structure in 3 steps. These steps are separated by arrows which indicate the different dispatches with barriers between them. This ensures that all work in the upper level of the tree has been done before continuing to the lower levels. The first step is to build set our top level bit masks and pointers. Then, in the next dispatch, we set our internal node bit masks and pointers. And lastly we set our lowest level node pointers and bit masks. After this dispatch we can start placing our voxel values.

4.3 Flip book animations

Flip book animations are the opposite of delta compression. It means that we store every frame as a separate data structure, disregarding any temporal coherency. So why mention them, when they are not what we need? If we look at the actual data being used we see that by far the largest amount of memory is being used by the actual voxel data. Depending on what tree structure we use, this can be a larger or smaller part of our structure. This leads to another interesting observation. If all these data structures use voxels as their leaf values, they could all share these leaf values. We can build different structures on top of a set of voxel bricks. For example, we can have a set of bricks in a brick map to run a simulation, then mesh these bricks, so we can use the hardware rasterizer to find primary intersections. After which we can traverse them using DDA (see Section 2.4.1) and do our shading. All these operations utilize the same underlying voxel data. So if we can get that as small as possible, we should be able to use an appropriate 'interface' to our data and have exactly what we need. Whether this is a flat or deep tree or even something like a triangle mesh or BVH. This is why we mostly look at compressing our voxel brick data as much as possible. For animations, we have thus decided to use a simple flip book animation for the tree, while using one large voxel brick buffer.

4.4 Texture compression

Keeping our voxel brick data in a large buffer can only do so much on the GPU. We are bound by 32-bit loads, have to set up a cache-friendly access structure ourselves, and most importantly, cannot use block compression (this only works on textures as it uses spatial coherence to compress). This is why the voxel data is stored in a sampled 3D texture. Using this texture we automatically get optimized access to our voxel data in the same way that 2D textures have fast access for filtering. The actual bricks should now be stored in texture slices. So, we can create a texture with a certain width, height and depth (divisible by 8, our brick size), then move each of our voxels from the old brick buffer to our new brick texture.

To compress our data there are a few new possibilities. Because we can skip these 32-bit loads, we can easily store 16-bit floating point or unsigned normalized values (8-bit floating point values between 0 and 1). These normalized values can be denormalized to get the original values back. So now we have an easy way of reducing our voxel data by $4\times$. But we can go even further. BC4 (See Section 2.5) is a block compression method made for grayscale values, perfect for our density values. It encodes grayscale values at 0.5 byte/pixel, meaning that we can store two voxels per byte. This already shrinks our data from 32bit per voxel to 4bits, an eight times reduction. However, we can use one last optimization using block compression. If we recall how block compression works, we see that it somehow compresses 4×4 pixels. Now if we use one of the block compression techniques which encodes full RGBA data we get four floating point numbers per pixel, resulting in $4 \times 4 \times 4$ values. We can exploit this by storing our voxel data as separate color channels in our texture. When using either BC7 or BC3 we can encode floating point values at 1 byte/pixel. However, each of our pixels contains four density values, thus shrinking our volume data even further to a rate of 2bits per voxel. This is a sixteen-times reduction over using full 32-bit floating point values. A slice of the resulting texture can be seen in Figure 13.

4.5 Clustering similar nodes

The fundamental idea behind compression is the reduction of duplicates. But when using volume data, or any floating point data, there is a low chance of having exact duplicate data. When looking at large-volume datasets however, there is a decent chance that two bricks will have roughly the same data. This can be the almost homogeneous area inside a cloud or a non-moving part of an animation for example. We can identify these similar bricks using clustering. This idea was inspired by [van der Laan et al., 2020] where they performed

clustering on the voxel geometry. However, unlike that technique, we specifically want our density data to be compressed. So we perform k-means clustering [MacQueen et al., 1967] using the normalized Manhattan distance, between 5 and 15 iterations and k is the desired number of new bricks. However, there are two issues with this which can be worked around. (1) low-density bricks will always be similar, and (2) bricks that are similar for almost all density values except for a few, result in high similarity, yet we lose a lot of important detail. Both of these issues can be negated by introducing a variance threshold. We normalize all density values inside a brick, and then calculate the variance of all values in the brick. If this is above some set threshold we simply do not include it in our clustering, and keep using the original value. This way we do lose the ability to cluster smooth gradients, but we do make sure that we keep as much detail as we want. DBSCAN [Ester et al., 1996] was also considered, but it did not seem like a good fit as we specifically only want to cluster bricks that are similar to each other. We do not want DBSCAN to form large chains where two end-points of the cluster are vastly different.

5 Implementation

Below are the different components that have been implemented to support this research. The entire implementation exists inside the Breda repository and is written in Rust and HLSL, this is why all code samples also use these languages.

5.1 Asset pipeline

The Breda asset pipeline is used to load all VDB files. This allows us to specify how our assets are loaded and processed, along with caching our final assets to reduce the amount of rebuilding. The main tweakable options for our VDB asset processing pipeline are the format, grid, filenames and clustering options. The format can be any of 32bit floating point, 16bit floating point, unsigned normalized (8bit float between 0 and 1) and BC7 2.5. The grid specifies which grid in the VDB file we use as our density data. Usually, there are multiple grids in each VDB file, we can have density, temperature and volume type for example. All of these grids are used for different parts in a rendering pipeline, but for this research, we are only interested in a single grid and that is the density grid. The filenames option is used to select one or multiple VDB files, when entering multiple filenames we assume that these are an animation sequence. The clustering options consist of three things: (1) the number of clusters, which is k when running k-means. (2) The number of iterations of k-means. And (3) the variance rejection threshold, which is used to specify how heterogeneous our bricks can be before we stop clustering them.

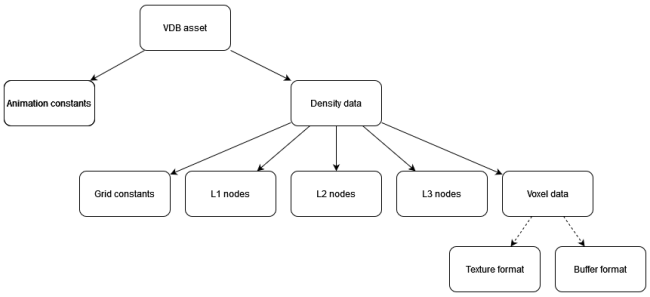


Figure 12: VDB asset data structure. Each VDB asset has constants for the entire animation, and density data. This density data has all the constants, nodes and voxel data. This voxel data can either be in buffer or texture layout.

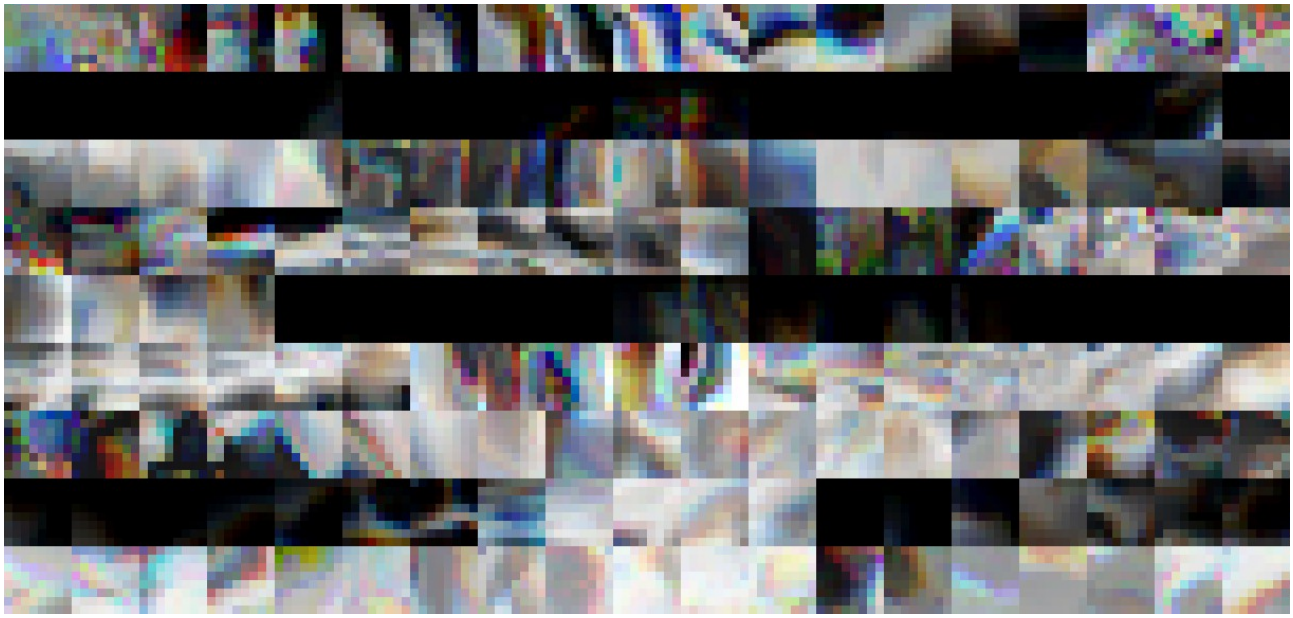


Figure 13: Voxel data texture captured by the Nsight frame debugger. Tiles of 8×8 pixels can be seen. The third dimension, to get our brick data, is encoded using the RGBA color channels and the next texture slice. The tiles often contain gray values, meaning that the voxels in the z-axis are roughly the same (a pixel with roughly similar color values will always be a grayscale). However, some pixels are more colorful, which means that not all color channels are the same and there is either one voxel in the z-axis which is significantly different, or there is a gradient. This slice of the voxel brick data also showcases that even on this small scale of the individual brick level, there are many high-frequency details.

Using these options we can compute our VDB asset (The entire pipeline can be seen in Figure 14). This consists mostly of constants and GPU-friendly vectors of data (see Figure 12). All nodes are implemented as structs which contain an unsigned integer to point to the next level of nodes and an array that contains unsigned

integers. These are interpreted as the active child bit masks. In code, we refer to the internal nodes as L1, L2 and L3 nodes. L1 being the top level node with size $(1 << 5)^3$, L2 having size $(1 << 4)^3$ and L3 having size $(1 << 3)^3$ and pointing into the voxel data. The different animation frames are all unique L1 nodes. So if we want to render a specific frame we can simply take the frame ID and use that to index into the L1 nodes buffer, all pointers after that are handled at the asset creation stage. The full VDB asset creation process is listed below and with an overview in figure 14.

1. First we read the metadata of our VDB files, which we then use to select which grid we want to read.
2. Now we load the VDB data using vdb-rs [Traverse Research, 2023] which was as part of this research.
3. Then we transform this data into our flat GPU-friendly data.
4. The first pre-processing step consists of our deduplication scheme described in Section 4.5.
5. After this we transform our data layout from our easy (and locally coherent) buffer layout to the texture layout.
6. Then, depending on which format options were used, we compress our data. So we reduce our floating point data to either 16bit, 8bit or Bc7 texture data.
7. After which we upload everything to the GPU.

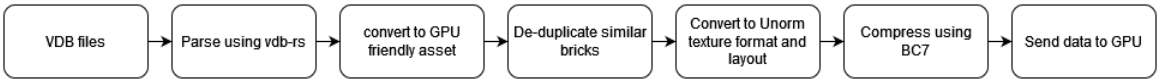


Figure 14: An overview of the asset creation pipeline.

5.2 Shaders

To integrate the built data structure into Breda, two shaders have been written, *VDB.hlsl* and *HDDA.hlsl*. The main way to interface with the data structure is contained in *VDB.hlsl* which has functionality for getting voxels, getting the activity status of certain voxels, and finding the deepest existing child node so we can efficiently step to the correct node boundary (recall that the different node layers have different sizes). *HDDA.hlsl* is largely the same as implemented by [Museth, 2013] but without the activity mask. The main difference is that we make use of callbacks on the GPU, which is done by passing a template argument to the traversal function. This template argument must implement *empty* and *operator()* which are called after every step, or compiled away if either of them is not used. These step functions are used to write clean yet fast traversal code (as seen in Figure 15).

5.3 Compression

As said in Sections 4.4 and 4.5, there are two types of compression of the voxel data going on. First, we will have a look at how to cluster similar bricks. This is largely done by using NDarray [rust ndarray, 2023] for different linear algebra methods. The clustering process is divided up into the following steps:

1. We parse our voxel data into a 2D matrix where every brick is one row, allowing us to use per-row clustering algorithms.
2. Then we calculate the variance of the density data of each brick and divide that by the brick's mean density to get a normalized variance.
3. We reject the brick if the normalized variance is higher than our set threshold. This way we only keep bricks that have low variance and thus have a low likelihood of containing unique features (see Section 4.5 for the reasoning behind this step).
4. After which we run k-means on all low variance bricks, with a given k and the number of iterations. Unfortunately, this can be an expansive step as k-means has an algorithmic complexity of $O(nk)$ where n is the number of bricks in this case, and k is the number of requested new bricks. So when both are large this step will become unpractical. However, most of the low variance bricks can be represented by a few new bricks, so k should not be large anyway.
5. After we have a new set of bricks we just have to fix the pointers of our L3 nodes and merge the filtered high variance bricks with the new bricks, and we are done.
6. In figure 20 we see the results of this compression technique.

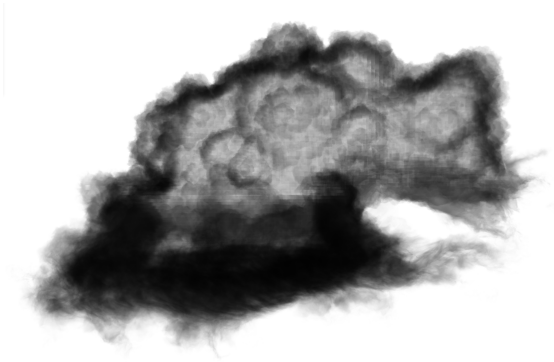
```
// The hdda march method has the following function signature
// We explicitly say which level of our tree we want to traverse
template <typename VoxelData, typename VoxelFn>
void march(inout VoxelFn fn, Layer callbackLayer);

// We define a struct that contains our traversal payload
struct FirstDensityFn {
    float4 color;

    bool operator()(float voxelData, float t0, float t1, uint dataIndex) {
        float chunkIndex = voxelData * 360;
        color = float4(hsvToRgb(chunkIndex, 1, 1), 1);
        return false; // we do not want to continue traversal
    }
    bool empty(float t0, float t1, uint dataIndex) {
        return true; // we continue traversal
    }
};

// Now we can simply perform our HDDA traversal as follows
float4 traverseVolume(){
    Hdda hdda = Hdda::new_(volumeGrid, ray, inverseRay);
    FirstDensityFn fn;
    fn.color = float4(0,0,0,0);
    hdda.march<float>(fn, LayerVoxels);
    return fn.color;
}
```

Figure 15: Simplified outline of the HDDA api.

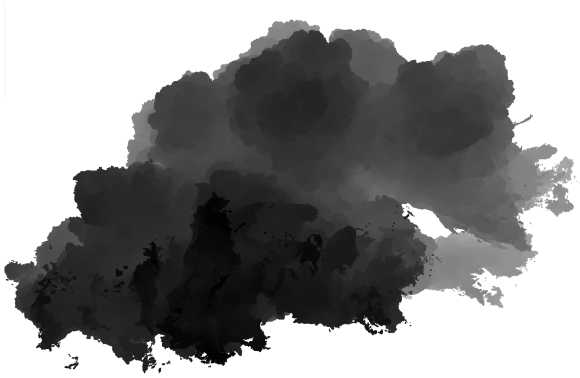
Texture block compression is done after rearranging the data from a buffer layout to a texture layout. With this data layout we can reduce the floating point precision, or even use the existing intel texture compressor [Intel, 2023] to encode our data into Bc7. After doing this we still have to tell our backend what the format is of our texture, and then we are done. We can reuse our HLSL code for all formats because HLSL automatically samples different floating point types correctly.

5.4 VDB viewer

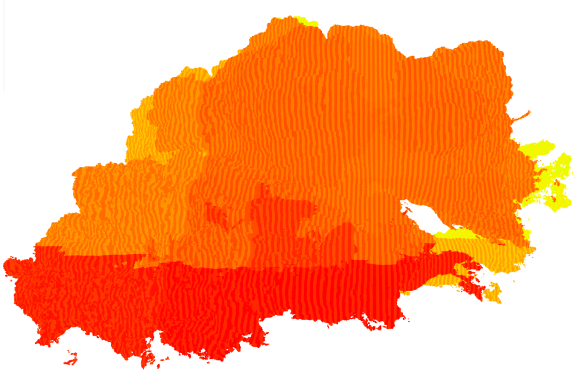
A prototyping application was created for visualizing VDB files. This application can be used to load a single VDB file and display multiple debug views of the model as can be seen in Figure 16. This application also allows quick testing of compression schemes as it bypasses the standard asset pipeline system that was discussed in Section 5.1.



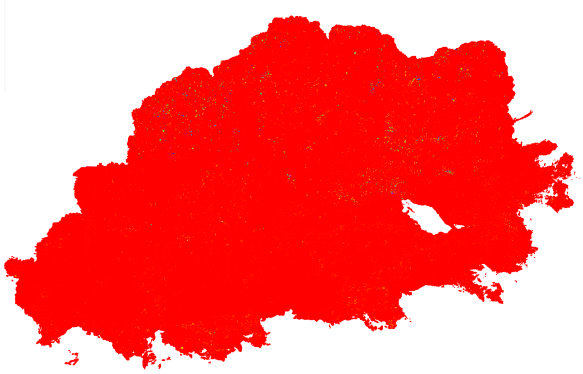
(a) Shaded by accumulating density along the primary ray. The body of the cloud is hollow as can be seen.



(b) Shaded by depth of first hit voxel.



(c) Shaded by index into the voxel index. Shown is a conversion from index to hue using the following formula: $x + y \times dim\_size + z \times dim\_size^2$. The striped pattern is a result of bricks (8^3 voxels) are a combination of two texture slices, these are the altering index colors. The larger shifts in color, which are in specific cubic regions, correlate to the different L2 nodes.



(d) Shaded by the density of the first hit voxel, where a low density is red and a high density results in different hues. Overall the entire outside edge of the cloud only shows low-density voxels, but there are some spots where deeper voxels are visible which have a higher density.

Figure 16: Some shading options of the VDB viewer application. The exact color values do not matter in this example, these figures exist to illustrate general patterns. All renders are done using the half-resolution version of the Disney Cloud [Disney Animation, 2018].

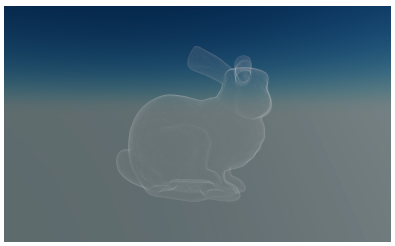
6 Results

In this Chapter, we will first describe the different models that are used for comparison, but not all models are used for every metric. The first metric we look at is geometry size as this was not the highest priority. Then we have a look at the effectiveness of block and clustering compression methods. These results all relate to requirements Asset size and Lossy compression. Lastly we cover the ray tracing performance which deals with Sampling speed. The two requirements that will not have results attached to them are Level of detail and Animation playback. The former was not implemented but can be added, and the latter is practically free as we use flip book animations where we only have to update a single index to switch between animation frames.

6.1 Models

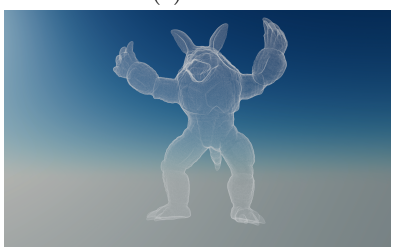
Multiple models were used to evaluate the implemented methods in Chapter 5. The model with the largest scale is the Disney Cloud [Disney Animation, 2018]. Our implementation is bound by volumes with size 4096^3, so we use the half-resolution version. We also use multiple models from the OpenVDB website and the JangaFX website. A mix of models was used with diverse axis sizes, number of voxels, animation frame numbers and shapes. Most models are static and thus consist of only a single animation frame, but to get a rough idea of the scaling of our structure size with the number of animation frames, we specifically used models with 10, 50 and 100 frames as well.

(a) Fire



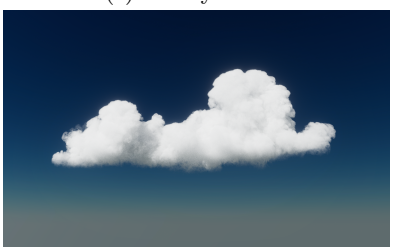
(b) Bunny

(c) Bunny cloud



(d) Armadillo

(e) Dragon



(f) Cloud pack

(g) Disney Cloud

(h) Chimney

(i) Shockwave

Model	Voxels	Dimensions	Animation frames
Fire	4,458,790	[160, 363, 152]	1
Bunny	5,513,993	[627, 620, 488]	1
Bunny cloud	19,210,271	[576, 571, 437]	1
Armadillo	22,734,512	[1,275, 1,518, 1,159]	1
Dragon	23,347,893	[2,022, 910, 1,346]	1
Cloud pack	49,869,596	[349, 178, 530]	10
Disney Cloud	188,358,293	[993, 675, 1,224]	1
Chimney	239,747,485	[160, 350, 505]	100
Shockwave	631,494,157	[732, 140, 729]	50

(j) VDB data

Figure 17: The used volume models for our results. Bunny, armadillo and dragon consist of only the bounding edge of the volume. Cloud pack, chimney and shockwave contain multiple animation frames and thus can make use of delta compression. The voxels column indicates the number of non-zero voxels in the base model. All renders are done using the Breda framework using 100 samples per pixel.

6.2 Tree size

In this section we will provide the results of the latter. As described in Section 4.3, we put our focus on compressing the voxel data instead of our tree structure size. Because of this, we did not implement any form of compression of the geometry (besides the sparse nature that already existed in VDB). This results in long animation sequences having large trees. In Table 3 we can see that the larger models and animation sequences are larger than the in Section 3.1.1 mentioned sizes. This needs to somehow be resolved, either by making our tree deeper or deduplicating bit masks somehow. The latter would be an interesting option. We could make every node in our tree consist of two indices, one child pointer and a bit mask pointer. On one hand, this would add an extra indirection to every level in our tree, but on the other, we will have many internal bit masks where every bit is true. All these nodes will thus point to the same underlying bit mask and pretty much always use cached data. The potential gains here are significant for our L3 nodes, even when all unique bit mask configurations are present. The maximum number of unique L3 nodes is $((1 << 3)^3)^2 = 262,144$ which is quite a lot fewer than the number of L3 nodes present in all but one of our models. In the best case, our shockwave model would reduce the size of our L3 nodes by 7×, and again, possibly improve performance. In conclusion, the tree sizes for long animation sequences are not very practical currently and are the most notable target for

follow-up optimizations.

Model	L1 Nodes	L1 size	L2 Nodes	L2 size	L3 Nodes	L3 size
Fire	1	4 KB	32,768	17 MB	65,536	4 MB
Bunny	1	4 KB	32,768	17 MB	307,200	21 MB
Bunny cloud	1	4 KB	32,768	17 MB	303,104	20 MB
Armadillo	1	4 KB	32,768	17 MB	1,339,392	91 MB
Dragon	1	4 KB	32,768	17 MB	1,302,528	88 MB
Cloud pack	10	41 KB	327,680	168 MB	905,216	61 MB
Disney Cloud	1	4 KB	32,768	17 MB	1,007,616	69 MB
Chimney	100	410 KB	3,276,800	1,678 MB	9,195,520	625 MB
Shockwave	50	205 KB	1,638,400	838 MB	9,658,368	657 MB

Table 3: Tree sizes of the different VDB models. Each node takes exactly one 32-bit index and their bit mask in size. Which makes each L1 node 4100 bytes, each L2 node 516 bytes, and each L3 node 68 bytes. The number of L1 and L2 nodes is directly correlated with the number of animation frames as each frame has a single L1 node, and each of the children of that L1 node is allocated upfront. This makes neither the L1 nor the L2 layer sparse, but the L3 layer is sparse and thus can be smaller than the L2 layer. It's also notable how the trees do not make use of delta compression which adds a fixed cost to every animation frame, making long animations like Chimney very large in memory.

6.3 Block compression

Block compression theoretically has a good compression ratio, by reducing our voxel data from 32 or 16 bits per voxel to 2. There are two main drawbacks to this method. Those being, (1) the reduced precision arising from the discretization into a single byte. And (2) the spatial dependency of voxel data, meaning that voxels within one brick can become inaccurate if there are outliers. We can single out the first issue by using unsigned normalized bytes (Unorm), interpolated between the minimum and maximum value of the volume. This gives us the same discretization as block compression but without the spatial dependency. For good measure, we also compare against a 16-bit float per voxel version. Most VDBs are already using 16-bit floating point data which will thus not result in any difference between it and the 32-bit version. In Figure 18 we see the difference between the different data formats.

Table 18g shows us that all models contain a non-zero RMSE even when using the 16-bit floating point precision (although the precision difference should not be noticeable). This is due to our path tracer being non-deterministic and thus always introducing noise in all kinds of ways. However, this does not have to be an issue when we want to evaluate the accuracy of our lossy compression. Visually there is no difference for any of the models between the 32 and 16-bit formats so, we can take that as a baseline. When we have a look at the bunny and dragon models, we barely see a difference in RMSE when using different formats (the visual difference is rendered in Figure 18c and Figure 18f). Thus, these models work great with this compression technique. When looking at Disney Cloud we see a spike when going from Unorm to BC7. This is visualized above in Figure 18a (the Unorm diff) and Figure 18b (the BC7 diff). Some voxels contain different values compared to their 32-bit counterpart. This likely has to do with quickly varying densities. When having a look at fire we also see a jump in error but this time when going from f16 to Unorm. When having a look at Figure 18d (the f16 diff) and Figure 18e (the Unorm diff), we see that the problem lies in the top half of the model. Specifically, where the densities are very low. This can be explained by the Unorm discretization not properly capturing the subtle differences in different low-density voxels.

This compression method offers very good compression rates with barely any quality loss in most cases. Only when the model contains many values that are not properly representable, like in Figure 18a, do we run into significant quality loss.

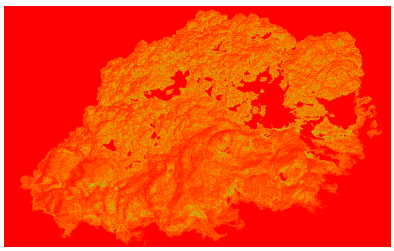
6.4 Clustering

Clustering similar bricks has proven more difficult than simply running k-means on all bricks. The main issue lies in figuring out which bricks contain high-frequency data, and which contain low-frequency data. The former should not be clustered as there is a very low chance that there will be any similar bricks, and thus, reduce the quality if we merge them. In Figure 19 we see what parameters contribute to a minimized loss of quality with the best compression ratio. As also can be seen in the previously mentioned figure, when bricks with high variance are also being clustered we quickly get blocky volumes, this is not acceptable in Breda. Only in sub-Figure 19d we no longer see blocky regions. A more low-level view of our data can be seen in Figure 20. Here we see that many similar clusters have been removed and our final voxel data texture contains fewer

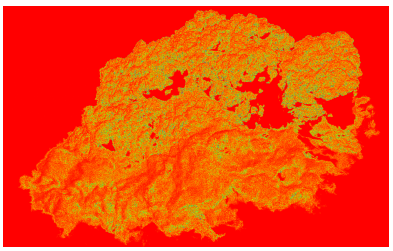
homogeneous grayscale bricks. In general, this deduplication method works very well for larger models where many bricks have the potential to be similar. But when models are already small, say less than 50 MB, there is almost nothing to be gained.

6.5 Ray tracing

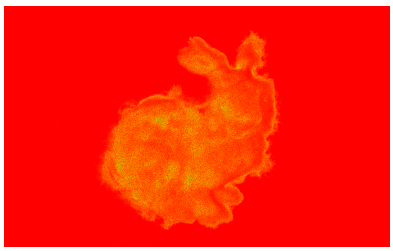
We compare our implementation against the results presented by [Braithwaite and Museth, 2020]. They use an NVIDIA RTX 8000 with 672 GB/s bandwidth and 16.31 Tera floating point operations (TFLOPS), and we use an NVIDIA RTX 3070 TI with 608 GB/s and 21.75 TFLOPS. These numbers do not necessarily indicate performance but do sketch a rough idea of the performance scale. In Figure 21 we show the performance of our implemented technique along with streaming multiprocessor (SM) occupancy and L1 pressure. On the GPU, memory bandwidth usually is an indicator for a shader being memory bandwith bound. Which can be solved by using latency hiding by increasing the SM occupancy. However, in our tests we did not run into L1 pressure being the limiting factor. This tells us that we are not memory bound, but compute bound.



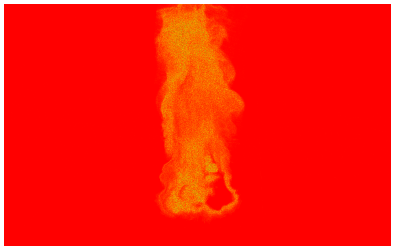
(a) Disney Cloud RMSE between 32-bit float and Unorm formats. We only observe inherent path tracing noise.



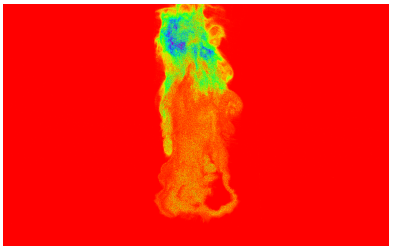
(b) Disney Cloud RMSE between 32-bit float and BC7 formats. There is an increase in noise across almost the entire volume, but not in large regions like in Figure 18e. This tells us that certain voxels in a brick can be wrong, but most of the voxels in each brick are still correct.



(c) Bunny cloud RMSE between 32-bit float and BC7 formats. We only observe inherent path tracing noise.



(d) Fire RMSE between 32-bit float and f16. We only observe inherent path tracing noise.



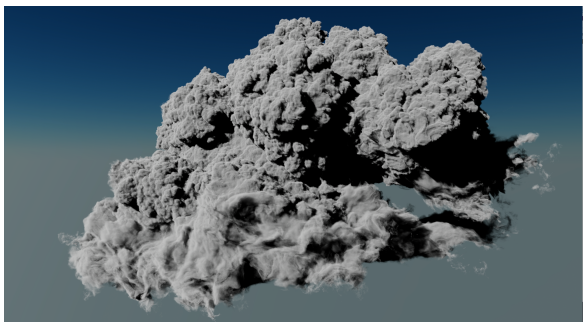
(e) Fire RMSE between 32-bit float and Unorm formats. We observe noise in the low-density regions at the top of the flame

(f) Dragon RMSE between 32-bit float and BC7 formats. We only observe inherent path tracing noise.

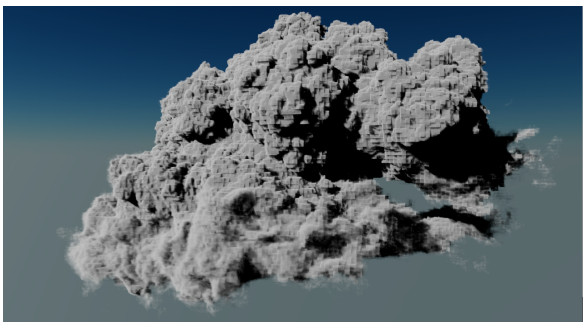
Model	**Data Format**	**RMSE**
Bunny	f16	0.0048
	Unorm	0.0050
	bc7	0.0050
Disney Cloud	f16	0.0089
	Unorm	0.0089
	bc7	0.0135
Dragon	f16	0.0062
	Unorm	0.0062
	bc7	0.0064
Fire	f16	0.0044
	Unorm	0.0099
	bc7	0.0100

(g) Table with the RMSE between 32-bit floating point per voxel, and other data formats.

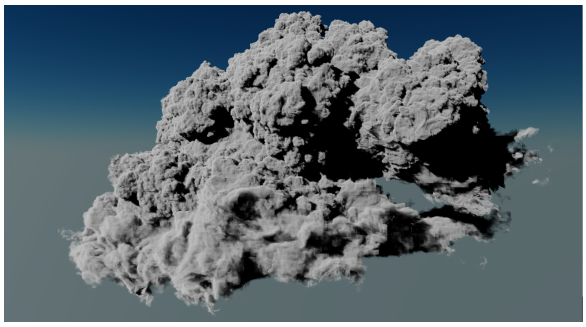
Figure 18: The Root Mean Square Error (RMSE) of the compression of four different models at 1000 samples per pixel with 32-bit floats per voxel used as a baseline. Red pixels have a low error and higher hue values (more green/blue) indicate a larger error. Higher samples per pixel counts reduce the inherent path tracing noise which changes the RMSE offset seen in the comparison with f16 (this format is practically identical to f32), however, converging properly with tens of thousands of samples was impractical. We calculate the difference against the full 32-bit floating point precision model as this is closest to the ground truth.



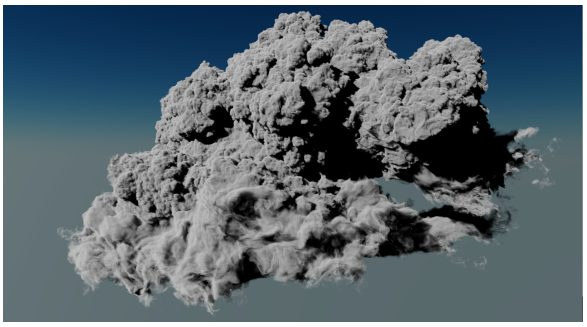
(a) Disney Cloud without cluster compression.



(b) Disney Cloud with all bricks being clustered.



(c) Disney Cloud with a variance reduction of 0.1, clustering roughly 70% of the bricks into 100 new bricks.



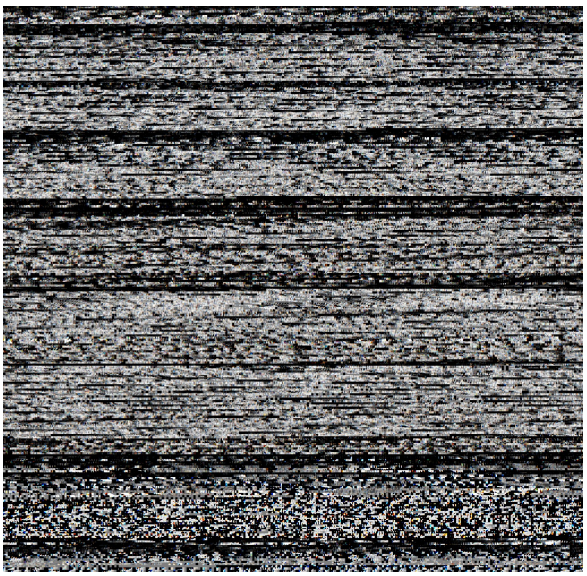
(d) Disney Cloud with a variance reduction of 0.01, clustering roughly 35% of the bricks into 100 new bricks.

Model	k	Iterations	Variance Threshold	Before	After	Compression Ratio	RMSE
Shockwave	1000	5	0.01	1877783	1621453	0.86349	0.0151
	100	10	0.01	1877783	1620553	0.86301	0.0151
	100	5	0.1	1877783	100	0.00005	0.0425
	100	5	0.01	1877783	1620553	0.86301	**0.0046**
Dragon	1000	5	0.1	123934	67031	0.54086	**0.0077**
	100	10	0.1	123934	66131	0.53359	0.0092
	100	5	0.01	123934	66131	0.53359	0.0096
	100	5	0.1	123934	66131	0.53359	0.0093
	100	5	0.99	123934	66131	0.53359	0.0279
Disney Cloud	1000	5	0.1	275730	73456	0.266405	0.227
	100	10	0.1	275730	72556	0.26314	0.0299
	100	5	0.01	275730	178663	0.64796	**0.0088**
	100	5	0.1	275730	72556	0.26314	0.0283
	100	5	0.99	275730	100	0.00036	0.0659
Chimney	1000	5	0.1	915701	27820	0.03038	0.0109
	100	10	0.1	915701	26920	0.02939	0.0138
	100	5	0.01	915701	401147	0.43807	**0.0046**
	100	5	0.1	915701	26920	0.02939	0.0131
	100	5	0.99	915701	100	0.00010	0.0185

(e) The input parameters k, Iterations and Variance Threshold, with their respective compression results.

Figure 19: The results of clustering-based compression. Four models were used for the comparison of the different clustering parameters. The tweakable clustering parameters are k, The number of iterations and the variance threshold. These are then used to cluster our brick data. The *Before* and *After* columns show the brick count. *Ratio* is the ratio between brick numbers before and after clustering. The RMSE is calculated using one of the frames for each model. All comparisons are done with BC7 compression enabled.

(a) Texture voxel brick data before clustering.



(b) Texture voxel brick data after clustering.

Figure 20: The results of our clustering compression technique. All of these results use Bc7 encoding. Figure 20a shows a slice out of the voxel brick data texture on the GPU. There are visible long chains of black or otherwise greyscale slices which means that these bricks are mostly homogeneous inside that brick. In Figure 20b we again see a slice of the voxel brick data texture, but this time after running our clustering algorithm. There are visibly fewer patches of grayscale bricks when compared to Figure 20a. This shows us that we are removing homogeneous bricks, and keeping unique nodes.

(a) Level set rendering the depth of the primary hit calculated with HDDA.

(b) Fog volume rendering the accumulated density along the ray path.

Method	**Fog Volume**	**SM Occupancy**	**L1 pressure**	**Level Set**	**SM Occupancy**	**L1 pressure**
NanoVDB	4.971	?	?	2.427	?	?
Ours	9.815	95.9%	30.9%	1.092	62.1%	36.6%

(c) Performance numbers between NanoVDB from [Braithwaite and Museth, 2020] and our implementation. All numbers are in milliseconds.

Figure 21: We attempted to benchmark using the same setup as [Braithwaite and Museth, 2020]. Unfortunately, they did not provide a thorough explanation of the benchmark beyond the resolution and the models which were used, these are shown in 21b and 21a. We do not know how exactly the models were placed in the NanoVDB benchmark, so we positioned the objects to cover as much of the screen as possible to capture the lower bound of the ray tracing performance. In Table 21c we show the fog volume and level set render timings in ms. We also show the streaming multiprocessor (SM) occupancy and the L1 pressure. These two metrics provide insight into bottlenecks and potential future optimization targets. Unfortunately these detailed performance metrics were not provided with the NanoVDB results, thus these fields are highlighted with questionmarks.

7 Conclusion

In the formulation of the research questions we already mentioned that there is not a single structure that can do it all. There is always a mix of trade-offs. So one thing we can do is optimize the parts that are the same for the desired structures and then build specialized structures around that. This method is described in Section 4.3. During this thesis, a VDB structure was implemented as it was a balance between structure size, sample timing and efficient HDDA. This structure along with our voxel data compression scheme allowed us to load in full animation sequences into our engine and render them at movie-level fidelity. Compared to previous techniques [Dolonius et al., 2017], that compresses RGB data to between 26.7 and 10.7 bits per voxel (between 7 and 17 times reduction over their original data), we have a better bits per voxel ratio, but do not achieve the same compression rates compared to our original data due to f16 voxels already being smaller than 24-byte RGB color values. Our block compression scheme allows for an 8× reduction in memory usage over the standard 16-bit floating point format. Along with that, we can get another reduction by between 14% and 57% without significant quality loss by clustering voxel bricks which contain roughly the same data. Both of these compression methods do not work for all models out of the box, some models might need to be slightly altered to prevent edge cases as seen in 18e. In Table 21c we see that our main bottleneck when rendering volumes using a sampling technique is computation (ALU bound), instead of memory. The same problem arises with our HDDA implementation, which has low occupancy due to high register usage, but this is not an issue as long as we are not memory-bound. Both of these results are surprising as GPU algorithms are most often memory-bound due to the large number of pixels and buffers that have to be read and written to. So when optimizing the implemented methods the main point of interest would be to reduce the number of steps taken and reduce the register usage of our HDDA implementation. The former can likely be achieved by utilizing a signed distance field as described in Section 2.4.1. A summary of this conclusion is written below by covering each of the listed requirements (as mentioned in Chapter 3.1):

1. The **Asset size** issues of volume data structures are partially solved. Our voxel data has been compressed to the point where most models use less than 64 MB. And by far our largest model (in terms of number of voxels), Shockwave, uses 256 MB for its voxel data. Unfortunately our tree sizes have not been optimized for animations yet which results in each animation frame linearly increasing the tree size by 20 to 30 MB, making long animation sequences impractical.

2. The **Sampling speed** is fast enough to utilize real time frame rates when using a technique like delta tracking as described in Section 2.2.2, or ray marching as shown in Figure **??**. HDDA traversal, as described in Section 2.4.1 is also fast enough to run in real time.

3. **Animation playback** is practically free as we only have to change the starting position of our tree traversal which is done by changing the animation frame index.

4. **Lossy compression** is effective to the point of reducing our voxel data by between 9 to 18 times, by utilizing both compression techniques, without significantly losing visible quality in most cases.

5. **Level of detail** could be added by mip-mapping one level of the voxel data texture. Additional data per internal node could also be added. No testing or benchmarking for either of these techniques has been done though.

7.1 Renders

(a) Path traced half resolution Disney Cloud path traced using multiple scattering.

(b) Ray marched smoke inside the Bistro scene coming from a chimney.

(c) The same scene later on the day shows a different frame in the animation sequence.

Figure 22: Multiple renders made in Breda using the created data structures. Figure 22a shows the largest tested model which was rendered by using path tracing. This was not real-time (about 5 frames per second) because we use multiple scattering, which means that our ray can bounce many times into different directions within the cloud. Figures 22b and 22c show two frames from a video made in the Bistro scene. This was rendered using our ray marcher which does run in real-time together with all other systems in Breda. The way that the light dynamically scatters throughout the chimney smoke is an effect that is currently not done in any AAA game.

7.2 Future research

- One point of improvement to the in Section 4.5 mentioned clustering method could be a more optimized rejection scheme. Currently, we reject if the normalized variance of all brick values is too high. This only allows us to cluster homogeneous bricks and not bricks that contain a smooth gradient, even though there are bricks that are simply a gradient in one specific axis which likely could be clustered. In the future, different distance functions could be used. This could reduce the dependance on our rejection scheme, and maybe purely rely on clustering.

- Another major issue with the clustering compression scheme are the build times. It currently takes multiple minutes to compress both the shockwave and the chimney animation, which makes development iteration times very slow.

- Different texture compression schemes can be experimented with. Certain platforms, like mobile, have different texture compression formats namely, Ericsson Texture Compression (ETC) and Adaptive scalable compression (ASTC). ETC has 1 bit per voxel which is twice the compression ratio of the method described in 4.4. And ASTC can compress between 0.15 and 1.19 depending on certain options. These compression schemes will change the quality of our data but might be worth it on the supported platforms.

- Different tree topologies can be experimented with. Most notably B+Tree's with uniform layer sizes (unlike our current setup with node sizes 32^3, 16^3 and 8^3). This might make it possible to deduplicate many of the bit masks and thus shrink the tree size. Another benefit as mentioned in [Hoetzlein, 2016], by using a tree where all internal nodes are 8^3 we get better ray tracing performance.

- The ideas about running simulations inside our VDB data structure as described in Section 4.2 can be implemented. During this research, there was a very limited time investment in making this feature work. So experimenting with the theorized technique could result in a nice piece of follow-up research.

References

[abje, 2017] abje (2017). quadtree traverser. https://www.shadertoy.com/view/Xs2czK. Accessed: 2023-03-01.

[Amanatides et al., 1987] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10.

[Attrach, 2022] Attrach, S. A. (2022). Vdb: A deep dive. https://jangafx.com/2022/09/29/vdb-a-deep-dive/. Accessed: 2023-02-13.

[Bikker, 2022] Bikker, J. (2022). How to build a bvh – part 1: Basics. https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/. Accessed: 2023-03-13.

[Bouma, 2022] Bouma, D. (2022). Bindless rendering — setup. https://medium.com/traverse-research/bindless-rendering-setup-afeb678d77fc. Accessed: 2023-02-10.

[Braithwaite and Museth, 2020] Braithwaite, W. and Museth, K. (2020). Accelerating openvdb on gpus with nanovdb. https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/. Accessed: 2023-03-22.

[Burnes, 2018] Burnes, A. (2018). Graphics reinvented: Ray tracing, ai, and advanced shading deliver a whole new way to experience games. https://www.nvidia.com/en-us/geforce/news/graphics-reinvented-new-technologies-in-rtx-graphics-cards/. Accessed: 2023-02-07.

[Bálint and Kiglics, 2021] Bálint, C. and Kiglics, M. (2021). Quadric tracing: A geometric method for accelerated sphere tracing of implicit surfaces. *Acta Cybernetica*, 25.

[Cao et al., 2010] Cao, T.-T., Tang, K., Mohamed, A., and Tan, T.-S. (2010). Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90.

[Careil et al., 2020] Careil, V., Billeter, M., and Eisemann, E. (2020). Interactively modifying compressed sparse voxel representations. In *Computer Graphics Forum*, volume 39, pages 111–119. Wiley Online Library.

[Dado et al., 2016] Dado, B., Kol, T. R., Bauszat, P., Thiery, J.-M., and Eisemann, E. (2016). Geometry and attribute compression for voxel scenes. In *Computer Graphics Forum*, volume 35, pages 397–407. Wiley Online Library.

[Disney Animation, 2018] Disney Animation (2018). Clouds. https://disneyanimation.com/resources/clouds/. Accessed: 2023-08-28.

[Dolonius et al., 2017] Dolonius, D., Sintorn, E., Kämpe, V., and Assarsson, U. (2017). Compressing color data for voxelized surface geometry. *IEEE transactions on visualization and computer graphics*, 25(2):1270–1282.

[Ester et al., 1996] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231.

[Hoetzlein, 2016] Hoetzlein, R. K. (2016). Gvdb: Raytracing sparse voxel database structures on the gpu. In *Proceedings of High Performance Graphics*, pages 109–117. Association for Computing Machinery.

[Intel, 2023] Intel (2023). Ispctexturecompressor. https://github.com/GameTechDev/ISPCTextureCompressor. Accessed: 2023-08-31.

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150.

[Kajiya and Von Herzen, 1984] Kajiya, J. T. and Von Herzen, B. P. (1984). Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174.

[Kämpe et al., 2013] Kämpe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):1–13.

[khronos, 2022] khronos (2022). Rendering pipeline overview. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. Accessed: 2023-08-21.

[Kim et al., 2022] Kim, D., Lee, M., and Museth, K. (2022). Neuralvdb: High-resolution sparse volume representation using hierarchical neural networks. *arXiv preprint arXiv:2208.04448*.

[Koch et al., 2020] Koch, D., Hector, T., Barczak, J., and Werness, E. (2020). Ray tracing in vulkan. `https://www.khronos.org/blog/ray-tracing-in-vulkan#Acceleration_Structures`. Accessed: 2023-02-14.

[Kutz et al., 2017] Kutz, P., Habel, R., Li, Y. K., and Novák, J. (2017). Spectral and decomposition tracking for rendering heterogeneous volumes. *ACM Transactions on Graphics (TOG)*, 36(4):1–16.

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

[Laine et al., 2013] Laine, S., Karras, T., and Aila, T. (2013). Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 137–143.

[Lin et al., 2022] Lin, D., Kettunen, M., Bitterli, B., Pantaleoni, J., Yuksel, C., and Wyman, C. (2022). Generalized resampled importance sampling: foundations of restir. *ACM Transactions on Graphics (TOG)*, 41(4):1–23.

[Lin, 2021] Lin, J. (2021). The perfect voxel engine. `https://voxely.net/blog/the-perfect-voxel-engine/`. Accessed: 2023-02-14.

[MacQueen et al., 1967] MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA.

[Museth, 2013] Museth, K. (2013). Vdb: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*, 32(3):1–22.

[Museth, 2021] Museth, K. (2021). Nanovdb: A gpu-friendly and portable vdb data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, pages 1–2. Association for Computing Machinery.

[Nießner et al., 2013] Nießner, M., Zollhöfer, M., Izadi, S., and Stamminger, M. (2013). Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, 32(6):1–11.

[NVIDIA, 2022] NVIDIA (2022). What is path tracing? `https://blogs.nvidia.com/blog/2022/03/23/what-is-path-tracing/`. Accessed: 2023-09-08.

[Oomen, 2022] Oomen, M. (2022). Render graph 101. `https://blog.traverseresearch.nl/render-graph-101-f42646255636`. Accessed: 2023-02-10.

[Quilez, 2008] Quilez, I. (2008). Rendering worlds with two triangles. `https://iquilezles.org/articles/nvscene2008/`. Accessed: 2023-03-31.

[Reed, 2012] Reed, N. (2012). Understanding bcn texture compression formats. `https://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/`. Accessed: 2023-08-23.

[rust ndarray, 2023] rust ndarray (2023). ndarray. `https://github.com/rust-ndarray/ndarray`. Accessed: 2023-08-31.

[Söderlund et al., 2022] Söderlund, H. H., Evans, A., and Akenine-Möller, T. (2022). Ray tracing of signed distance function grids. *Journal of Computer Graphics Techniques Vol*, 11(3).

[Traverse Research, 2023] Traverse Research (2023). vdb-rs. `https://github.com/Traverse-Research/vdb-rs`. Accessed: 2023-08-28.

[van der Laan et al., 2020] van der Laan, R., Scandolo, L., and Eisemann, E. (2020). Lossy geometry compression for high resolution voxel scenes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(1):1–13.

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

[Whitted, 1979] Whitted, T. (1979). An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14.

[Yang et al., 2020] Yang, L., Liu, S., and Salvi, M. (2020). A survey of temporal antialiasing techniques. In *Computer graphics forum*, volume 39, pages 607–621. Wiley Online Library.