Universiteit Leiden
The Netherlands

Opleiding Informatica

Volumetric Ray Tracing with Vulkan

R. F. Voetter

Supervisors:
Dr. K. F. D. Rietveld & Prof. Dr. Ir. F. J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/06/2019

Abstract

Volumetric ray tracing is since the advent of powerful general purpose graphics hardware understood to be a viable alternative to other visualization algorithms, and is preferred due to the better quality of images this method produces. In this thesis we describe an implementation of a volumetric ray tracing software using Vulkan, a new and modern graphics and compute API intended for interfacing with GPUs. Several different ray traversal algorithms are implemented: one based on uniform grids, and four on sparse voxel octrees. An empirical evaluation shows ray traversal over the uniform grid is faster in the general case due to its simplicity, however, if a sparse voxel octree is sufficiently pruned algorithms based on this datastructure can prove to be more efficient.

Contents

1 Introduction

In biomedical research scanners often produce volumetric image data; a sequence of images forming a density scalar field. Each pixel of each of these images represents a 3-dimensional box, commonly called a *voxel*, with its value indicating the density value at the voxel's position in space. Visualizing this data is important to understand the phenomena that are studied, and thus much research went into rendering volumetric datasets. Even though modern graphics hardware is optimized at rendering polygons, and methods exist to transform volumetric datasets into sets of polygons, its often simpler and more efficient to visualize the data using volume ray tracing. This project explores the implementation of such a volume ray tracer with the recent Vulkan graphics and compute API.

1.1 Motivation

The main motivation behind this project is a larger project at the Leiden Institute for Advanced Compute Science (LIACS). This project aims to offer a pipeline for scanning, processing and visualizing volumetric datasets from microscopes. One of the goals of said project is to visualize volumetric datasets on a tiled display environment, the *BigEye* (see Figure 1). This system contains 12 1920x1080 monitors arranged in a 3 by 4 fashion for a total of 5760 by 4320 pixels. Each column of 4 monitors is connected to a single Nvidia GTX 660 graphics processing unit (GPU) with 2 GB video memory each.

Currently employed visualization solutions fail to make use of all GPUs of the BigEye: these programs were not designed to work on such an environment, and can only use a single GPU for rendering. Different operating systems have different solutions for multi-GPU systems. Microsoft Windows for example, solves this by creating a virtual monitor the size of all physical monitors combined and make the program render to that. Individual parts are then sent to the relevant GPUs to be rendered on their connected monitors. Others, such as GNU/Linux, limit the program's output area to the monitors of the GPU the program is rendering with. Suffice to say is that both methods do not provide an acceptable solution.

Figure 1: The BigEye setup. Wallpaper by Tambako the Jaguar, licensed under CC BY-ND 2.0.

The general-purpose GPU programming (GPGPU) API Vulkan provides functionality to write programs optimized for such environments. Furthermore, there is as far as known to the authors at the time of writing no other Vulkan-based implementation of volume ray tracing software, which provides an interesting research opportunity.

1.2 Contributions

In this project, we discuss the implementation of a volume ray tracing program with emphasis on using Vulkan. We evaluate whether Vulkan is an appropriate API for the job, and if its benefits outweigh its drawbacks. While this project does not focus on implementing *multi-GPU* ray tracing software, we do implement multi-GPU functionality by duplicating the rendering architecture across multiple devices to show the simplicity of implementing such a feature with Vulkan. Several different ray traversal algorithms and accompanying data structures will be implemented and evaluated on a number of different GPUs. Furthermore, to get a broad performance estimation, we will also discuss a performance evaluation of the traversal algorithms and compare it to a similar project.

The remainder of this thesis is sectioned as follows:

- In Chapter 2 some of the background work related to this topic will be outlined.
- Chapter 3 will discuss the implementation of the software and will explain the implementation of the rendering architecture.
- Chapter 4 will give a performance evaluation of the different ray traversal algorithms, and also gives a performance comparison with similar ray tracing software.
- Finally in Chapter 5 the results of the evaluation will be discussed and the work will be concluded.

2 Background

We can generally divide the rendering of volumetric data in two camps: Direct Volume Rendering and Indirect Volume Rendering. The latter category refers to preprocessing the volume to a more friendly format before rendering. An example of this method is the well-known marching-cubes algorithm [LC87]. The dataset is transformed into a triangular mesh, which can then be efficiently rendered on common graphics hardware. This method comes with an inherent problem however: as a mesh is a hollow approximation of a surface, it cannot convey volumetric information very well.

Instead we consider a direct volume rendering approach: volumetric ray tracing. In the following sections, the basis of the ray tracing method is explained, as well as GPU programming principles, basic accompanying ray traversal data structures and related work is discussed.

2.1 Volumetric Ray Tracing

The basic ray tracing algorithm is one of the simplest methods of rendering. Images are generated by tracing imaginary rays of light from an eye point through the scene. When the ray intersects with an object, the ray bounces on the object and continues its path. A pixel's color intensity is determined by all interactions of the ray and objects of the scene. This method offers superior image quality when compared to traditional rendering method such as rasterization of triangles, but tracing the ray around comes at a great computational price. Because of this ray tracing is usually not done in a real-time manner, but rather in situations where long rendering time is acceptable, such as in the movie industry.

Volumetric ray tracing, or volumetric ray casting, works in a similar way. Instead of bouncing when a ray intersects an object, the ray is traced through the entire volume. The color intensity associated with a particular ray is determined by all the voxels of the volume it intersects with, often done by following the ray by taking very small steps, and sampling the volume along the way. This task is trivial to parallelize: each ray only reads from the volume, and does not interact with any other ray. For this reason we can utilize the parallel processing power of modern graphics hardware.

2.2 Programming Graphics Processing Units

The basic idea behind graphics processing units is to exploit data-parallel nature of rendering calculations. While traditional GPU rendering pipelines consisted of fixed-function logic modules dedicated to render polygons, modern GPUs can be programmed to perform as a general purpose parallel computing unit. To achieve high performance, a GPU is divided into groups of threads called *compute units*. Threads in a compute unit execute in lockstep, which means each thread must always execute the same instruction at a time. This architecture reduces the amount of hardware circuitry needed, but comes at a price: GPUs respond poorly to heavily branching programs. When a branch such as an if/else-statement is encountered, unless all threads take the same branch, all threads of a compute unit will have to evaluate both cases. This wastes precious computing power for threads where a branch is not taken but still has to be evaluated.

Vulkan[1] is a low-level API intended for interfacing with GPUs. In contrast to its predecessor, OpenGL, Vulkan is intended to be flexible and modern, which has a number of advantages that are relevant to this project:

- Vulkan is both a graphics and compute GPU programming API. This removes the need for complex co-operation between a compute API such as CUDA and a graphics API such as OpenGL for visualization software.
- Multi-GPU support is an integral part of Vulkan. In combination with explicit resource control, fine grained allocation of GPU time and resources is possible.
- Vulkan is designed to be platform independent. This would for example allow a relative simple implementation of a program that can be executed without recompilation on hardware of different types, manufacterers and families.
- Vulkan can be extended by manufacturers with platform *dependent* functionality. A notable extension is `VK_KHR_surface`, which must be enabled to allow Vulkan programs to render to a monitor. This allows Vulkan programs to work as well on systems without suitable video output if one does not require such functionality.

Note that Vulkan does not in fact run on a GPU, but rather deals with managing related tasks such as initialization, memory and resource management, and dispatching calculations. Vulkan's flexibility comes at the price over a higher code complexity however, and thus greater care has to be taken when implementing Vulkan programs. This is partly mitigated by *validation layers*, debug components which can be enabled to spot common programming errors.

Code that runs on a GPU is called a *shader program*. Shader programs can be written in various languages, but the efforts behind Vulkan also include a standardized binary representation: SPIR-V. In this research shader programs are written in the OpenGL Shading Language (GLSL), and are compiled offline to SPIR-V.

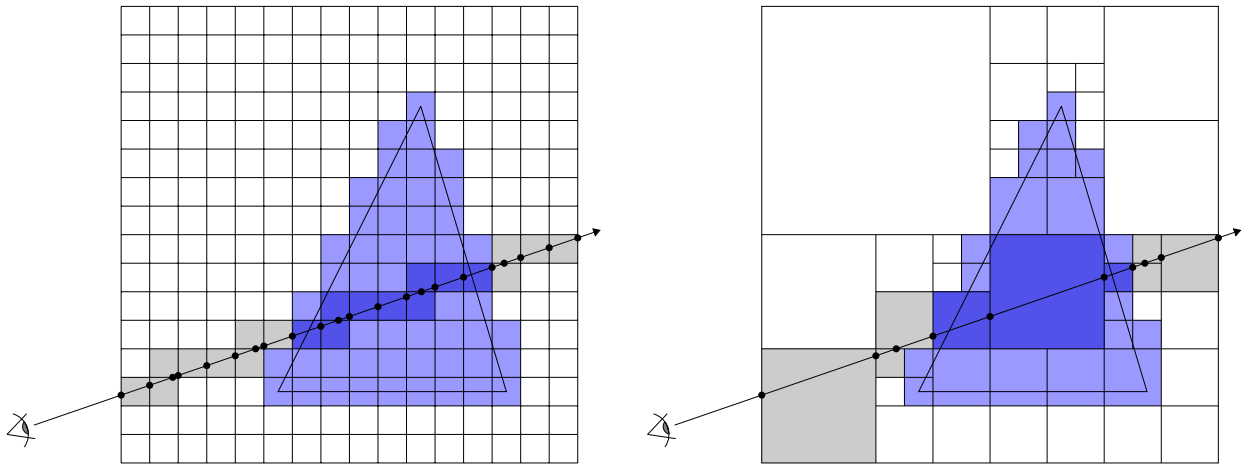
2.3 Acceleration Data Structures for Volumetric Data

Acceleration structures are often used to store volumetric data when ray tracing to increase performance. These data structures often exploit the spatial structure of the data to reduce the amount of ray tracing steps required. Many different acceleration structures have been devised, of which the two most relevant are explained in the following sections.

2.3.1 Uniform Grid

The simplest method of storing volumetric data is in an *uniform grid*, of which a schematic example of traversal is given in Figure 2a. This uniform grid is a 3-dimensional array, where each element contains the density value of a represented voxel. Uniform grids can be stored on a graphics processing unit in 3-dimensional *texture* objects, where voxel densities are represented by texture elements, *texels*. Efficient constant-time lookup is provided by specialized hardware components to deal with textures, such as caches optimized for 3-dimensional spatial access as opposed to 1-dimensional access patterns commonly used for general memory.

[1] `https://www.khronos.org/vulkan/`



(a) Traversing a uniform grid

(b) Traversing a sparse voxel octree

Figure 2: Cross sections of traversing different acceleration structures

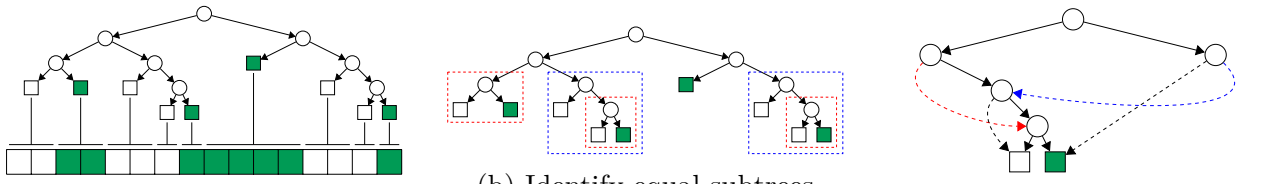
A disadvantage of storing volumetric datasets as uniform grids is that they provide no means of compressing data. This means that each voxel of a large volumetric dataset requires an element in the uniform grid, even if the dataset is mostly similar or empty, which limits the maximum resolution at which the dataset can be stored. For example, a 2048^3 volumetric dataset stored as a uniform grid requires 8 GiB of space if 8 bit per voxel is assumed. In contrast, a typical consumer-grade GPU has about 4 to 8 GiB of memory at the time of writing, while high end GPUs have up to 32 GiB.

2.3.2 Sparse Voxel Octrees

Octrees are a recursive tree-like data structure that can be used to partition 3-dimensional space. Each node of the octree has the shape of an axis-aligned cube, and each intermediary node is subdivided into eight equally-sized octants. Leaf nodes store information about the original volume. A *sparse* octree is obtained by pruning subtrees which contain similar space that can be stored in a single leaf.

Glassner [Gla84] first showed its applications in ray tracing by storing in each leaf node a list of which objects of the scene it intersects with. Traversing the tree happens in a top down manner: The ray is tested against a node, and if it intersects its children are traversed. If a leaf node is traversed, the ray is tested against all objects of the list. Large numbers of objects can be be ruled out of intersection with the ray simultaneously, which reduces the total number of intersection tests that need to be performed, and thus increases ray tracing performance.

A sparse *voxel* octree is obtained by letting each leaf node represent a single voxel. A large cube of similar voxels can be represented by a few nodes, which also makes the tree suitable as a compression structure. With this more complex data structure however, comes also the cost of a more complicated traversal algorithm, to which GPUs not always respond well. A schematic example of the leaf nodes of a sparse voxel octree and the leaves a ray might intersect is given in Figure 2b.



(a) A binary tree (b) Identify equal subtrees (c) Eliminate equal subtrees

Figure 3: Reducing a binary tree into a directed acyclic graph

2.4 Related Work

Ever since the beginning of computer graphics much of research has gone into rendering of volumetric models. The basis of 3-dimensional ray tracing was laid in 1987 by Amanatides & Woo [AW+87], who showed a simple and efficient method of determining the next voxel when casting a rays over an uniform grid.

Based on this work, Laine & Karras [LK10] present a state-of-the-art algorithm for ray tracing Sparse Voxel Octrees. To improve rendering quality and reduce the size of the tree, each voxel is augmented with *contour* information: a pair of parallel planes approximating the intersection of the represented surface. By defining the shape of a voxel to be that of a cube intersected with all contours of it and its ancestors, the quality is greatly increased. However, as this method is primarily focussed on encoding surface information of a mesh, it is ill-suited as a volumetric data.

Kämpe et al. [KSA13] improve the ratio of information that can be stored in an octree by observing that the location of a voxel in an octree is not defined by its node, but only by the path taken from the root of the tree to the node. A sparse voxel octree is reduced to a directed acyclic graph (DAG) by eliminating equal subtrees, bringing down the amount of nodes in the data structure losslessly. An example of an equivalent reduction from a binary tree to a directed acyclic graph is shown in Figure 3.

Brayns[2] is part of the Blue Brain project [Mar06], and offers a visualizer for different types of data, including volumetric. A GPU-based rendering engine powered by Optix [Par+10] is available, however, at the time of writing volumetric rendering is only implemented in the CPU-based rendering engine powered by Ospray [Wal+16]. Brayns can output to a variety of backends, such as local or via a web interface. Of particular interest is the ability of Brayns to stream output to a video wall.

GVDB[3] [Hoe16] is a GPU-based ray tracing library aimed at visualization and computation of volumetric datasets, based on the VDB voxel data structure developed by Museth [Mus13]. GVDB uses CUDA and aims to solve the problem of animation for volumetric rendering, while also providing efficient GPU-based ray traversal, and general ray tracing based compute capability. It is similar in rendering functionality when compared to the implementation described later on in this thesis, but uses a different optical model and acceleration structure.

[2] `https://github.com/BlueBrain/Brayns`
[3] `https://github.com/NVIDIA/gvdb-voxels`

3 Implementation

The implementation of this project is called *Xenodon*[4]. Xenodon is written in C++17 against the Vulkan 1.1 specification, using the Vulkan-Hpp[5] C++ headers. Program options are specified via command line parameters, and occasionally via configuration files if options might become too complex. This chapter describes the implementation of Xenodon by first giving an overview of the application's architecture and later going into detail of the implementation of each of the different traversal algorithms and accompanying data structures.

3.1 Architecture Overview

Xenodon is divided into two main subsystems: a frontend and a backend. The backend part deals with platform abstraction and initialization of render outputs. This way the frontend can deal with rendering related tasks indifferent of what kind of surface it is actually rendering to, or where input events come from.

3.1.1 Backend

The backend provides access to the host system's resources through a *Display* interface. This interface can be used to query for a list of *Render Devices*: a fully initialized GPU, along with other related resources. Each Render Device is associated with a number of *Outputs*, interfaces representing a renderable surface such as a monitor. Each output occupies an *output region*, a rectangle in a virtual *display region* which represents the backend's output. See Figure 4. Backends also gather system events, for example when the user presses a key, and emits these through a callback to the frontend.

Each output is associated with a number of Vulkan *images*: GPU memory areas used for storing image data. These images are special however, as they are intended for receiving the render output. Memory is typically received from the host operating system's display server via a Vulkan *swapchain* object. When the Vulkan *present* function is called at the end of a frame, the image will be sent to the surface connected to the swapchain (such as a monitor) to be displayed. The swapchain's images may randomly invalidate (when the user resizes the program's output window for example), in which case the backend emits an appropriate event.

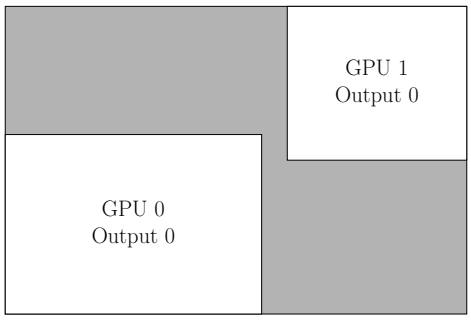
Three different backends have been implemented:

X.org
X.org mode can either be used to start the program in a single window like any other desktop program, or can be passed a configuration file specifying on which X.org screens[6] it should render. In the latter option, for each of the screens specified a full-screen window is opened. Events are gathered through X.org's event system.

[4] `https://github.com/Snektron/Xenodon`
[5] `https://github.com/KhronosGroup/Vulkan-Hpp`
[6] An *X.org screen* is a virtual monitor that encompasses all monitors connected to a GPU.

GPU 0 Output 3	GPU 1 Output 3	GPU 2 Output 3
GPU 0 Output 2	GPU 1 Output 2	GPU 2 Output 2
GPU 0 Output 1	GPU 1 Output 1	GPU 2 Output 1
GPU 0 Output 0	GPU 1 Output 0	GPU 2 Output 0

(a) Possible display region for the BigEye, with 12 output regions



(b) Possible display region for a double-GPU system. The gray area is not rendered.

Figure 4: Examples of display regions

Because this backend depends on system libraries which are not always available (some X.org libraries are for example not always available on headless systems), this backend can be disabled at compile time.

Direct
Direct mode allows the program to take control of the displays without any display server running. To this end the Vulkan `VK_KHR_display` extension is used to directly render to the display. Using this extension we are able to query information about monitors connected to the system such as resolution and physical size. Information such as position relative to other monitors is not available though, and instead has to be passed to the program in a configuration file.

Since `VK_KHR_display` only works on GNU/Linux at the time of writing, keyboard events are gathered through Linux' input subsystem, *evdev*. The desired input device path is passed via above configuration file, and is taken over via the `EVIOCGRAB ioctl` call to enable the program to receive input events without it being ran with super user rights. Due to dependencies on the Linux kernel, this backend can also be disabled at compile time.

Headless
Headless mode allows the program to render without any video output present on the system. In this case, an output's images are not received from a swapchain, but are allocated by the program instead. After a frame is rendered images are either discarded so the renderer can be benchmarked, or can be downloaded from GPU memory and saved on disk as a PNG-image. GPUs to be used and their render areas are again specified via a configuration file. No events are generated by this backend, which means that no user interaction can happen.

3.1.2 Frontend

The render pipeline is similar for each different ray traversal algorithm and data structure. It is divided into a one-time initialization procedure and a render procedure repeated for each frame rendered.

Initialization

Initialization of the rendering pipeline comprises of the following steps:

1. In the first step, common resources are initialized. This includes uploading the dataset to the GPU and creating the *uniform buffers*, buffers in GPU memory which hold rarely changing shader inputs. Each output gets its own uniform buffer containing the output region's rectangle, the total display region's rectangle, a user-supplied voxel density multiplier, and the size of the model and of an individual voxel.
2. In the second step, the Vulkan pipeline object is created. This pipeline object specifies a sequence of operations that are executed when rendering by configuring the GPU's fixed-function stages. Different types of pipelines are available in Vulkan, for example the most common *graphics* pipeline which can be used for rasterization of triangles. In our implementation however, a *compute* pipeline is used. This is the simplest pipeline, in which only a single shader program is executed: in our case the ray tracing shader.
3. Next, the *descriptor sets* and related resources are initialized. These objects contain information how GPU resources such buffers and images are to be bound to the shader program's input variables. Note that in this step, the descriptor sets are only created and bound to a shader input variable, but no information about buffer resources is written to them yet. This is because such information may change over frames, for example when an output's images invalidate.
4. Fourth, the *command buffers* are created: CPU buffers in which rendering commands are to be stored. When a command buffer is submitted to a particular GPU, all commands in it are transferred at once and executed in an out-of-order fashion. One command buffer is created for each output, to make sure they can be written to by the CPU and read from by the GPU at the same time without race conditions.
5. Lastly, the relevant data is uploaded to the uniform buffers and descriptor sets. This binds the actual Vulkan buffer and image to the shader program's inputs. When an output's images invalidate, only this step has to be repeated.

Rendering

A frame is rendered by filling the command buffers with rendering commands and submitting that to a GPU. Rendering is further divided in the following steps, which are repeated for each output of each render device:

1. In the first step, an output image and an accompaying command buffer are selected by the video driver. We then emit a command into the command buffer to *transition* it to its correct state. This operation ensures no other operations are currently working with the image.
2. In the second step, the actual ray tracing is performed. This involves enabling the renderer's resources (the pipeline and the descriptor sets), and dispatching the ray traversal shader program for each pixel of the output. Per-frame changing shader inputs such as camera viewing direction and position are also uploaded in this stage. These variables are passed directly to the shader program by encoding them into the command buffer through a Vulkan *push constant buffer*. In cases of often changing variables this method is more efficient, as it saves binding and uploading variables through uniform buffers.

3. Finally, we transition the output image yet again to let the Vulkan implementation know that any image operations (and thus by extension the ray traversal shader program) must be completed before advancing.

After all commands for each output have been recorded into a command buffer and submitted to their GPU, we request the backend to present the current output images to their surfaces.

The render viewpoint is specified to the rendering system via a camera structure. This structure can be controlled via two main methods, selectable via a command line parameter:

Orbit The orbit camera controller responds to the backend's callback events such as key presses to rotate the camera. This allows the user to interactively view the dataset, by rotating the camera around the origin and zooming in and out.

Script This camera controller reads camera transformation information from a file. Each frame, the next camera transformation is fetched from the input file, and when the end of the file is reached the program stops. This camera is useful for benchmarks for example, as the same path can be taken each time the program is invoked.

3.2 Ray Traversal

Ray traversal happens for every shader in a similar manner: The ray traversal shader program is launched for every pixel of every output. A ray is calculated from the camera's eye point, viewing direction and the pixel's location in the final image. Calculation of the pixel's color is also similar in every shader, as it is solely based on which voxels the ray passed through when traversing the input volume.

In the following sections we discuss this common functionality, as well as the details of each individual traversal algorithms. To this end, the following common (immutable) variables are introduced:

- $\vec{v}$ is the camera's viewing direction.
- $\vec{o}$ is the ray *origin*, the camera eye's viewpoint.
- $\vec{d}$ is the ray *direction* associated to the pixel that is currently being processed.
- $\vec{S}$ is the dimension of the input volume. In octrees, all elements of this vector are equal and a power of 2.

3.2.1 Ray Generation

The first step in an invocation of a ray traversal shader program is to calculate the ray direction $\vec{d}$ corresponding to a pixel p the shader program is being executed for. This is done by projecting p on a virtual *image plane*, which represents the display region in world coordinates. The ray $(\vec{o},\vec{d})$ is then defined by the ray that starts at the camera's eye point $\vec{o}$ and passes through the projection of p. See Figure 5 for a schematic overview.

Some ray traversal algorithms depend on dividing a value by $\vec{d}$. To avoid divide-by-zero errors we adjust $\vec{d}$ element-wise, were any element equal to 0 is replaced by a tiny non-zero value.

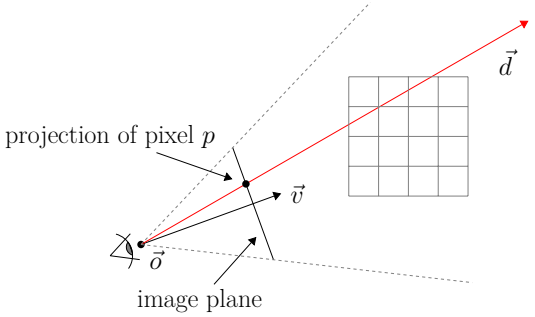
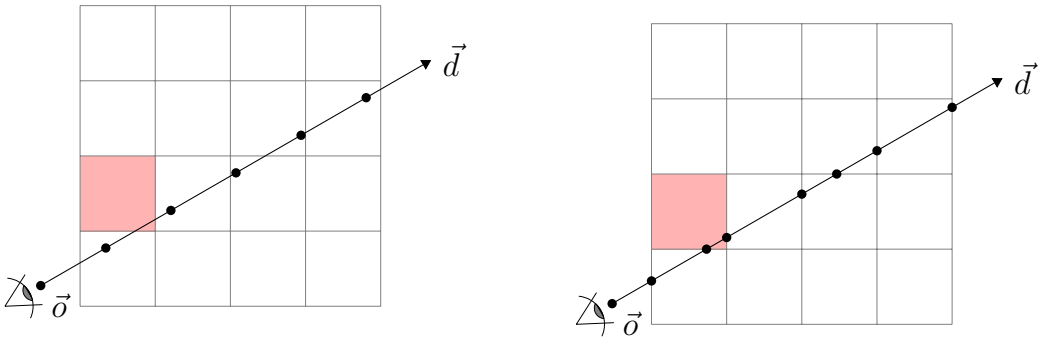


Figure 5: Schematic overview of generating ray $\vec{d}$ (cross section)



(a) Sampling by taking a constant step

(b) Sampling with variable steps

Figure 6: Different sampling methods

3.2.2 Shading and Compositing

Voxel densities are stored in a similar way for each of our implemented acceleration data structures. For each voxel of the volumetric dataset a 24-bit RGB (8 bit per channel) value is stored, which indicates the emission color of that particular voxel. We then obtain the color intensity I_p of a pixel p by integrating over the volume along the ray $(\vec{o}, \vec{d})$ associated to p. Due to its computational cost, we do not consider the scattering of light. Instead, I_p is calculated by a simple emission-only optical model as descibed by Max [Max95], given by Formula 1.

$$I_p = E \cdot \int_{t_{min}}^{t_{max}} c(\vec{o} + \vec{d} \cdot t) dt \tag{1}$$

where $c(\vec{x})$ returns the RGB emission value at $\vec{x}$, and where the ray $(\vec{o}, \vec{d})$ enters the volume at $\vec{o} + \vec{d} \cdot t_{min}$ and exits it at $\vec{o} + \vec{d} \cdot t_{max}$. E is a user defined emission coefficient, which can be set by the user via a commandline parameter.

Formula 1 is implemented by iterating over all voxels that $(\vec{o}, \vec{d})$ passes through, multiplying their emission color values by the intersection distance of $(\vec{o}, \vec{d})$ through the voxel, and summing them. In contrast to an implementation where the ray is simply divided into a number of equally spaced

steps, this method is not vulnerable to errors where a voxel is accidently not included in the final color calculation. For example, in Figure 6a the density in the red cell is not sampled even though the ray passes through it, whereas in Figure 6b all voxels are correctly sampled. Additionally, in our implementation the effect of a voxel's emission color is proportional to the distance travelled through it, in contrast to the other method, where a voxel's emission color is proportional to the distance by which each of the steps along the ray are seperated.

3.2.3 Digital Differential Analyzer (DDA) Traversal

The most basic traversal algorithm we implement is a variation on the algorithm described by Amanatides & Woo [AW+87]. This algorithm is a variation on the *Digital Differential Analyzer* line drawing algorithm. The acceleration structure accompanying this traversal algorithm is a uniform grid, as described in Section 2.3.1.

In our implementation, voxel data is loaded from disk as a TIFF image. Each directory of the TIFF image represents a xy-plane of the grid. The volumetric data is then uploaded to each GPU as a 3-dimensional Vulkan image object. Each element is stored as a 32-bit RGBA value, because 3-dimensional images with 24-bit RGB values is not supported on every GPU. The image is accessed through a *sampler*, a Vulkan object that specifies a the way an image is read and filtered by a GPU's specialized texture circuitry.

The main idea of this algorithm is to step along the axis-aligned planes seperating the voxels. The voxels of the volume are assumed to reside at integer locations, which means that each seperating plane also resides at an integer location. A vector $\vec{s}$ saves the distance along the ray to each of the seperating planes. The distance to the next intersection of a voxel can then be calculated by finding the smallest element of $\vec{s}$. If the smallest element of $\vec{s}$ is $\vec{s}.i$ for $i \in x, y, z$, then the ray can be advanced one voxel by adding the distance it takes for $\vec{d}$ to advance one voxel in direction i, $\vec{t_\Delta}.i$, to $\vec{s}.i$. Algorithm 1 outlines this traversal algorithm, and Figure 7 gives a schematic overview.

On line 1 of the computation, the intersections of $(\vec{o}, \vec{d})$ with the volume is calculated by a ray/box intersection as discribed by Glassner [Gla89, Chapter 4]. This function returns the nearest and furthest intersection distance of the ray $(\vec{o}, \vec{d})$ and the axis aligned box from $\vec{0}$ to $\vec{S}$. If the ray misses the box, t_{min} will be greater than t_{max}. If the origin $\vec{o}$ is inside the box, t_{min} will be less than 0.

The original algorithm by Amanatides & Woo uses heavy branching in their implementation, which is inefficient on a GPU. To optimize this, we utilize a *boolean vector* indicating which element of $\vec{s}$ is the smallest, on line 12 of Algorithm 1. Line 17 then advances the ray to the next voxel, while line 18 advances the position of the currently sampled voxel in the dataset. Additionally, we modify the original algorithm further by calculating the intersection distance of ray $(\vec{o}, \vec{d})$ and the current voxel every step by finding current ray distance by the smallest value of $\vec{s}$. The intersection distance is then obtained by calculating the difference with the previous ray distance t on lines 14 and 15.

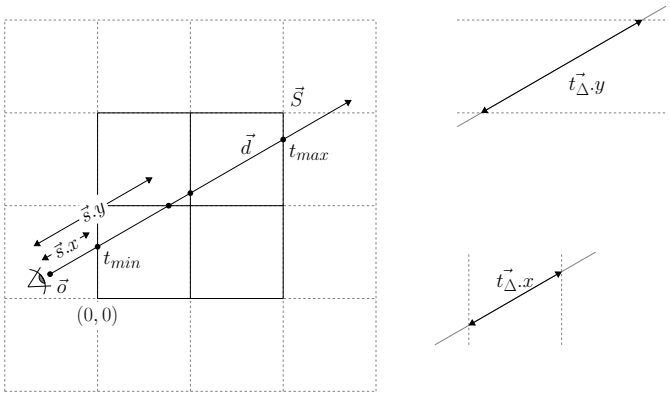


Figure 7: Schematic overview of the DDA algorithm's variables (cross section)

Algorithm 1 The DDA traversal algorithm

Require: V is a 3-dimensional texture of dimension $\vec{S}$

$(t_{min}, t_{max}) \leftarrow \text{INTERSECT}(\vec{o}, \vec{d}, \vec{0}, \vec{S})$ ▷ Calculate the intersections of $(\vec{o}, \vec{d})$ and the volume.
if $t_{min} > t_{max}$ **then** ▷ The ray misses the volume.
    return $\vec{0}$
end if
$t_{min} \leftarrow \text{MAX}(t_{min}, 0)$ ▷ Advance the ray until the eye, if its inside the volume.
$\vec{v} \leftarrow \vec{o} + \vec{d} \cdot t_{min}$ ▷ Calculate the ray entry location.
$t_\Delta \leftarrow |1.0/\vec{d}|$
$\vec{s} \leftarrow (\text{SIGN}(\vec{d}) \cdot (\lfloor \vec{v} \rfloor - \vec{v} + \frac{1}{2}) + \frac{1}{2}) \cdot t_\Delta$
$\vec{c} \leftarrow \vec{0}$
$t \leftarrow 0$ ▷ The distance relative to t_{min} of the entry of a voxel
while $t < t_{max} - t_{min}$ **do**
    $\vec{mask} \leftarrow \vec{s} \leq \text{MIN}(\vec{s}.yzx, \vec{s}.zxy)$ ▷ Element-wise comparison.
    $u \leftarrow \text{MIN}(\vec{s}.x, \vec{s}.y, \vec{s}.z)$ ▷ Calculate the distance relative to t_{min} of the exit of a voxel
    $step \leftarrow u - t$
    $t \leftarrow u$
    $\vec{c} \leftarrow \vec{c} + \text{TEXTURE}(V, \lfloor \vec{v} \rfloor) \cdot step$
    $\vec{s} \leftarrow \vec{s} + \vec{t_\Delta} \cdot \vec{mask}$ ▷ Update the distances to the seperating planes
    $\vec{v} \leftarrow \vec{v} + \text{SIGN}(\vec{d}) \cdot \vec{mask}$ ▷ Update the current position in the volume
end while
return $\vec{c}$

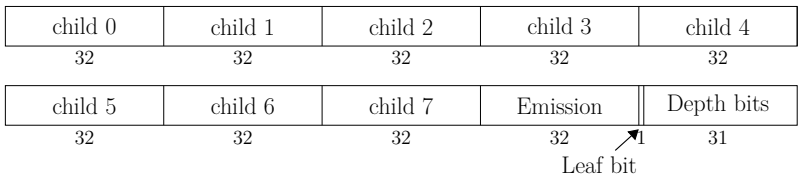


Figure 8: An octree node. Note that this schematic may not represent the actual bit layout in memory because of the host system's endianness.

3.2.4 Sparse Voxel Octrees Construction

All sparse voxel octree traversal algorithms presented in this research use the same general octree acceleration structure. Each node of the octree contains 8 32-bit child pointers, a 32-bit RGBA (alpha is used for padding) emission value, and a 32-bit integer indicating a leaf node and the depth of the tree. The last field is a bitfield, of which the most significant bit indicates whether the node is a leaf and the other 31 bits indicate the node's depth. See Figure 8. The index of a child pointer in the node structure relates to the location of the child in the parent. This index can be calculated by $i = f(x) \cdot 4 + f(y) \cdot 2 + f(z)$, where $f(a) \to \{0, 1\}$ returns 1 if the node resides in the upper half the parent node in axis a.

We store the tree in a single array of nodes, where a child pointer indicates an index of that array. The root node of the tree always resides at index 0. Using 32-bit values allows about $2^{32} = 4'294'967'296$ nodes, or a maximum size $2^{32} \cdot 40$ bytes = 160 GiB. As discussed in Section 2.3.1, this amount is more than enough to fill all memory of current high-end GPUs.

Sparse voxel octrees are constructed in a simple top-down fashion. Starting at the root node, each node is constructed by calculating the average emission color value of the volume represented by the node. Construction of the tree continues depending on a heuristic function: if the node is decided to be *split* according to the heuristic, its 8 child nodes are recursively added to the tree. Otherwise, the construction stops and the node is marked as a leaf. Note that because each dimension is halved when another level is added, the volume represented by the root node of the tree must be a cube where each side contains a power of two number of voxels. The size of the root node cube is given by the next power of two equal or larger than the largest element of the source volume's dimensions. To guarantee an accurate representation, a node is always split if it intersects the boundary of the source volume.

In the simplest case, the split heuristic function might determine to split the tree only when all voxels contained in a node contain different emission colors. This yields the most accurate representation of the volume: if construction a node is terminated before it reaches the most detailed level, it is only because all voxels in it have the same emission color, and no information is lost.

If the source dataset contains tiny variations, for example due to a microscopy scanner picking up background noise, above method might yield a very large tree. Each subvolume will cause the the tree to split, even when all voxels in it might appear to have the same emission color. So solve this, we utilize a *lossy* split heuristic: a node will be split if the voxels in the volume represented by it are different *enough*. This yields a smaller tree by terminating construction on nodes containing low

detail, while preserving subvolumes with high detail. Lower memory consumption may be traded in for detail by making the heuristic split less aggressively.

We implement 2 different lossy split heuristics:

standard deviation This heuristic calculates the standard deviation of the voxel in the subvolume represented by a node while constructing it. Differences are calculated by the euclidean distance between the RGB values of two emission colors. A split is then decided by comparing the standard deviation to some threshold.

maximum difference, channel-wise This heuristic calculates the largest difference in the subvolume represented by the cube for each of the emission color channels. The largest of these values is then also compared to a theshold to decide whether a node is split or not.

Additionally, we reduce the tree's size by tranforming the sparse voxel octree into a directed acyclic graph (DAG) similar to the method described by Kämpe et al. [KSA13]. Our implementation achieves this by storing a mapping of each unique Node to a pointer of where it occurs in the tree's storage array. Before a node is inserted into the tree, it is first looked up in the mapping. If it exists, the pointer of the node is returned. Otherwise, a new node is added to the tree's storage and the node is inserted into the mapping. Note that this means that the children of a node must be inserted into the tree *before* its parents: the parent node's complete information must be available when its inserted into the tree, including its child pointers. In our implementation, a node is thus constructed in the following order: first, determine if the node will be split. If so, recursively construct its children. Finally, insert the node into the tree if it does not exist yet.

3.2.5 A Naive Octree Traversal Algorithm

The simplest octree traversal algorithm we implement repeatedly finds the node for a point along the ray, calculates the intersection distance of the ray and the node, and advances the ray by this amount. The color intensity I_p is calculated by for each node that the ray intersects multiplying the node's emission color by the intersection distance of the ray through it, and adding these. In this and the following ray traversal algorithms, the tree is assumed to have a side length of 1 in world coordinates and is placed at the origin.

Algorithm 2 implements the `find` function: the function starts at the root of the tree, and every iteration selects the child node the point $\vec{v}$ is in until a leaf is found. Function `leaf` simply determines if the leaf bit of a node is set, and `child` retrieves the child pointer of tree T at index i. On line 10 v is compared to the center position of node n, and returns a boolean vector where an element is set if v is in the upper octant of n in that axis. The child index i is then calculated on line 11 by the method described in Section 3.2.4.

3.2.6 A Depth First Traversal Algorithm

One might observce that the sparse voxel octree traversal algorithm outlined in the previous section is not very efficient: for each step along the ray, the *entire tree is traversed* from root to leaf, even though the actual path taken only differ by a few steps from the previous one. We could avoid this by constructing a new, recursive depth-first algorithm, however, in the common case GPUs

Algorithm 2 Find the node which a point lies in

Require: T is an array of nodes, representing a sparse voxel octree

function FIND($\vec{v}$)
  $n \leftarrow 0$ ▷ Initialize n with the root node.
  $l \leftarrow 1$ ▷ The side length of the cube represented by n.
  $\vec{x} \leftarrow \vec{0}$ ▷ The position of the corner of n. n occupies from $\vec{x}$ to $\vec{x} + (l, \vec{l}, l)$
  loop
    if LEAF($T[n]$) **then** ▷ Check if node n is a leaf.
      return $(n, \vec{x}, l)$ ▷ Return the node index, offset and side length.
    end if
    $l \leftarrow \frac{1}{2}l$ ▷ We go down a level, so the size halves.
    $\vec{mask} \leftarrow \vec{v} \geq \vec{x} + l$ ▷ Build a mask determining which octant the $\vec{v}$ is in.
    $i \leftarrow \vec{mask}.x \cdot 4 + \vec{mask}.y \cdot 2 + \vec{mask}.z$ ▷ Calculate the index of the child.
    $\vec{x} \leftarrow \vec{x} + \vec{mask} \cdot l$ ▷ Adjust the offset to the new node.
    $n \leftarrow$ CHILD($T[n], i$) ▷ Look up child i of node n.
  end loop
end function

do not respond well to this at all. Due to this reason, recursion is forbidden in GLSL. Instead we implement a depth-first traversal algorithm by means of a stack.

In the naive implementation of this new algorithm we are required to save two values on the stack: a node, and the index of the child that is currently examined. First, we intersect the ray with the child of the node at the top of the stack. If the ray misses the child, we simply advance to the next child and repeat the process. If the ray hits the child and it is not a leaf we traverse down the tree by pushing the child onto the stack along with the child's first child, and repeat the process. If the child is a leaf instead, we calculate the intersection distance of the ray through it and add it to our total emission intensity I_p, pop the top of the stack, and advance to the next child. When there are no more childeren of the node at the top of the stack to advance to, also pop the stack. If the stack is empty, the entire tree is processed and the traversal procedure ends. Note that this method does not traverse the tree in any particular order along the ray.

We optimize this algorithm again by only pushing to the stack when there is another child left to process in the current node when traversing down the tree. This avoids popping the stack multiple times in a row. By popping a node we then immediately traverse up the tree to the first node-and-child pair that still needs to be processed.

3.2.7 Efficient Sparse Voxel Octrees

Similar to the previous algorithm, *efficient sparse voxel octrees* (ESVO) by Laine & Kerras [LK10] implements a ray traversal algorithm based on stacks. This algorithm operates in a slighty different manner however: voxels are traversed along the ray in an ordered fashion, from nearest to furthest. When a node is intersected, traversal continues by determining which of its children the ray intersects first and pushing that on the stack. After the subtree of this first child is processed, the next child

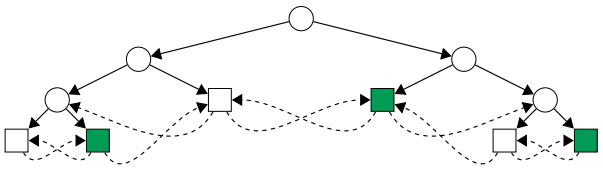


Figure 9: A binary tree with ropes

along the ray in the node is determined and processed. If all children that the ray intersects are processed, the tree is traversed back up by popping a node of the stack.

This traversal algorithm was designed for rendering opaque volumes: a ray is cast into the dataset, and the algorithm computes the nearest intersection. Additionally, Laine & Kerras explain the usage of *contours* to refine the obtained image, which are only useful when rendering mesh data. We implement a modified version of *efficient sparse voxel octrees* that is suitable for rendering true volumetric datasets.

The new version is obtained by making a few modifications to the original algorithm: first, instead of terminating the ray traversal process when a leaf node is hit, we simply add the emission color to our total color intensity I_p and act as if the ray missed the leaf. Secondly, we do away with any logic concerning countours. Finally, we adopt the algorithm for use with the data structure discussed in Section 3.2.4, as our sparse voxel octree structure differs from that of the original implementation.

3.2.8 Rope Trees

In general, a sparse voxel octree ray traversal algorithm finds all leaf nodes of the tree that intersect with a ray. In the algorithm outlined in previous sections, this is all achieved by repeatedly traversing the nodes of the tree, and accumulating the color intensity I_p if that node happens to be a leaf. This means that a significant portion of the time, the traversal algorithm will be processing inner nodes.

In this section we present an implementation of *ropes* for sparse voxel octrees, analogous to how these are implemented by Havran et al. in *Ray Tracing with Ropes* [HBZ98]. The tree structure is augmented by giving each leaf node a list of pointers (ropes) to their directly adjacent neighboring nodes. The ray traversal process can then be accelerated by following the ropes after a leaf node is processed, which avoids a complex procedure of traversing up and down the tree. Note that a drawback of this data structure is that it cannot be subjected to the directed acyclic graph transformation outlined in Section 3.2.4. See Figure 9 for a schematic of a binary tree with ropes.

We keep a total of 6 pointers per leaf node, one for each face of the cube that represents the node. When a face of a leaf is adjacent to more than one leaf node, the rope points to the smallest common ancestor of them. These are stored in the child pointers of the leaf nodes, which are otherwise empty. When there is no neighbor adjacent to a face of a node, for example on the edge of the

volumetric dataset, the rope points to the root (index 0).

Algorithm 3 gives the pseudocode of the algorithm. On lines 6 through 9 the nearest node that the ray intersects with is computed, and the emission color of the ray through that node is calculated. This requires a special case because the ray origin might start inside a node (when the camera is within the volume). On line 12, a modified version of Algorithm 2 is used that accepts an initial node to start searching from. The initial side length (l in Algorithm 2) is determined by the depth field of the initial node. Line 16 determines which face of the current node n the ray $(\vec{o}, \vec{d})$ leaves and translates this into a child index. Child pointer f of node n is then the rope that leads to the adjacent node in the direction of the ray.

Algorithm 3 Rope tree traversal

Require: T is an array of nodes, representing a sparse voxel octree with dimension $\vec{S}$

1: $(t_{min}, t_{max}) \leftarrow \text{INTERSECT}(\vec{o}, \vec{d}, \vec{S}))$
2: **if** $t_{min} > t_{max}$ **then**
3: **return** $\vec{0}$
4: **end if**
5: $t \leftarrow \text{MAX}(t_{min}, 0)$
6: $(n, \vec{x}, l) \leftarrow \text{FIND}(\vec{o} + \vec{d} \cdot t)$
7: $(u_{min}, u_{max}) \leftarrow \text{INTERSECT}(\vec{o}, \vec{d}, \vec{x}, \vec{x} + l)$
8: $t \leftarrow u_{max}$
9: $\vec{c} \leftarrow \text{EMISSION}(n) \cdot (u_{max} - \text{MAX}(u_{min}, 0))$ ▷ Calculate the emission of the first node
10: **while** $n \neq 0$ **do**
11: $\vec{v} \leftarrow \vec{o} + \vec{d} \cdot t$
12: $(n, \vec{x}, l) \leftarrow \text{FIND\_CHILD}(n, \vec{v})$ ▷ Find the child of n at $\vec{v}$.
13: $(u_{min}, u_{max}) \leftarrow \text{INTERSECT}(\vec{o}, \vec{d}, \vec{x}, \vec{x} + l)$
14: $\vec{c} \leftarrow \text{EMISSION}(n) \cdot (u_{max} - u_{min})$
15: $t \leftarrow u_{max}$
16: $f \leftarrow \text{EXIT\_FACE}(\vec{o}, \vec{d}, \vec{x}, \vec{x} + l)$
17: $n \leftarrow \text{CHILD}(n, f)$
18: **end while**
19: **return** $\vec{c}$

4 Evaluation

In order to get an overview of the performance of the program and the different ray traversal algorithms, we perform a set of empirical experiments. Tests were executed on a machine with two Nvidia Titan X (Pascal) GPUs, each with 12 GB of video memory, and two 20-core Intel Xeon Silver 4114 CPUs with a total of 64 GB of system memory.

To ensure an accurate evaluation, we render each dataset from a number of different camera angles designed to simulate user input. The camera first rotates a half circle around the volume, zooms in while rotating vertically around the volume, and finally performs a complete rotation while inside the model. Each of these steps is divided in 50 camera angles, which form a continuous motion, for a total of 150 frames. Results are gathered from a single experiment for each dataset, though for each different camera angle the volume is rendered 4 times consecutively, for a total of 600 frames per experiment.

Measurements are retrieved by querying the GPU timestamp before and after a ray traversal shader program is dispatched, through a Vulkan *timestamp query*. Furthermore, we also measure the total amount of CPU time an experiment takes. From these time differences we can then gather statistics such as the amount rays cast per second per frame and the average framerate (frames per second) of an experiment. All datasets are rendered in full HD (1920 by 1080 pixels, for a total of 2 073 600 pixels and 1 244 160 000 rays) on only a single GPU. The experiments are performed using the headless backend, where images are discarded after they are rendered.

We also perform a similar benchmark on GVDB by Hoetzlein [Hoe16]: a CUDA-based volumetric rendering- and computation library using ray tracing. GVDB offers several different traversal strategies of which the most comparable one, deep traversal, is benchmarked. The same machine and camera translations are used. Volumes are first converted from TIFF to the VDB format [Mus13], which can then be loaded by GVDB. Although GVDB originally used an absorption-emission optical model, we tweaked the renderer's parameters until a sufficiently similar resulting image was obtained. Note that GVDB renders the model in an axis system which is flipped in the X direction. For this reason the camera transforms where also modified, but as a result the images are rendered mirrored.

4.1 Test data

Four base datasets were used to evaluate performance:

- A CT-Scan of the well-known stanford bunny by Stanford University Computer Graphics Laboratory. The scan consists of $512 \times 361 \times 512$ voxels.
- A CT-Scan of an Amazon False Fer-de-lance snake (Xenodon Severus), scanned by the University of Michigan Museum of Zoology (UMMZ) as part of the openVertebrate project[7], and hosted on MorphoSource[8] [Boy+16]. The original volume is $1941 \times 1207 \times 1977$ voxels, but was resampled to $768 \times 478 \times 768$ voxels.

[7] `https://www.floridamuseum.ufl.edu/science/overt/`

[8] `https://www.morphosource.org//Detail/MediaDetail/Show/media_id/31014`

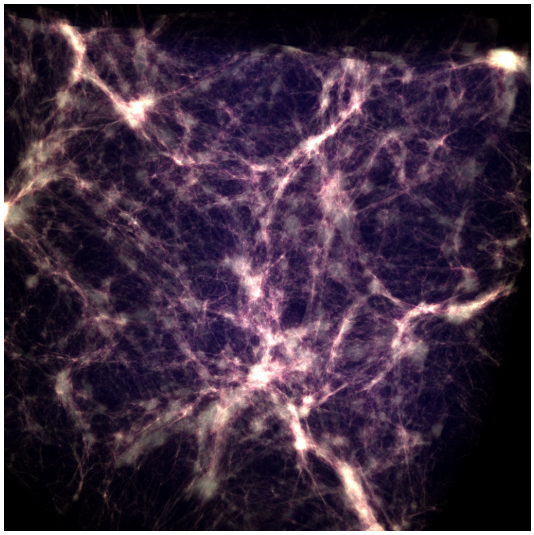
Dataset	Dimensions	Type	Heuristic	Nodes	Memory consumption (MB)	Memory consumption ratio
Bunny	$512 \times 362 \times 512$	Grid			378	1
	512^3	DAG		1 500 800	60	0.158
	512^3	Rope		19 678 369	787	2.074
	512^3	DAG	**cd** 50	287 925	11.5	0.030
	512^3	Rope	**cd** 50	2 472 385	98.9	0.261
	$480 \times 368 \times 448$	GVDB			326	0.86
Snake	$768 \times 478 \times 768$	Grid			1 128	1
	1024^3	DAG		32 221 028	1 289	1.143
	1024^3	Rope		290 470 521	11 619	9.015
	1024^3	DAG	**cd** 70	1 217 339	48.7	0.043
	1024^3	Rope	**cd** 70	9 911 921	396	0.352
	$768 \times 480 \times 768$	GVDB			2 176	1.930
TNG300-3	1024^3	Grid			4 294	1
		DAG		25 099 576	1 004	0.234
		Rope		524 709 457	20 988	4.886
		DAG	**sd** 40	15 302 459	612	0.143
		Rope	**sd** 40	199 476 673	7 979	1.858
		GVDB			7 224	1.806
TNG100-2	2048^3	Grid			34 360	1
		DAG		69 945 197	2 798	0.081
		Rope		1 554 751 473	62 190	1.810
		DAG	**sd** 40	35 557 026	1 422	0.041
		Rope	**sd** 40	575 723 705	23 028	0.670
		GVDB			61 303	1.916

Table 1: Datasets of volumes and their properties. Memory consumptions marked in red exceed 12 GB, and thus these volumes cannot be tested.

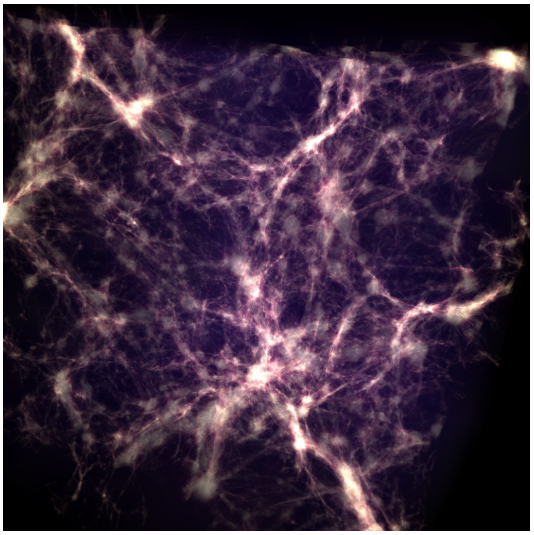
- A volume synthesized from over 224 million gas cells of snapshot 99 of the TNG300-3 cosmological galaxy formation simulation [Pil+17]. The particle coordinates are processed to a volume of 1024^3 voxels, and are then normalized and shaded.
- A similar volume of the larger TNG100-2 simulation. 2048^3 voxels are constructed from almost 700 million gass cells from snapshot 98.

See Appendix A for example renderings of the volumes.

The latter dataset is of particular interest because of its size: as a uniform grid with 32 bits of RGBA emission data per voxel, this dataset takes up 32 GiB. This amount would fill up the entire virtual memory of the highest end GPUs, let alone the 12 GB of the GPUs at our disposal.



(a) No heuristic, 69 980 842 nodes, 2 799 MB



(b) **sd** 40, 37 824 57 nodes, 1 513 MB

Figure 10: Renderings of trees of the TNG100-2 volume. Even though the pruned tree (right) is almost half the size of the unpruned tree (left), there is at first sight almost no difference between the obtained images.

Through the construction and compression methods described in Section 3.2.4, we obtain the datasets displayed in Table 1. Differing dimensions of the bunny and snake datasets can be explained by the dimension requirements of datastructures and the characteristics of the volume data. For the tree-based data structures, a cube with side dimensions equal to a power of two is required. As for the GVDB volumes, a bounding box of the voxels is calculated during construction, which due to empty regions or size constraints similar to those of the octree-based datastructures might end up with different dimensions than those of the associated uniform grid.

The usage of the maximum difference heuristic is denoted by **cd** and a parameter, and the usage of the standard deviation heuristic is denoted by **sd** and a parameter. Both parameters take the form of an 8-bit channel value in range 0 up to and including 255. Volumes are related to traversal algorithms in the following fashion: the DDA algorithm outlined in Section 3.2.3 is rendered with volumes of type **grid**. The naive algorithm outlined in Section 3.2.5, the depth-first algorithm discussed in Section 3.2.6 and the efficient sparse voxel octrees adoption outlined in Section 3.2.7 are rendered with volumes of type **DAG** (obtained through transforming an octree to a directed acyclic graph). The rope tree traversal algorithm outlined in Section 3.2.8 is rendered with type **rope**, which are sparse voxel octrees with ropes. Note that the size of a rope tree is equal to the size of an octree that is not transformed into a directed acyclic graph. Finally, volumes of type **GVDB** are rendered with GVDB.

Split heuristic parameters were manually selected by comparing different combinations of parameters and heuristics until a good compression ratio was obtained with minimal quality loss. For the bunny volume the maximum difference heuristic proved to be most effective as the standard deviation heuristic introduced artefacts into the volume, even at low parameters. The snake volume follows a similar story: the standard deviation heuristic causes visually significant subtrees to be pruned,

Experiment	Rays per second (millions)		Framerate
	Average	Std. Dev	
DDA	338.31	101.69	134.25
Naive	125.35	54.65	47.36
Depth-First	30.22	6.30	13.91
ESVO	122.60	28.14	54.07
Rope	133.44	86.71	43.32
Naive (**cd** 50)	371.05	179.83	116.51
Depth-First (**cd** 50)	133.01	42.60	55.80
ESVO (**cd** 50)	434.74	162.55	158.53
Rope (**cd** 50)	752.95	166.80	281.25
GVDB	7.09	2.98	2.84

(a) Summary obtained from the bunny volume.

Experiment	Rays per second (millions)		Framerate
	Average	Std. Dev	
DDA	159.98	28.27	71.99
Naive	13.26	3.16	6.04
Depth-First	2.60	1.31	1.00
ESVO	18.89	3.80	8.76
Rope		Took too long	
Naive (**cd** 70)	143.97	55.49	55.46
Depth-First (**cd** 70)	42.74	6.47	19.94
ESVO (**cd** 70)	153.90	30.63	68.51
Rope (**cd** 70)	203.47	94.87	72.89
GVDB	1.73	0.77	0.67

(b) Summary obtained from the snake volume.

Experiment	Rays per second (millions)		Framerate
	Average	Std. Dev	
DDA	59.90	28.70	19.50
Naive	4.79	1.42	2.10
Depth-First	1.15	0.42	0.49
ESVO	5.54	2.70	2.16
Rope		Too large	
Naive (**sd** 40)	13.44	4.95	5.62
Depth-First (**sd** 40)	3.31	1.30	1.38
ESVO (**sd** 40)	13.86	6.84	5.36
Rope (**sd** 40)		Took too long	
GVDB		Crashed	

(c) Summary obtained from the TNG300 volume.

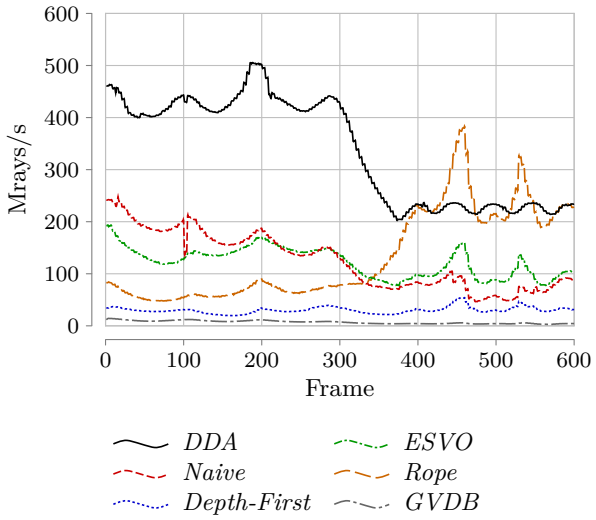
Experiment	Rays per second (millions)		Framerate
	Average	Std. Dev	
DDA		Too large	
Naive	2.33	0.49	1.06
Depth-First	0.67	0.16	0.30
ESVO	2.61	0.65	1.18
Rope		Too large	
Naive (**sd** 40)	11.07	3.54	4.87
Depth-First (**sd** 40)	3.18	1.10	1.39
ESVO (**sd** 40)	12.47	4.53	5.37
Rope (**sd** 40)		Too large	
GVDB		Too large	

(d) Summary obtained from the TNG100 volume.

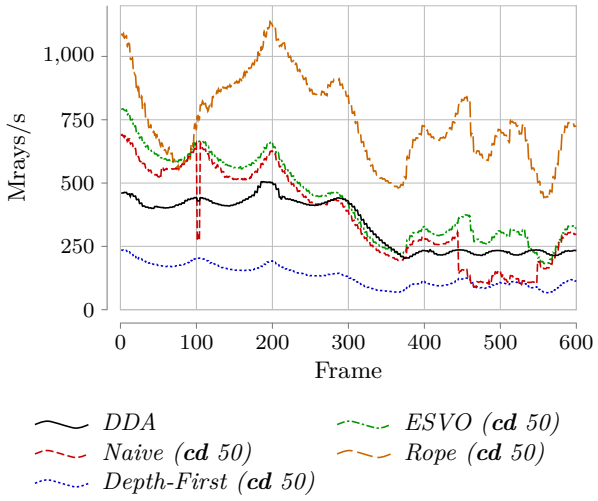
Table 2: Summary of gathered results.

leading to artefacts. The maximum difference heuristic, in contrary, provides quite good quality conservation even at higher parameters. In the case of the TNG300-3 and the TNG100-2 volumes, both split heuristics proved effective at removing visual insignificant features from the original volume. Figure 10 shows a comparison of an unpruned tree and a pruned tree: at first sight, there is almost no difference between the two images. However, one might notice that in the image of the pruned tree small, insignificant galaxy filaments are lacking. Since the primary features of the volume are preserved and the size of the volume is signficantly smaller, we believe that this is an acceptable sactifice to make. See Appendix B for more renderings of trees created with different split heuristics and parameters.

Memory consumption is determined by the total amount of GPU memory the volume uses. In the case of GVDB, this is calculated by the sum of the sizes of the allocated textures for the bricks and the topology. Table 1 also displays the ratio of memory consumption of a volume to the memory consumption of the uniform grid. Striking is how the memory consumption of some trees, in particular the rope trees which cannot be transformed into a directed acyclic graph, is higher than the memory consumption of their associated uniform grids. In these cases, the complexity of the source volume causes the memory overhead of the tree structure to outweigh the gained benefit by pruning, even with high split heuristic parameters. The tree can obviously be pruned more, however, a significant loss of quality must be considered when doing so.



(a) Rendering results of the unpruned trees, grid, and GVDB.



(b) Rendering results of the pruned trees and the grid.

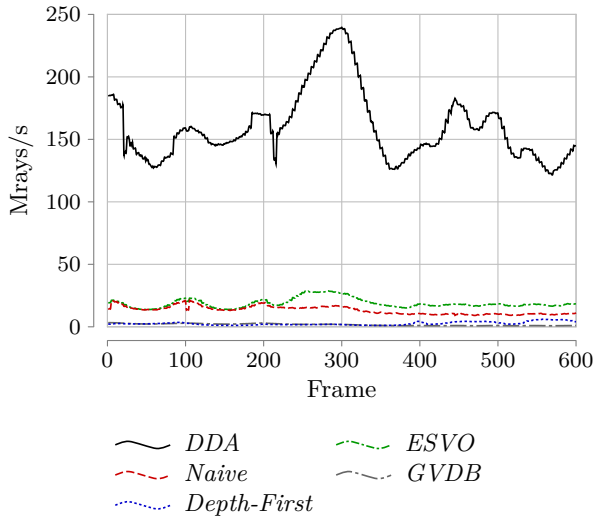
Figure 11: Per-frame render performance of the bunny volume.

4.2 Results

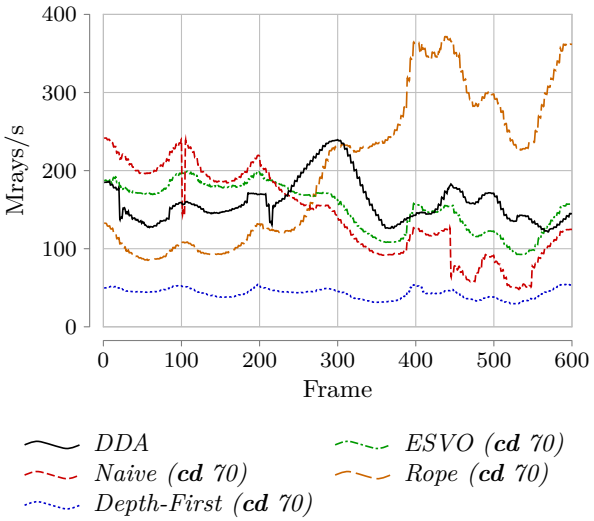
Table 2 gives a summary of the average number of rays per second, the standard deviation over the number of rays per second as calculated over all 600 frames of an experiment, and the average framerate (number of frames per second) of an experiment. The framerate is calculated from the total CPU time of the experiment, including overhead of the Vulkan API calls, while the number of rays traversed per second is calculated from *just* the time actually spent traversing rays. Figures 11 through 14 give the ray traversal performance in millions of rays traversed per second for all 600 frames of the experiments, in the order of the previously described path of camera angles.

Several experiments failed to be completed. In addition to the ones of which the dataset exceeds the size of the test GPU's memory capacity (outlined in red in Table 1), experiments Rope of the snake volume, and Rope (**sd** 40) and GVDB of the TNG300-3 volume failed to render. The root cause of the former problem was traced to be a violation of the maximum size of a buffer bound to a shader program's input variables, causing undefined behaviour in the traversal algorithms. While the value of this limitation is filled in by GPU driver, its field is according to version 1.1.117 of the Vulkan specification limited to 32 bits (allowing a maximum of 4 GB). The GVDB experiment crashed due to a memory access violation in GPU memory when the volume was rendered from a certain camera angle. We believe this to be a bug in the implementation of GVDB, however, a root cause was not identified.

From Figures 11 through 13 we can see that in the unpruned case, the DDA algorithm generally attains a higher amount of rays per second than the tree-based traversal algorithms, with a large difference in performance in the case of the snake and TNG300 volumes. This can be explained by the relevant simplicty of the DDA traversal algorithm, and by the fact that more specialized GPU

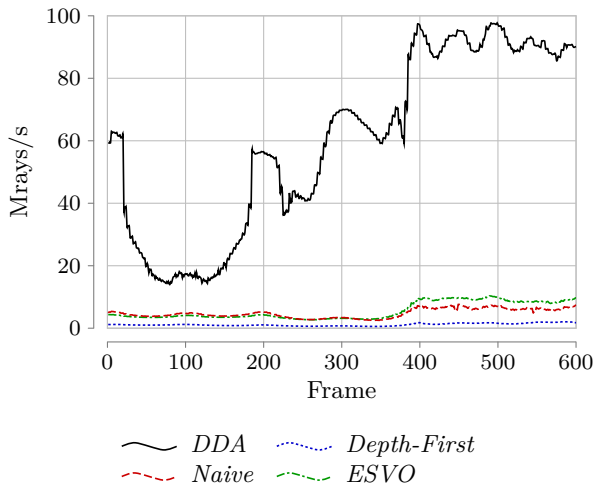


(a) Rendering results of the unpruned trees, grid, and GVDB.

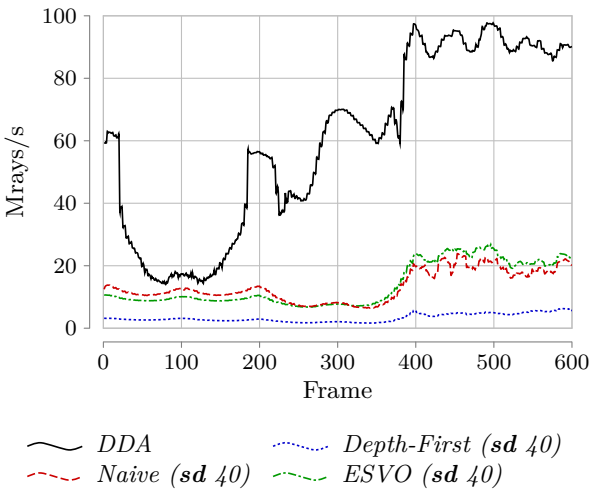


(b) Rendering results of the pruned trees and the grid.

Figure 12: Per-frame render performance of the snake volume.

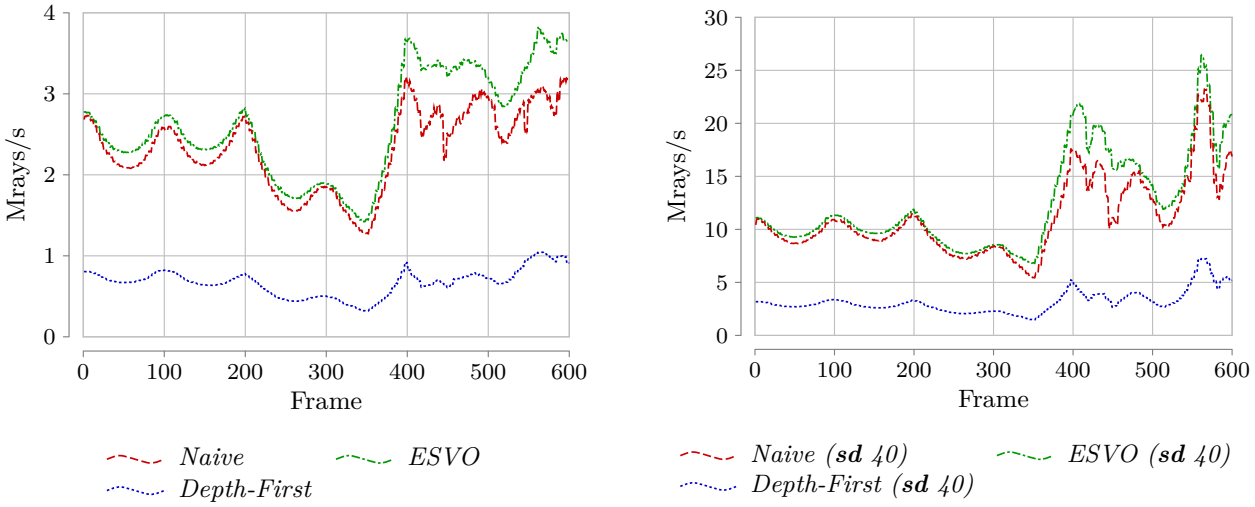


(a) Rendering results of the unpruned trees and the grid.



(b) Rendering results of the pruned trees and the grid.

Figure 13: Per-frame render performance of the tng300 volume



(a) Rendering results of the unpruned trees.

(b) Rendering results of the pruned trees.

Figure 14: Per-frame render performance of the tng100 volume

hardware can be used than with the sparse voxel octree traversal algorithms.

It is immediately clear that in all cases, traversal of the pruned trees outperforms that of the unpruned trees. This is to be expected but provides proof that pruning does not only lead to a more compact representation of the volume. Traversal performance of the pruned trees also tells a different story when compared to the DDA algorithm: whereas the DDA algorithm outperforms every other algorithm for every tree in the unpruned case, it can only just compete with the naive, *efficient sparse voxel octrees* and rope tree traversal algorithms when rendering the bunny (Figure 11b) and snake (Figure 12b) volumes. In the case of the TNG300-3 volume (Figure 13b) however, the DDA algorithm still outperforms the traversal of the pruned trees quite significantly.

An interesting trend that shows itself in almost all experiments is that the ray traversal performance changes drastically after around 300 frames. This coincides with when the camera is zoomed into the volume, and although when the camera is inside the volume the rays have to be traversed less far, more of the render area is covered by the volume. Different algorithms respond differently for different datasets: in Figure 11a it can be seen how the DDA algorithm's ray traversal performance is halved after the camera has moved into the volume, but the performance of the rope tree traversal of the pruned snake volume (Figure 12b) triples. We argue that this phenomenon can be explained by characteristics of the dataset: for a dataset with an higher complexity, when the camera is inside the volume the amount of time that is saved by having to traverse less far through the volume outweighs the time that is saved by not having to traverse rays that miss the volume when the camera is outside it. This depends on the distance of the camera to the volume of course.

When we compare just the sparse voxel octree based ray traversal algorithms, we can see that all four algorithms have similar performance characteristics: especially prominent in Figures 11b

and 14b, we can see that each algorithm has performance bumps in similar spots in each experiment, typically about every 100 frames. This coincides with when the camera's viewing direction is exactly parallel with the x-, y- or z-axis. Similar performance characteristics may be explained by the fact that each algorithm traverses similar nodes, but different performance is attained due to the different strategies applied when doing so. Secondly, we find that the depth-first approach discussed in Section 3.2.6 is in all experiments much slower than the others. This can be explained by the fact that, while the algorithm is simple on paper, many different conditions must be considered: Is the stack empty? Does the ray intersect a child? Is this child the last? etc. This may yield code that is inefficient for a GPU due to its high amount of branches. Furthermore, we see that *efficient sparse voxel octrees* performs quite consistently just a small amount better than the naive algorithm, even though the naive algorithm is in theory much less efficient. We again believe that this can be explained by the nature of GPUs: While the *efficient sparse voxel octrees* algorithm needs a few branches to implements its logic, the entire naive algorithm can be implemented with only two simple loops and two conditions in total. Finally, we observe from the results of the bunny volume (Figures 11a and Figures 11b) and the snake volume (Figure 12b) that the rope tree traversal algorithm in these cases offers competitive performance when compared to the DDA, naive and *efficient sparse voxel octrees* algorithms, although performance is somewhat worse when the camera is outside the volume. A drawback of this method is the memory requirement however, which is why it could only be evaluated in three out of eight experiments.

Similar to the rope tree traversal method, GVDB is often limited by its memory requirements. In the cases where it can be evaluated however (see Figures 11a and 12a), it is clear that it is outperformed by most of our implemented traversal algorithms, except the depth-first traversal algorithm with the snake volume.

When no API overhead is considered, the amount of rays per second is linearly dependent on the framerate: 30 frames per second, which is about the minimum amount required for real-time rendering, is equal to about 62.2 Mray/s on the resolution used in our experiments. If we examine the total amount of frames per second in the summary of the experiments given in Table 2, we see that this amount is only achieved on the bunny volume, and certain cases of the snake volume. From the graphs showed in Figures 11 through 14 we can see that the ray traversal performance widely varies over the course of the experiments. This means that from certain points of view a real-time performance can be achieved, while from others not at all, which might yield a very stuttery and unpleasant user experience.

5 Conclusion

In this thesis, we describe an implementation of volumetric ray tracing software using Vulkan. This software provides three different output backends which may be selected: X.org, Direct and Headless. Five different ray traversal algorithms are implemented or adopted, one based on a uniform grid and four based on a sparse voxel octree acceleration data structure.

An evaluation of the different ray traversal algorithm shows that the DDA algorithm is, in general, a very efficient approach due to the simplicity of the traversal algorithm and the ability to utilize certain specialize GPU hardware functionality. If a sparse voxel octree is sufficiently pruned with a lossy split heuristic however, a tradeoff between storage size, image quality, and render performance can be achieved. This allows tree-based ray traversal algorithms to attain higher efficiency, sometimes even outperforming the DDA algorithm. Another benefit of storing volumes as trees is the ability to render volumes which would otherwise exhaust GPU resources. Of the four different sparse voxel octree based ray traversal algorithms implemented, the *efficient sparse voxel octrees* and naive implementation appear to be the most competitive. The depth-first approach suffers from a lower performance than the others due to the relative high complexity of the algorithm. Finally, when the rope tree traversal algorithm is not limited by its memory requirements, it can offer a higher performance than the other algorithms.

Based on our experiments we conclude that for volumetric datasets of realistic proportions, real-time performance may be achievable. If a sparse voxel octree becomes too complex however, or a grid volume becomes too large, framerates will drop too low for the user experience to be interactive. A performance comparison with similar ray tracing software, CUDA-based GVDB, shows a large performance difference between it and the traversal algorithms implemented in our work.

From a programming point of perspective, we found Vulkan to have a high learning curve at first, but as time went on it became more logical and straight forward to work with. Limitations such as the one discussed in Section 4.2 are easy to miss, however, the Vulkan validation layers can help with this in some regard. Its wide array of extensions allowed addition of features such as being able to select different output backends or multi-gpu support by duplicating resources across different devices, with relative ease.

5.1 Future Work

In the following sections some methods are outlined which may improve render- and storage efficiency.

5.1.1 N-ary Voxel Trees

We could obtain a more efficient acceleration data structure by combining the storage efficiency of sparse voxel octrees with the traversal efficiency of a uniform grid. One method is to store a brick of N^3 voxels in a leaf node, as implemented in Gigavoxels [Cra+09]. The internal nodes of the tree can be traversed by a usual sparse voxel octree traversal algorithm, while the leaf nodes are traversed by a DDA algorithm.

Another method is to also store internal nodes as bricks of N^3 instead of 2^3 as with an octree. VDB [Mus13] and by extension GVDB [Hoe16] store the tree in a <5,4,3> format for exampe: Leaf bricks have $(2^3)^3$ voxels, the level higher has $(2^4)^3$ children per node and finally the highest level of nodes has $(2^5)^3$ children. The root node is a uniform grid of pointers to nodes of the highest level. The DDA algorithm can again be used to traverse the nodes of every level.

5.1.2 Naive Multi-GPU Rendering

Although the implementation presented in this research impelements naive multi-GPU rendering capabilities, where resources are simply duplicated over all configured graphics devices, no efforts where made to properly test the relative performance of it. Due to the way ray tracing can be parallelized however, we have reason to expect an almost linear increase in performance. Rays cast for one output do not interact with rays cast for another, after all.

5.1.3 Dividing the Volume

Note that the optical model outlined in Section 3.2.2 does not require the volume to be traversed in any particular order. That means the volume can be divided over multiple GPUs, which independently render their part of the volume, and the individual results can be combined to obtain the final image. The advantage of this method when compared to naive multi-GPU rendering, where the volume is simply sent to all GPUs and each renders a part of the final image, is that it allows for a much larger volume to be rendered. The volume may be as large as all the GPU memory sizes combined, instead of just the memory size of the GPU with the smallest amount of memory.

A drawback of this method however would be the last step, when the final image is put together. As each GPU renders to an image that represents the total view area, which need to be communicated to the right GPU to form the final image(s). GPUs typically are not optimized for downloading the amount of data required for this communication from GPU memory to CPU memory: an empirical evaluation yielded 53 MiB/s on our Nvidia Titan X (Pascal) GPU, which is equal to about 9 1080p 24-bit RGB images. At a minimal framerate of 30 FPS this is obviously not feasible. An alternative would be to utilize a technology such as NVLink, which connects the GPUs directly. NVLink offers a considerably higher inter-GPU connection speed, however, as such a connection was not installed on our GPUs at the time of writing we were unable to evaluate it.

A Volumes

Volumes presented in this appendix are rendered by Xenodon.

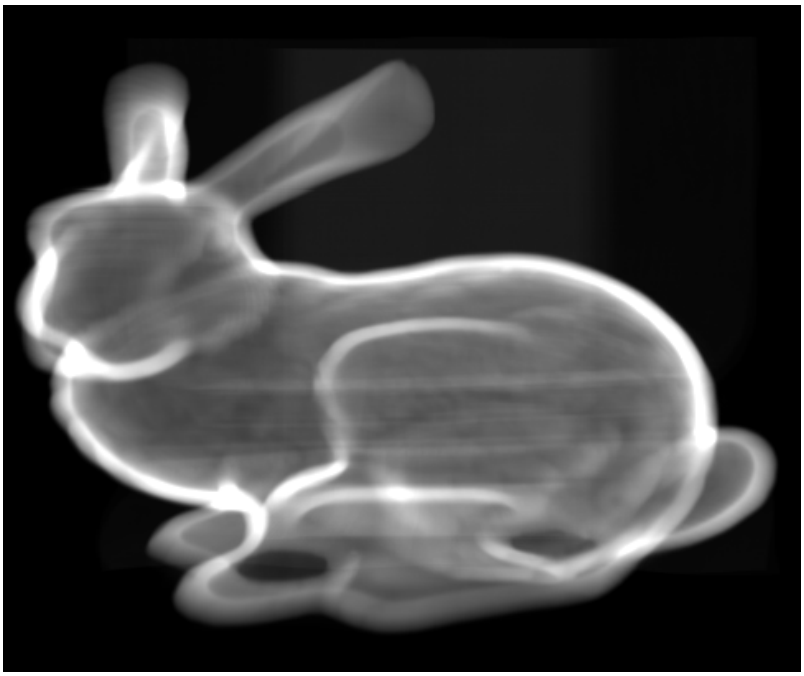


Figure 15: The bunny volume. Dataset courtesy of the Stanford University Computer Graphics Laboratory.

Figure 16: The snake volume. Dataset courtesy of the University of Michigan Museum of Zoology.

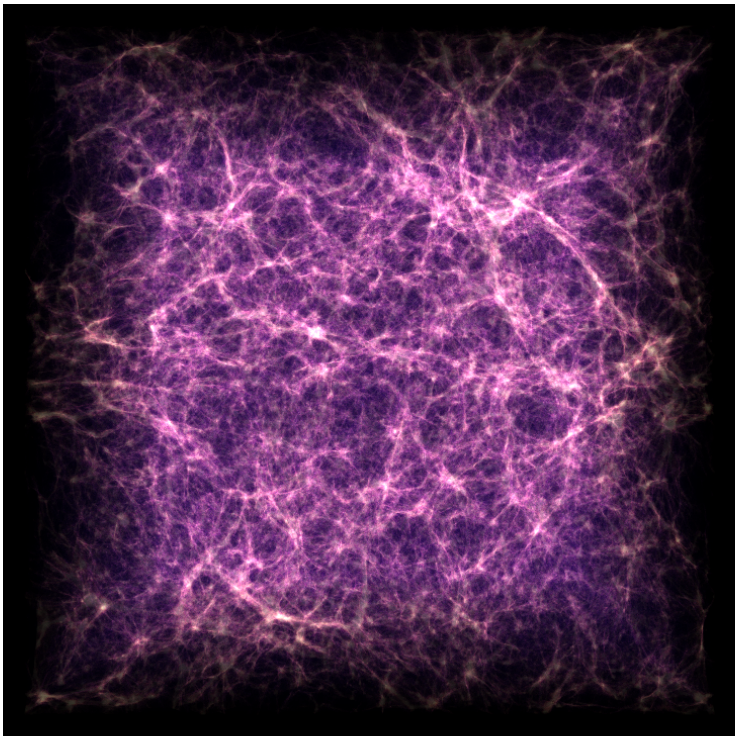


Figure 17: The TNG300-3 volume. Dataset courtesy of the TNG project, `http://tng-project.org`.

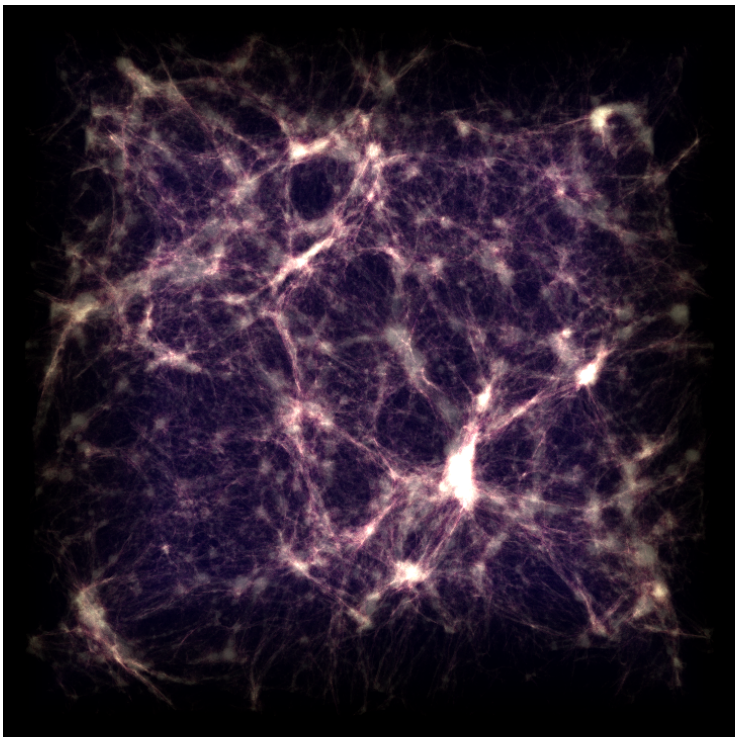
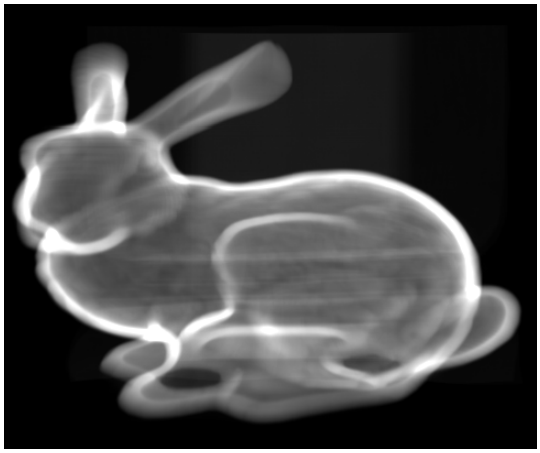


Figure 18: The TNG100-2 volume. Dataset courtesy of the TNG project, `http://tng-project.org`.

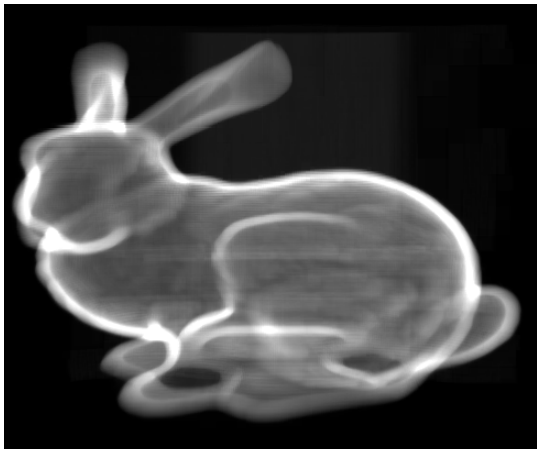
B Lossy Split Heuristic Effects



(a) No heuristic, 1 500 800 nodes, 60 MB

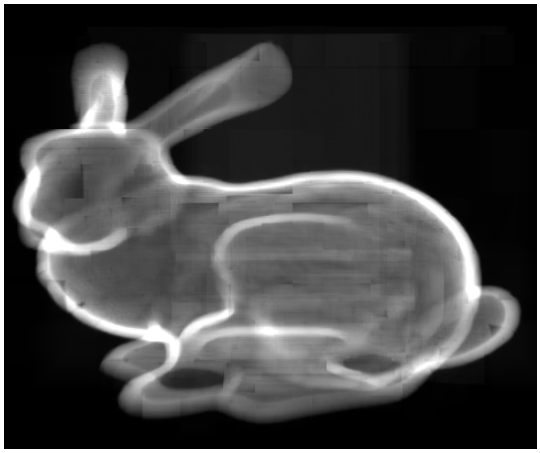
(b) **cd** 50, 287 925 nodes, 11.5 MB

(c) **cd** 60, 217 594 nodes, 8.7 MB



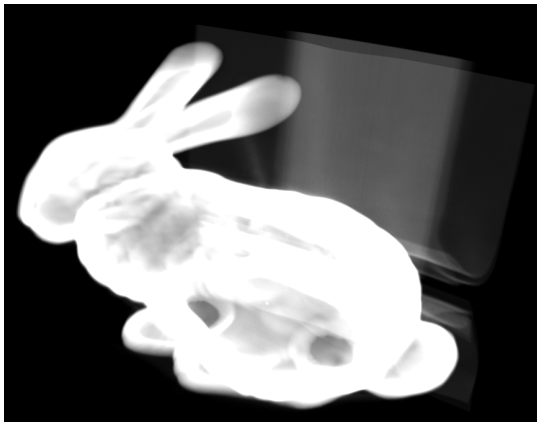
(d) **cd** 70, 155 110 nodes, 6.2 MB

(e) **sd** 20, 417 816 nodes, 16.7 MB

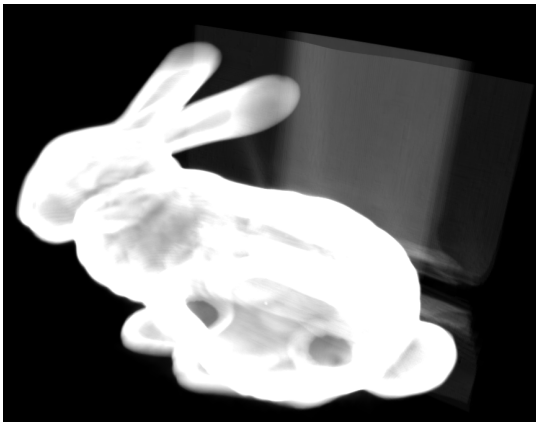


(f) **sd** 30, 235 644 nodes, 9.4 MB

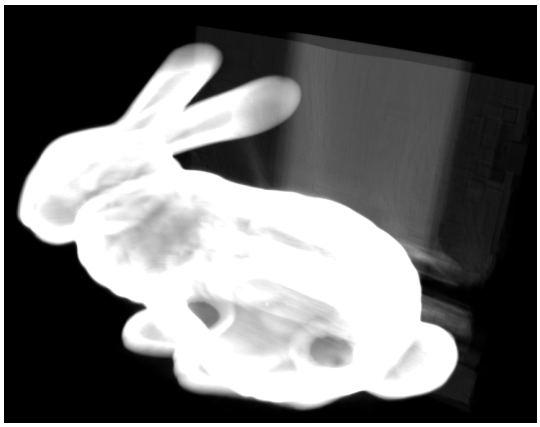
Figure 19: Selection of renderings of trees of the bunny volume created with different heuristics and parameters.



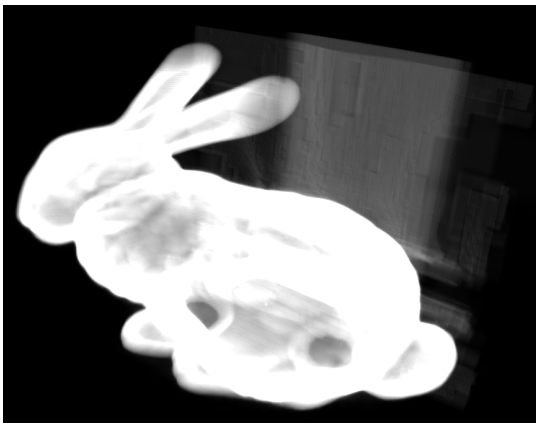
(a) No heuristic, 1 500 800 nodes, 60 MB



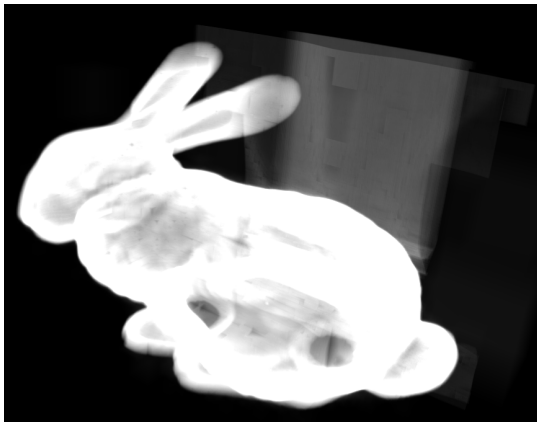
(b) **cd** 50, 287 925 nodes, 11.5 MB



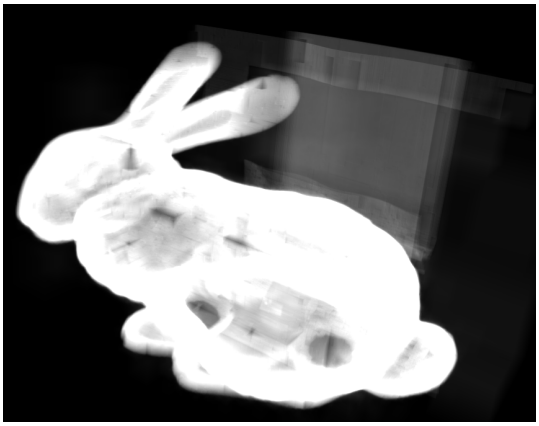
(c) **cd** 60, 217 594 nodes, 8.7 MB



(d) **cd** 70, 155 110 nodes, 6.2 MB



(e) **sd** 20, 417 816 nodes, 16.7 MB



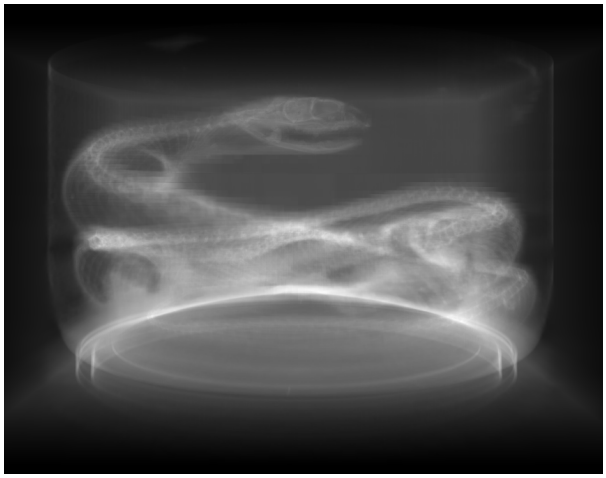
(f) **sd** 30, 235 644 nodes, 9.4 MB

Figure 20: Renderings highlighting the deterioriation of the back side of the bunny volume with different heuristics and parameters.

(a) No heuristic, 32 221 028 nodes, 1 289 MB

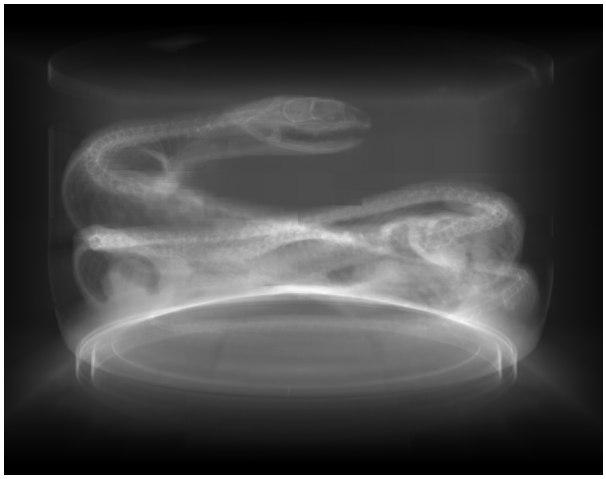
(b) **cd** 50, 2 468 999 nodes, 98.8 MB

(c) **cd** 70, 1 217 339 nodes, 48.7 MB



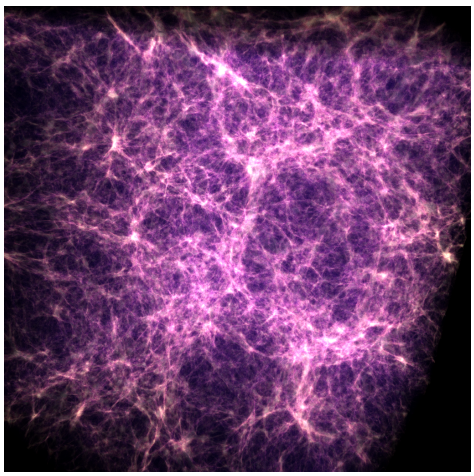
(d) **cd** 90, 652 363 nodes, 26.1 MB

(e) **sd** 20, 3 011 680 nodes, 120 MB

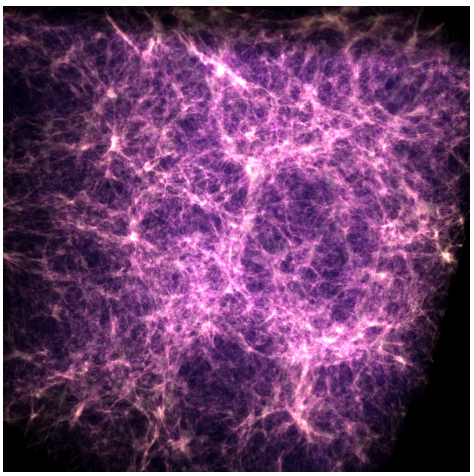


(f) **sd** 30, 1 315 38 nodes, 52.6 MB

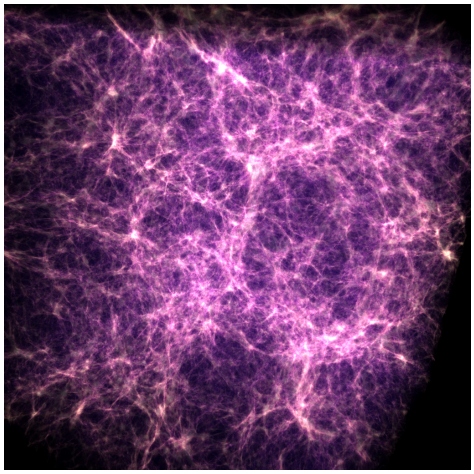
Figure 21: Renderings of trees of the snake volume created with different heuristics and parameters.



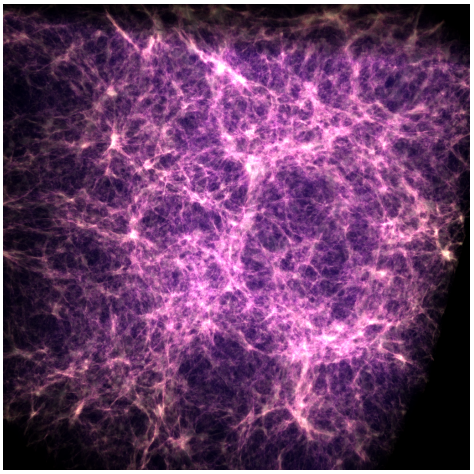
(a) No heuristic, 25 099 576 nodes, 1 004 MB



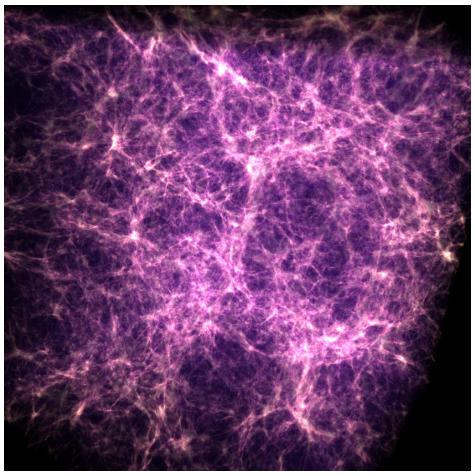
(b) **cd** 50, 23 292 819 nodes, 932 MB



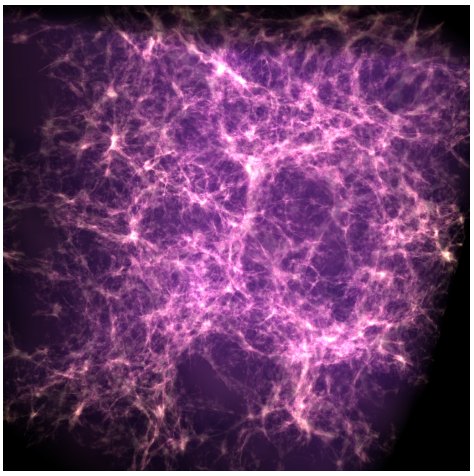
(c) **cd** 100, 21 442 080 nodes, 858 MB



(d) **cd** 150, 15 782 722 nodes, 631 MB

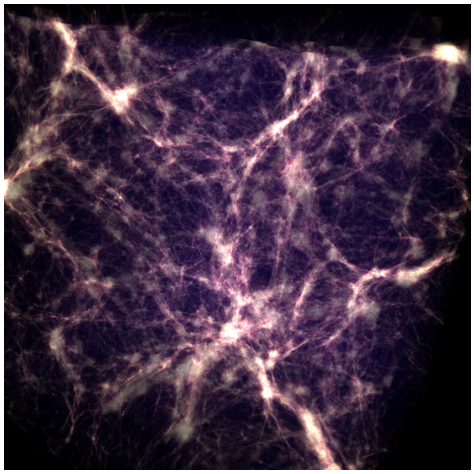


(e) **sd** 40, 15 302 459 nodes, 612 MB

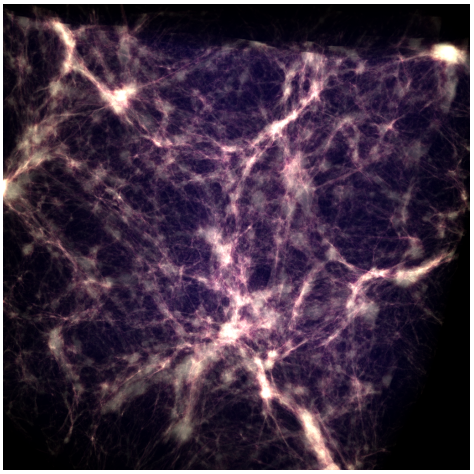


(f) **sd** 60, 8 352 560 nodes, 334 MB

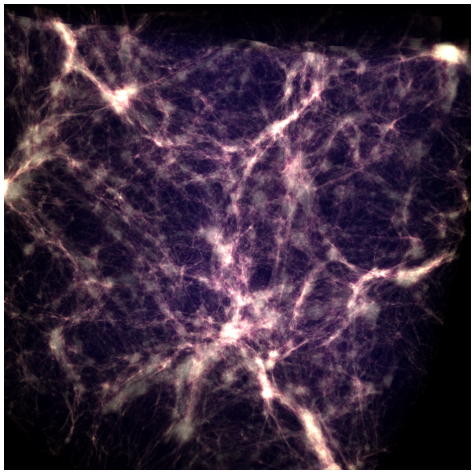
Figure 22: Renderings of trees of the TNG300-3 volume created with different heuristics and parameters. Note that while most of the images look the same, very minescule differences exists.



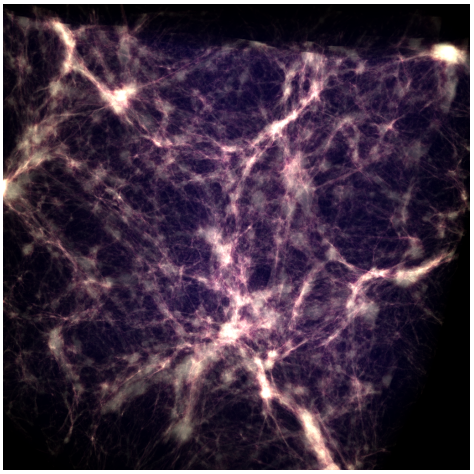
(a) No heuristic, 69 945 197 nodes, 2 798 MB



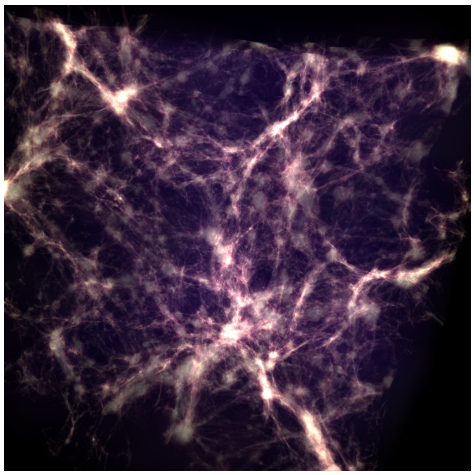
(b) **cd** 50, 66 293 307 nodes, 2 652 MB



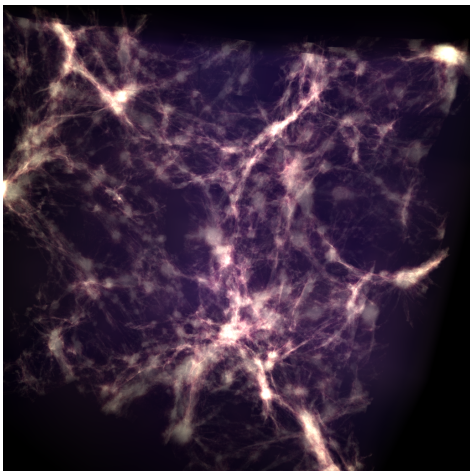
(c) **cd** 100, 63 113 324 nodes, 2 525 MB



(d) **cd** 150, 52 622 087 nodes, 2 104 MB



(e) **sd** 40, 35 557 026 nodes, 1 422 MB



(f) **sd** 50, 24 785 793 nodes, 991 MB

Figure 23: Renderings of trees of the TNG100-2 volume created with different heuristics and parameters. Similar to the TNG300-3 volume, most of the images look the same, but are slightly different where the splitting heuristic caused insignificant subtrees to be pruned.

Bibliography

[LC87] William E Lorensen and Harvey E Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM siggraph computer graphics*. Vol. 21. 4. ACM. 1987, pp. 163–169.

[Gla84] Andrew S Glassner. "Space subdivision for fast ray tracing". In: *IEEE Computer Graphics and applications* 4.10 (1984), pp. 15–24.

[AW+87] John Amanatides, Andrew Woo, et al. "A fast voxel traversal algorithm for ray tracing". In: *Eurographics*. Vol. 87. 3. 1987, pp. 3–10.

[LK10] Samuli Laine and Tero Karras. "Efficient sparse voxel octrees". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2010), pp. 1048–1059.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "High resolution sparse voxel DAGs". In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 101.

[Mar06] Henry Markram. "The blue brain project". In: *Nature Reviews Neuroscience* 7.2 (2006), p. 153.

[Par+10] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. "OptiX: a general purpose ray tracing engine". In: *Acm transactions on graphics (tog)*. Vol. 29. 4. ACM. 2010, p. 66.

[Wal+16] Ingo Wald, Gregory P Johnson, Jefferson Amstutz, Carson Brownlee, Aaron Knoll, Jim Jeffers, Johannes Günther, and Paul Navrátil. "Ospray-a cpu ray tracing framework for scientific visualization". In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 931–940.

[Hoe16] Rama Karl Hoetzlein. "GVDB: raytracing sparse voxel database structures on the GPU". In: *Proceedings of High Performance Graphics*. Eurographics Association. 2016, pp. 109–117.

[Mus13] Ken Museth. "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM Transactions on Graphics (TOG)* 32.3 (2013), p. 27.

[Max95] Nelson Max. "Optical models for direct volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108.

[Gla89] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[HBZ98] Vlastimil Havran, Jirí Bittner, and Jirí Zára. "Ray tracing with rope trees". In: *14th Spring Conference on Computer Graphics*. 1998, pp. 130–140.

[Boy+16] Doug M Boyer, Gregg F Gunnell, Seth Kaufman, and Timothy M McGeary. "Morphosource: Archiving and sharing 3-d digital specimen data". In: *The Paleontological Society Papers* 22 (2016), pp. 157–181.

[Pil+17] Annalisa Pillepich, Volker Springel, Dylan Nelson, Shy Genel, Jill Naiman, Rüdiger Pakmor, Lars Hernquist, Paul Torrey, Mark Vogelsberger, Rainer Weinberger, et al. "Simulating galaxy formation with the IllustrisTNG model". In: *Monthly Notices of the Royal Astronomical Society* 473.3 (2017), pp. 4077–4106.

[Cra+09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM. 2009, pp. 15–22.