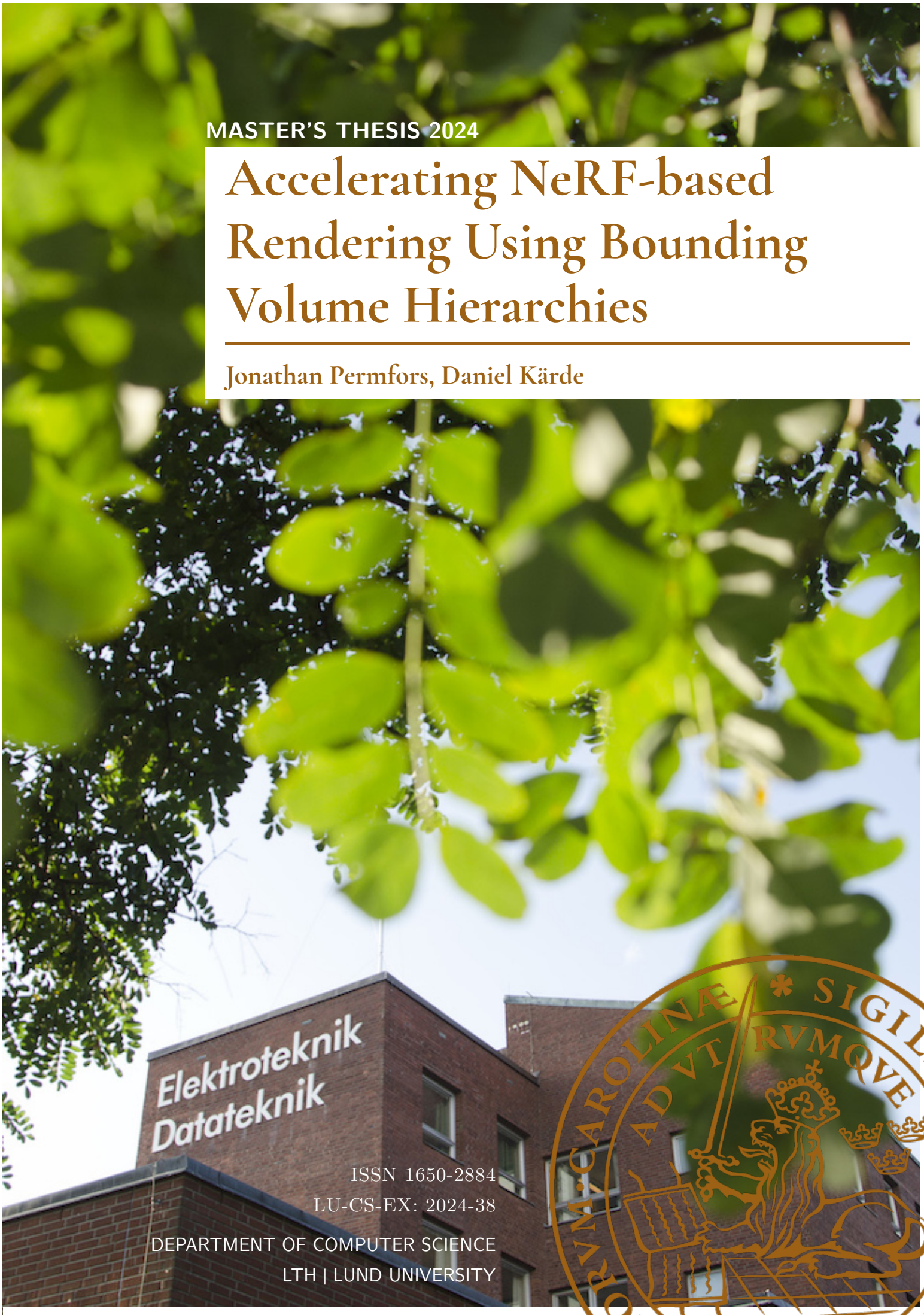


MASTER'S THESIS 2024

Accelerating NeRF-based Rendering Using Bounding Volume Hierarchies

Jonathan Permfors, Daniel Kärde

ISSN 1650-2884
LU-CS-EX: 2024-38

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-38

Accelerating NeRF-based Rendering Using Bounding Volume Hierarchies

Acceleration av NeRF-baserad rendering genom användning av BVH

Jonathan Permfors, Daniel Kärde

Accelerating NeRF-based Rendering Using Bounding Volume Hierarchies

Jonathan Permfors
permforsjonathan@gmail.com

Daniel Kärde
daniel.karde@gmail.com

June 25, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Rikard Olajos, rikard.olajos@cs.lth.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

Originally introduced in 2020, Neural Radiance Fields (NeRFs) have become a great area of interest within the research community. NeRFs combine neural networks with volumetric rendering in order to synthesize new images of 3D scenes by training on a sparse set of 2D images. The neural network describes the scene as a continuous field and as such must be sampled from in order to render the final image. The computational cost of inference on these samples is very high, which incentivizes reducing the number of samples to a minimum while maintaining image quality. We explore a method of accelerating the rendering of NeRFs through the usage of Bounding Volume Hierarchies that are constructed from point clouds exported from a pre-trained model. From the point clouds, we utilize K-Means to group the points into clusters, from which we instantiate our bottom-level bounding boxes. We experiment with varying point cloud sizes and cluster counts, along with different tree construction methods. By modifying a CUDA-based ray tracer we are able to efficiently compute intersection points which we use to place samples where the density exceeds some threshold value. The reduction in samples results in a significant decrease in rendering times while maintaining the overall image quality, however, some artifacts become visible in close-ups of specific scenes. We limit our work to that of bounded scenes and note that there are many interesting areas for future research.

Keywords: Graphics, Neural Radiance Fields, Bounding Volume Hierarchies, Ray Marching

Acknowledgements

We would like to thank our supervisor Rikard Olajos for his guidance throughout this thesis project. His knowledge and support were truly invaluable to us. We also want to express gratitude to Michael Doggett for the amazing graphics courses, which inspired us to pursue a thesis in computer graphics. Finally, we thank Ellen for helping us turn our initial sketches into polished graphics.

Contents

Chapter 1
Introduction

This thesis explores the field of *Neural Radiance Fields* (NeRFs). NeRFs use neural networks (NN) in combination with volumetric rendering in order to achieve realistic rendering of 3D scenes, having trained the network only on a set of 2D images. The field is rapidly evolving; initially proposed by Mildenhall et al. [11], recent developments have achieved impressive improvements in training speeds, rendering times, as well as quality [5]. Despite this, there is still lots of room for improvement. Rendering performance is crucial for many applications, such as gaming and virtual reality. The rendering of NeRFs requires taking many samples, often millions per image, each requiring a forward pass through the neural network. While the network architecture is fairly simple for a neural network, millions of samples become very computationally demanding. Reducing the number of samples taken is therefore a crucial task. We propose using a *Bounding Volume Hierarchy*, or BVH for short, in order to constrict the sampling space, thereby improving rendering times.

1.1 Research Questions

The purpose of this thesis is to answer the following research questions:

- What impact does the usage of BVHs have on the rendering times and image quality of NeRF-rendered scenes.
- How can BVHs be constructed to optimize NeRF rendering times while retaining image quality.

1.2 Project Scope

NeRFs can be used to reconstruct two different types of scenes, bounded scenes and unbounded scenes. Bounded scenes are constricted to sample only a limited region of space both during training and rendering, ensuring that all learned information is captured in this region. Many real-world scenes are more accurately described as unbounded scenes, which are difficult to confine. Such scenes require more adaptive methods of sampling in order to accurately capture all relevant scene information while still maintaining computational efficiency. The scope of this thesis is limited to bounded 3D scenes. Furthermore, it only focuses on the rendering of such scenes, not the training of the networks.

1.3 Contribution to the State of Knowledge

Our contributions to the state of knowledge can be summarized as follows:

- A methodology for constructing Bounding Volume Hierarchies from pre-trained NeRFs by exporting point clouds and clustering the points using K-Means.
- Demonstrated the possibility of integrating BVH-based ray tracing in the Nerfstudio framework.
- Provided a demonstration of the effects on rendering performance and image quality using a wide variety of point cloud sizes and cluster counts across diverse datasets.
- A comparison of the rendering performance of different split heuristics used during BVH construction.
- Achieved a significant speedup in NeRF rendering performance while maintaining image quality on most datasets. On average the FPS is increased ~ 7X using Nerfacto and ~ 16X using Vanilla-Nerf.

Chapter 2
Background and Theory

This chapter covers the background and theory that this project is based on. It will cover what NeRFs are, and explain volumetric rendering, ray marching, and bounding volume hierarchies. Furthermore, a section about previous work will describe other efforts that have been made to reduce the amount of samples needed.

2.1 Neural Radiance Fields

Neural Radiance Fields (NeRFs), originally introduced by Mildenhall et al. [11], are deep neural networks, which create a continuous 5D representation of the scene. The neural network takes as input a 5D vector, composed of a 3D position (x, y, z) and a viewing direction (θ, ϕ), and outputs a color in RGB along with a density. NeRFs utilizes a positional encoding $f : \mathbb{R}^3 \rightarrow \mathbb{R}^n$, where $n \gg 3$, that maps the position and viewing direction into a higher dimensional space before passing it through the neural network, allowing it to learn more high-frequency components of the scene.

First, the positional encoding of the 3D position passes through 8 fully-connected layers of 256 units each, all using a ReLU activation function, which retains positive outputs and makes negative outputs zero. This positional encoding is also concatenated to the fifth layer's activation output, which is often referred to as a skip connection or shortcut connection. Such layers are beneficial for deep neural networks in that they mitigate the vanishing gradient problem [6], and prevent information from being lost in the earlier layers. Following these 8 layers is another layer that outputs the density at the position, and a 256-dimensional vector. This vector is concatenated with the positional encoding of the viewing direction, before being passed to a final layer of 128 units, which outputs the RGB value. It should be emphasized that this structure ensures that the density is only a function of the 3D position, while the color also depends on the viewing direction. Figure 2.1 shows the model architecture.

The final trained neural network then becomes a continuous function that describes the

color and density of any point in the 3D scene, viewed from a particular viewing angle.

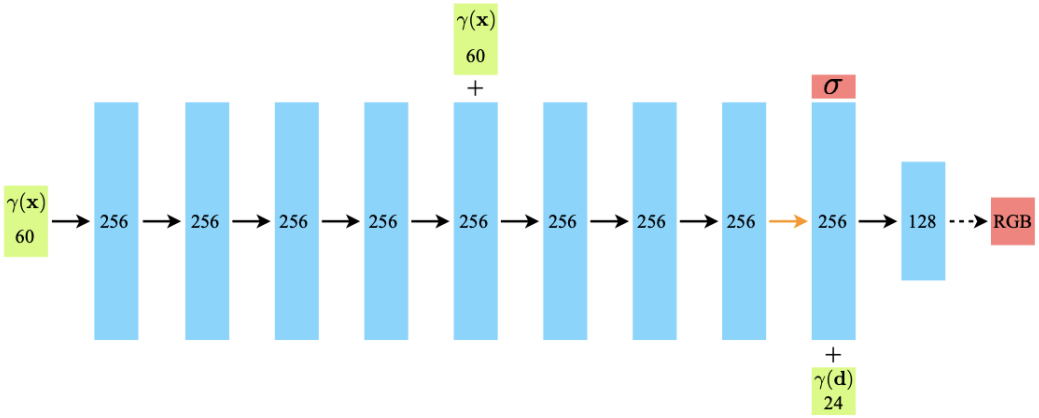


Figure 2.1: Visualization of the NeRF architecture. The green boxes signify inputs, while the red boxes denote outputs. The blue boxes are the layers of the neural network. Black solid arrows denote the usage of a ReLU activation function, orange arrows no activation, and dashed arrows a sigmoid activation function. The + denotes vector concatenation. Source: Mildenhall et al. [11].

In order to render the final image, NeRFs rely on the principles of volumetric rendering, which is distinctly different from traditional, surface-based rendering. In surface-based rendering, 3D objects are represented as surfaces, commonly using triangle meshes. These triangle meshes contain all the necessary information to render the final image. In volumetric rendering, on the other hand, there is a continuous field representing the scene or object. As such, the field must be discretely sampled. This is done by shooting a ray through each pixel, sampling the scene along the ray, and then accumulating the sampled values in order to compute the RGB value of the pixel. This process is known as *ray marching*. In principle, this means estimating the following integral:

$$C(r) = \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t), d)\, dt \tag{2.1}$$

where $r(t)$ is a point at distance t from the origin along the ray, $C(r)$ the accumulated color of the ray, $\sigma(r(t))$ the density at the point, $c(r(t), d)$ the emitted color at the point in the direction, d, of the ray and $T(t)$ the transmittance from t_n to t. t_n and t_f represent the near and far bounds of the scene.

This integral can be estimated using N sampled points as follows:

$$\hat{C}(r) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i(t_{i+1} - t_i)))c_i \tag{2.2}$$

where t_i is the distance along the ray of sample i.

The original sampler proposed by Mildenhall et al. [11] is referred to as a hierarchical sampler. It entails training two networks concurrently, one designated as “coarse” and one

"fine". The coarse network is first evaluated by querying 64 samples to the neural network. Using the output from this network, each sample is assigned a weight that describes how much the sample contributes to the final color. After the samples are normalized, they are used to create a piecewise constant probability density function (PDF) along the ray, indicating where to sample further. By sampling from the PDF, we get an additional 128 samples. These samples are then combined with the initial 64 samples, which are then queried to the fine network. The outputs of the fine network are then used to render the final image.

One of the main issues with the original NeRF implementation is the slow rendering times, which can be several seconds per frame. The high computational cost stems from the fact that each sample gets passed through a deep fully-connected neural network. Another contributing factor is the high dimensionality of the input to the NN, as the 5D samples are mapped into a 60-dimensional space, which also puts a high demand on the GPU memory. In order to achieve high-quality renders, a total of 192 samples are passed through the fine network for each pixel in the image, in addition to the 64 samples passed through the course network. For an image rendered in a 1920×1080 resolution, this results in $(192+64)\times 1920\times 1080 = 530,841,600$ forward passes. The fact that each ray gets assigned a fixed number of samples, although it might only pass through a very short section of density, or even none at all, presents a potential area of optimization.

2.2 Improvements to NeRFs

Since the publication of the original NeRF paper, the field has been rapidly evolving. Many papers have been published introducing unique methods of optimization, often combining numerous techniques to improve training times, rendering times, and image quality. This section will briefly present a selection of them.

2.2.1 Camera Pose Refinement

Training a NeRF requires a set of 2D images along with their respective camera poses, which is defined by a camera-to-world matrix consisting of a rotation and a translation. These camera poses are estimated and as such may contain errors, which may result in a loss of quality and artifacts. A method that mitigates this error is that of camera pose refinement — where the camera pose parameters are treated as trainable parameters [18]. During the training of the NeRF, the loss gradients are computed and backpropagated jointly with the NeRF parameters.

2.2.2 Per Image Appearance Conditioning

The images used for training a NeRF may contain variations caused by weather, lighting, and camera exposure. Martin-Brualla et al. [10] describes a method that attempts to model such variations using what are referred to as "appearance embeddings". By capturing such variations in a trained appearance embedding for each input image instead of capturing them in the NeRF model itself, renders may become more realistic, especially using real-world training images.

2.2.3 Proposal Sampling

The proposal sampler was first introduced by Barron et al. [3]. Alongside the NeRF, an additional neural network is trained, the proposal network. The proposal network is a much smaller MLP compared to the NeRF and is not trained with the images the NeRF is trained on. Rather, it is trained on the weight distribution generated by the NeRF and only predicts density without RGB. Initially, the samples are placed uniformly along the ray, which are then passed through the proposal network, which outputs a weight vector that describes the density distribution along the ray. Samples are then taken from this distribution, which can again be passed to the proposal network to further refine the distribution. This is done for a fixed number of iterations and allows for the initial uniform samples to be condensed down to a much smaller number of samples.

2.2.4 Scene Contraction

Unbounded scenes present a challenge for NeRFs as samples need to be allocated efficiently to both near and distant objects. This becomes computationally expensive if distant objects use the same sampling density as near objects. Scene contractions efficiently deal with this issue by contracting the scene into a bounding box. This can be done in a number of ways [3, 16], however, the key concept is that distant objects are contracted more aggressively as opposed to near objects. This essentially means that fewer samples are allocated to distant objects, as they need less precision to be rendered.

2.3 Bounding Volume Hierarchies

A *bounding volume hierarchy* (BVH) is a hierarchical structure consisting of *bounding volumes* [1]. A bounding volume is a volume encapsulated by some simple geometry, for example, a box or a sphere. The idea is that the simple geometry is easier to perform calculations on than what it encloses, resulting in a much higher performance of a program. Such calculations can be, amongst other things, intersection tests, which is what this project will be using it for. If a ray does not intersect with the bounding box, there is no need to perform intersection tests with the objects it encapsulates. The usage of BVHs can therefore drastically reduce the number of intersection tests that need to be performed, which is at the root of the performance gains.

The hierarchy is constructed as a tree consisting of nodes. The top node, or *root node*, has children, which in turn have children of their own, all leading down to the leaf nodes at the bottom of the tree. The leaf nodes themselves are what contain the actual geometry which we are interested in. An illustration of this can be found in Figure 2.2, where a simple 2D scene consisting of three objects has been divided into a BVH. There are many ways of constructing this tree, some methods being real-time, while others can take a considerable amount of time. However, using a hierarchical structure with bounding volumes can potentially let you skip some, or in special cases even close to all, calculations that need to be performed each frame, which often makes the time spent constructing it worthwhile. In general, the usage of a hierarchical structure brings the complexity from $O(n)$ to $O(\log n)$, as it then becomes a binary search problem.

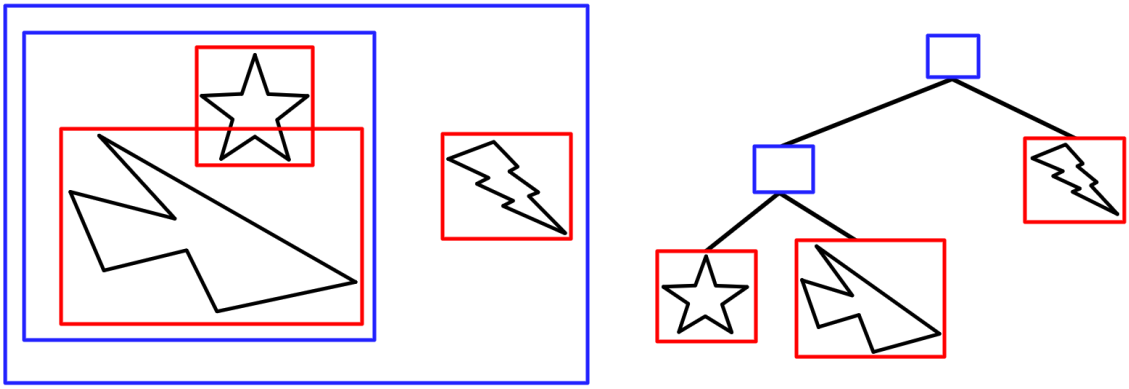


Figure 2.2: Example of a scene divided into a bounding volume hierarchy structure. The scene is to the left while the tree structure of this can be seen to the right.

While there are many ways of creating the bounding volumes that surround the geometry and make out the nodes in the hierarchy tree, we have decided on using *axis-aligned bounding boxes*. When the geometry has been encapsulated by these bounding volumes, they have to be split up into subsets in some way to create the tree structure. Determining how to perform this split most effectively, while having it be as fast to traverse as possible, is an entire research topic itself, but the two common methods we will be using are *surface area heuristics* (SAH) and *median split*. The following sections will describe these three concepts in more depth.

2.3.1 Axis-Aligned Bounding Boxes

Axis-aligned bounding boxes, or AABBs, are boxes that can be defined as having normals on each face that coincide with the basis axes of the scene, which means that each box will be oriented the same way no matter where in the scene it is. Every AABB can be described by two points, the maximum and minimum x, y, and z coordinates [1]. A picture showing this can be found in Figure 2.3.

In the context of this project, AABBs will be used to define where in the scene density can be found. The idea is that intersection tests with the AABBs are computationally cheap compared to a forward pass through the neural networks, and therefore having the placement of density predetermined by these boxes results in fewer unnecessary samples being placed such as the samples to the coarse network described in Section 2.1.

To utilize the AABBs, intersection tests are performed which determine where in 3D space each ray intersects the enveloping density, and effectively where the sampling should start and end. There exist different methods to perform ray-AABB intersection tests, a popular one being the “Slab” method introduced by Kay and Kajiya [8].

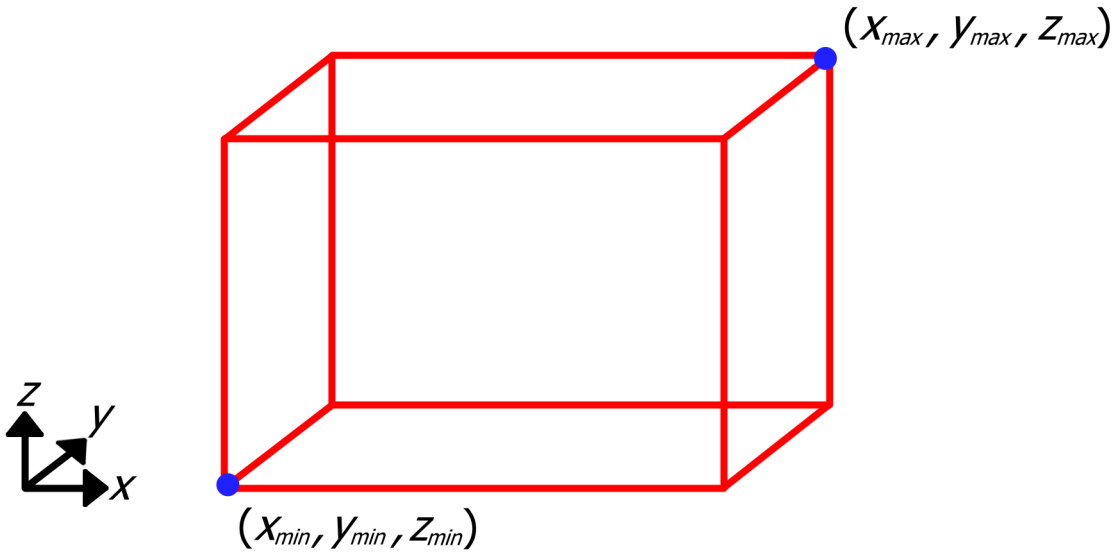


Figure 2.3: Example of an AABB, with the maximum and minimum coordinate points on it marked as blue dots. Figure based on an image from Akenine-Moller et al. [1].

2.3.2 Median Split

The median split partitioning method is a fairly simple method that involves two stages [4]. In the first stage, it is determined which axis (x, y, z) to split the primitives on. This can be done in numerous ways, for instance by selecting the axis with the greatest extent, as follows:

$$k = \arg\max_i(\mathbf{max}_i - \mathbf{min}_i) \tag{2.3}$$

where $\mathbf{max}_i$ and $\mathbf{min}_i$ are the maximum and minimum values along axis i, respectively. Once the partitioning axis has been determined, it needs to be determined where along the axis the splitting is to be done. First, the centroids of the objects that are to be partitioned are computed along the partitioning axis. The splitting position is then simply determined as the median value of these centroids. As such, the objects will be evenly distributed amongst the two splits, resulting in a balanced tree. Determining the median can be done in $O(log\ n)$ by sorting the points which result in a fairly fast construction time in the context of this thesis. An example of the resulting split when using median split can be found in Figure 2.4.

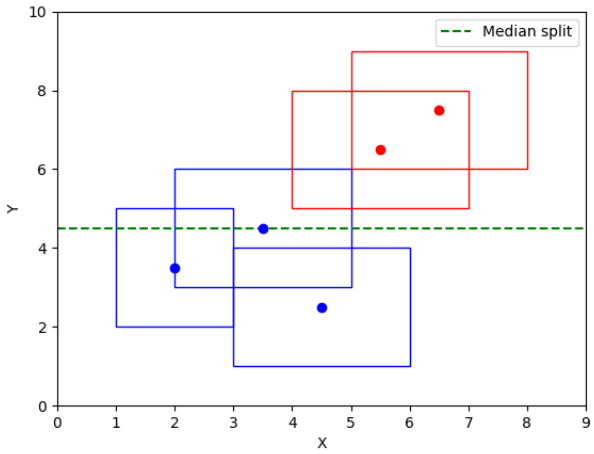


Figure 2.4: Example of the resulting partitioning using a median split. In this case, the Y-axis has the greatest extent and is therefore the partitioning axis.

2.3.3 Surface Area Heuristics

Surface area heuristics is a method in which the cost of performing a certain split is approximated to determine the best division of nodes. The cost of the split is calculated as the sum of the left and right AABBs surface area times the amount of nodes inside the AABB, respectively [7, 9]. The cost at each split is therefore computed as follows:

$$C_{SAH} = N_{left} \cdot A_{left} + N_{right} \cdot A_{right} \tag{2.4}$$

where C_{SAH} is the cost of the split, N_{left} and N_{right} is the number of leaf nodes in the left and right split respectively and A_{left} and A_{right} is the surface area of the AABBs respectively. The surface area of the AABBs can be calculated as:

$$A = 2 \cdot (x \cdot y + x \cdot z + y \cdot z) \tag{2.5}$$

where x, y, and z are the absolute values of the difference between the max and min values of the x, y, and z coordinates respectively. To determine what split is the best one to use, an exhaustive search is done where for each axis, the midpoint of every leaf node is used as a suggested split point. Pseudocode for the SAH method can be found as Algorithm 1.

Algorithm 1 SAH split method

```
bestAxis ← init
bestPos ← init
bestCost ← MaxInt
for all axes do
    for all AABBs do
        candidatePos ← aabb.centroid
        cost ← EvaluateSAHCost(node, axis, candidatePos)
        if cost < bestCost then
            bestPos ← candidatePos
            bestAxis ← axis
            bestCost ← cost
        end if
    end for
end for
axis ← bestAxis
splitPos ← bestPos
```

2.4 Related Work

Wadhwani and Kojima [17] introduce a caching mechanism, effectively reducing the number of samples by reusing previous network outputs. To make the caching feasible in terms of memory requirements, they split the NeRF into two networks, one position-dependent and one direction-dependent. This approach achieves frame rates of 200 FPS on powerful consumer-level GPUs.

Neff et al. [13] are able to reduce the number of samples per ray to only 4, by placing them around the area of the first ray-surface intersection. This is done using a separate network that predicts the sample locations.

Another way of reducing the number of samples produced is by utilizing occupancy grids, as done by Instant-NGP [12]. This method uses a grid that represents the scene and, while training, saves and updates the density value of each grid cell. Another grid is simultaneously updated to hold a single bit value representing if the grid cell is considered occupied or not. The density value that is saved in the first grid gets a threshold which determines the bit value. When sampling is performed, the occupancy bit is checked, and if it is set to low, no sample is placed and the ray marching continues.

Chapter 3
Implementation

In this chapter, we describe the implementation details of our work, such as BVH construction and sampling scheme. The majority of the code is written in Python and is adapted to the Nerfstudio framework, the basics of which will also be introduced. We also explain the adaptations made to the CUDA-based *cubvh* ray-tracer and bring up some challenges encountered during implementation. Furthermore, we explain how we evaluate our implementation, including what datasets we use.

3.1 Nerfstudio

To implement our solution, without having to build a NeRF implementation completely from scratch, we have turned to the tool Nerfstudio. There are a lot of different types of NeRF implementations that have been developed by a variety of researchers and developers. Nerfstudio brought these implementations together into one open-source library, as well as modularize the components of NeRFs. The modularization has made it more user-friendly and facilitated the growth of a community in which people can more easily build on each other's contributions and combine different components. The library consists of many implementations, stretching from the original NeRF model to its own implementation that combines ideas from different articles. The library is well documented and has tutorials on how to get started which makes setting up the environment needed easy.

3.1.1 Methods

We evaluate our sampler on two methods built into Nerfstudio. The first one is called "Vanilla-Nerf", which is an implementation following the NeRF description in Section 2.1. The second one is called "Nerfacto", described by Tancik et al. [16], which implements several improvements from different research papers in order to achieve much better training and

rendering performance, at a slight cost to the image quality. All the improvements mentioned in Section 2.2 are part of the Nerfacto model.

We train one model for each dataset for both Vanilla-Nerf and Nerfacto. During training, we simply use the original sampler, as our sampler only works during inference. We then use these pre-trained models, but replace the sampler with ours, leaving all other components intact.

3.1.2 Nerfstudio Pipeline

The modularization made by Nerfstudio has resulted in a pipeline that consists of two main parts, a DataManager, and a Model [16]. The different parts of the pipeline will be described below, with an overview of it shown in Figure 3.1.

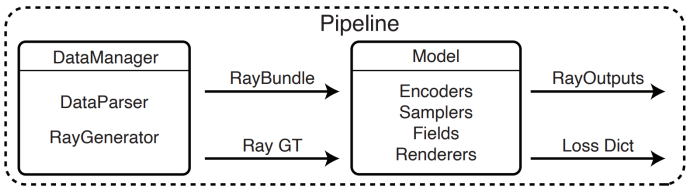


Figure 3.1: The pipeline used by Nerfstudio. Source: Tancik et al. [16].

DataManager

The DataManagers main task is to take data, which in the context of NeRFs is a set of 2D images with accompanying information about the camera's 3D position and angle, and convert this into RayBundles which is a format the Model can understand. The reasoning behind the DataManager is that it should be possible to use many types of programs to produce the input data. Previously most NeRF projects only supported COLMAP, which can be challenging to set up, but with Nerfstudio support for tools such as Polycam, Record3D, and Metashape has been added.

RayBundle and RayGT

A RayBundle is a data structure that discretizes the 3D space the NeRF works in. The content of the RayBundle is in its most basic form Tensors of origins, directions, and pixel areas. This information stored in the RayBundle is later needed in the Model to produce RaySamples in the Samplers. RayGT, which is short for Ray Ground Truth, contains the ground truth values needed during the training of the Model to calculate the losses in the Loss Dict. The RayBundles are used in the forward pass of the Model.

Model

In the pipeline, the Model consists of the parts that make any NeRF unique. They all take RayBundles as input, but depending on the NeRF, the Sampler, Field, Renderer, Encoder, and even the RayOutput will be different. The RayOutput can contain a lot of different values (also called quantities), but usually has at least RGB values and density for each ray

inputted with the RayBundle. In our case we also return the amount of samples for each Ray as well as the sample positions (for debugging purposes).

Sampler

The sampler is the main component we have made changes to. Nerfstudio provides many samplers, some more complex than others. The Vanilla-Nerf model uses a uniform sampler for the coarse network, which simply places the samples at uniform distances along the ray. The fine network uses a PDF sampler, which samples from a distribution defined by the outputs of the coarse network, as described in Section 2.1. Nerfacto instead utilizes the proposal sampler, which is implemented as described in Section 2.2.3.

Field

At the core of every NeRF, there is a neural network. Different NeRFs use different types of neural networks, and these can be defined in the field part of the pipeline. The field will in the most typical case take RaySamples as input, and return the RGB values and density.

3.2 Construction of Bounding Volume Hierarchies

We employ a bottom-up approach to generating the BVH. Leaf nodes are created first, which are then iteratively split into parent nodes using a heuristic function that estimates a good splitting position. The leaf nodes are constructed from the clusters of a point cloud of the scene.

3.2.1 Point Cloud Generation

Bounding Volume Hierarchies are commonly used to organize a discrete set of geometrical objects. In our case, there is no explicit set of objects, but rather a continuous field. It is therefore necessary to discretize this field, which can be done in a number of ways. In our implementation, this is done by exporting a point cloud of the scene. This point cloud is generated by placing random samples in the scene and keeping the ones where the density is above a certain threshold value. Nerfstudio has a built-in method to produce point clouds this way, which is what we use in our implementation.

There are other methods to discretize the field, such as by exporting a mesh of the scene, which is also a built-in functionality in Nerfstudio. Some early exploration of this method did however result in inferior image quality, which we attribute to the mesh not perfectly enveloping all the density (most of which is placed right on the edge of the model). Instead, it focuses on creating a smooth, good-looking, surface, resulting in a loss of information. While simply expanding the mesh in the direction of the normal helped alleviate this issue, it required such a substantial expansion that the performance suffered greatly. We therefore chose to utilize a point cloud, as it seems to more accurately capture the density of the scene.

Figure 3.2 demonstrates an image rendered from a model trained on the Lego dataset. Figure 3.3 visualizes the points exported from the model, using a point cloud size of 3,000,000 points. One can clearly see the structure of the object, with very few outliers.

Figure 3.2: Rendered image of the Lego dataset.

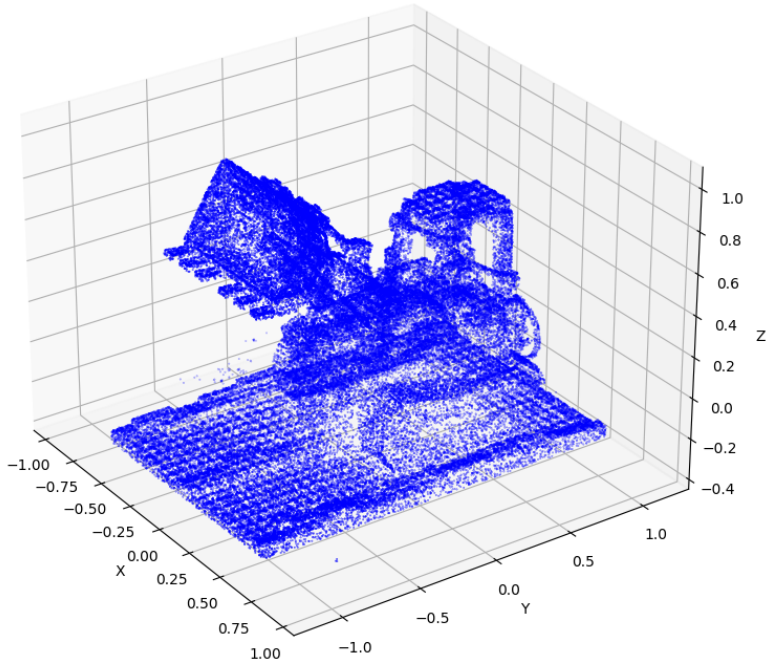


Figure 3.3: Point cloud generated from a Nerfacto model trained on the Lego dataset. The point cloud consists of 100,000 points randomly sampled from a point cloud of 3,000,000 points.

3.2.2 K-Means Clustering

K-Means is an iterative clustering algorithm, popular for its efficiency and simplicity. It is commonly initialized by randomly assigning each data point to a cluster. At each following iteration, the cluster centroid gets recalculated from the newly assigned data points, and each data point is reassigned to the closest cluster centroid based on the Euclidean distance.

We utilize a GPU-accelerated K-Means implementation from the *cuML* project [14]. For each point cloud, the K-Means algorithm is run three times, each time using a different initiation seed. The algorithm then returns the best result as measured by inertia, which is the sum of the squared distances of each point to its assigned cluster centroid.

We explored different implementations of K-Means before deciding on the *cuML* implementation. The reason for this choice relies on two key benefits. Firstly, the GPU-acceleration improves performance greatly for our largest point clouds and as the number of clusters increases, making it possible to explore even larger sizes. Secondly, it is a batched implementation. This is crucial as the GPU-acceleration relies on samples being able to fit into the GPU memory. Using batching, we do not need to fit all samples into memory simultaneously.

3.2.3 Leaf Node Creation

The leaf node AABBs are constructed from the point cloud clusters. From each cluster, the maximum and minimum values along each axis (x, y, z) is computed. The resulting maximum (x, y, z) and minimum (x, y, z) points defines the resulting AABB.

In order to evaluate the constructed leaf nodes, the Euclidean distance between the maximum and minimum points of each AABB is computed:

$$d = \sqrt{(x_{\max} - x_{\min})^2 + (y_{\max} - y_{\min})^2 + (z_{\max} - z_{\min})^2}. \tag{3.1}$$

The largest distance d is then compared to a threshold value that depends on the point cloud size and the number of clusters. If it exceeds the threshold, K-Means is repeated as explained in Section 3.2.2, up to five times. If all attempts exceed the threshold, the result with the smallest value d is used for the tree construction.

In Figure 3.4 the leaf nodes generated from a very small cluster count are displayed, along with the points from which they are generated. The points are plotted using a different color for each cluster. While it shows a significant reduction in the area of interest for sampling, each AABB still contains a fairly large portion of empty space. As the number of clusters grows, they will more closely bind the object, effectively reducing the empty space being captured.

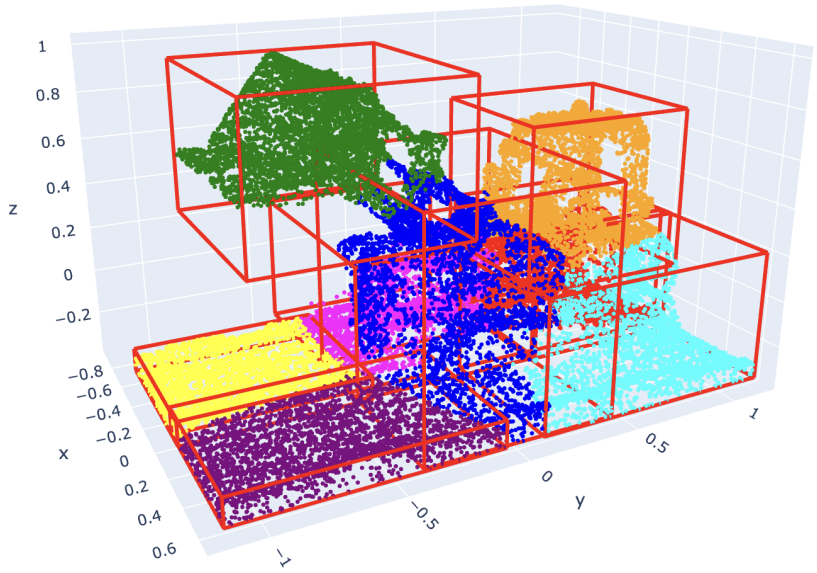


Figure 3.4: Visualization of the clusters generated by K-Means and the resulting leaf nodes using a very small number of clusters on the Lego dataset. The different clusters are assigned different colors.

3.2.4 Tree Construction

From the leaf nodes, the tree is constructed bottoms-up using either median split or SAH, as explained in Sections 2.3.2 and 2.3.3 respectively, depending on which of the methods we evaluated. The resulting tree is a binary tree, where each node is defined by an AABB as well as a pointer to its left and right child.

To facilitate the usage of the BVH in the cubvh project, as described in Section 3.3, the tree is saved as a tensor to a file. The tensor consists of the three x, y, and z coordinates of the min and max values as well as the index in the tensor of the left and right child of each node. To indicate that a node is a leaf node, the index value for the left and right child is set to -1. The resulting tensor has eight values for each node (three each for the min and max coordinates plus two for the children).

3.3 CUDA-based Ray Tracing

The ray tracing is done using a modified CUDA-based ray tracer from the *cubvh* project [2]. The project includes methods for constructing BVHs and for performing ray tracing for batches of rays. In order to gain greater control of the construction, we decided to construct our BVHs in Python using the same tree representation as in cubvh. Furthermore, the cubvh project focuses on ray tracing objects using a mesh representation, which then utilizes ray-triangle intersections. As our scene is not represented by meshes but rather by the bounding boxes themselves combined with the NeRF, it was necessary to modify the ray tracing portion as well. These modifications are explained in greater detail in Sections 3.3.1 and 3.3.2.

3.3.1 Bounding Volume Hierarchy Representation

The original cubvh project had an implemented BVH construction method, but since we wanted to focus on two specific methods, SAH and median split, we decided not to use this. We used our own BVH construction implementation from Python, as explained in Section 3.2, and saved the tree as a tensor to a file. We then imported and converted this into an array which was in turn used to construct a BVH using their internal representation. By doing it this way we made sure the BVH had the intended structure while still being able to utilize the GPU acceleration provided by the cubvh project.

3.3.2 Ray Intersection

The ray intersection implemented in the original project was unfortunately not suited for our needs. While it was very fast, we required intersections with the bounding boxes themselves, not the meshes they were intended to encapsulate. Furthermore, the intersection method only returned the nearest intersection (and could be modified to also return the furthest without too much work), but we needed the intersections with all leaf nodes of the BVH. In order to achieve this, prior to the ray tracing we pre-allocate memory for up to 128 intersections for each ray. This seems to work well in our testing but is admittedly a fairly arbitrary number and should probably be adapted to the size of the BVH. Each intersection is stored as two float32 values, representing the intersection entry and the intersection exit. Therefore, each ray needs 1024 bytes pre-allocated for this purpose. Additionally, we return an integer for each ray storing the number of registered intersections.

3.4 Sampling Scheme

We utilize an adaptive sampling scheme which allows for a different number of samples for each ray being traced. For each ray-AABB intersection, samples are placed uniformly between the intersection entry and exit using a fixed sampling distance of 10^{-3}. The number of samples to place along the ray within a particular AABB is computed as follows:

$$N = \min\left(\left\lceil \frac{t_{\text{exit}} - t_{\text{entry}}}{10^{-3}} \right\rceil, 100\right) \tag{3.2}$$

The first sample is always placed at the intersection entry, and the remaining $N - 1$ samples are placed using the given sampling distance. This sampling strategy presents multiple benefits. Firstly, it allows us to completely skip sampling along rays that do not have any intersections at all. This is quite helpful for scenes composed of objects in "empty space", where this will occur for a notable portion of the rays. Secondly, it allows us to use fewer samples along rays that only pass through a small portion of occupied space.

The sampling strategy also presents some downsides. While it may lead to fewer samples for some rays as compared to a sampling strategy with a fixed sample size, it may also lead to many more, if the ray has a long distance of travel in occupied space. It also makes memory usage unpredictable, as rays are processed in batches where the number of samples in each batch may vary greatly. Finally, the approach will lead to extra samples being taken in areas where AABBs overlap. This is demonstrated in Figure 3.5.

It should be emphasized that the sample placement is done per ray-AABB intersection, independent of one another, and an intersected AABB will always get at least one sample, as can be seen in Equation 3.2. As the number of clusters grows and—correspondingly—the cluster sizes decrease, it may lead to a situation where the clusters are so tightly packed such that the sampling distance will effectively decrease if the distances between AABBs become smaller than the sampling distance.

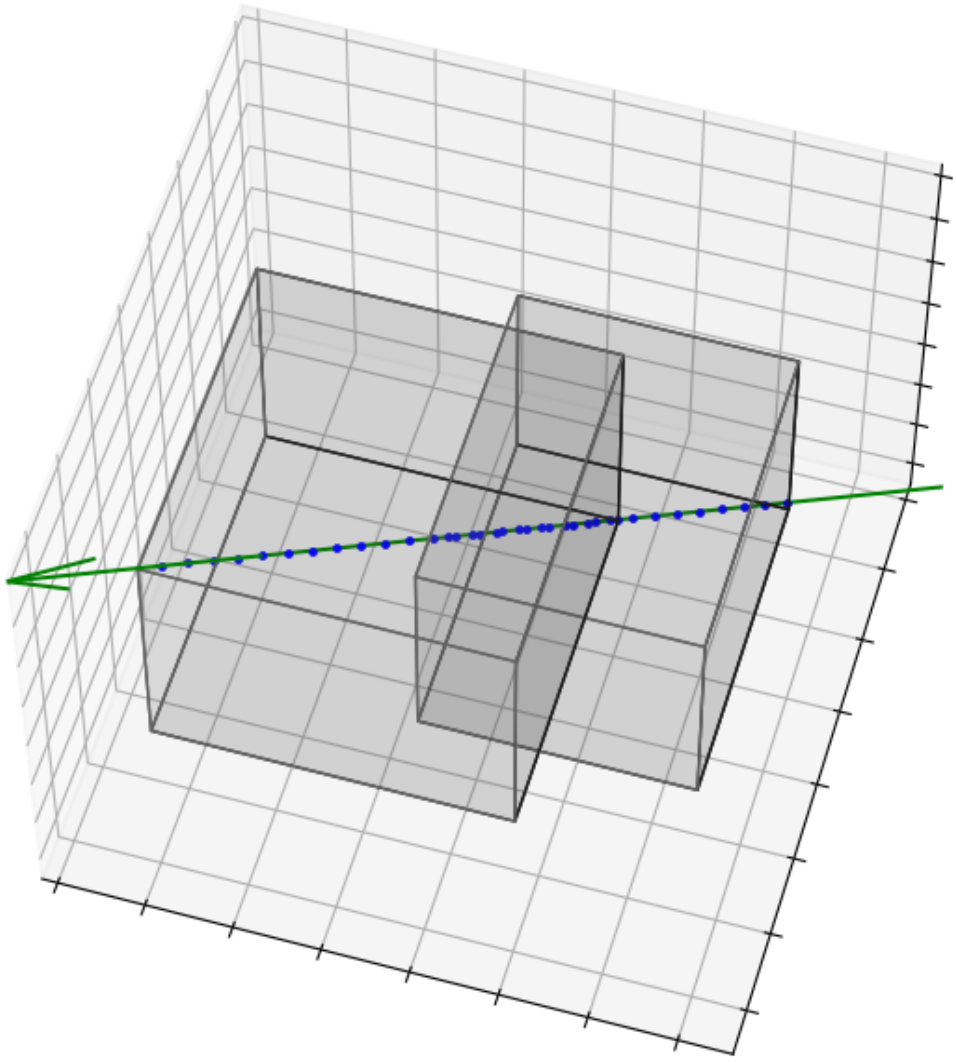


Figure 3.5: Sample positions along a ray intersecting two slightly overlapping AABBs. The overlap is exaggerated to demonstrate that sampling is done per AABB and ray, which may lead to extra sampling in areas where AABBs overlap. For the larger cluster counts, there are rarely more than one or a few samples per AABB.

3.5 Description of the Datasets

We evaluate our method on a synthetic dataset, referred to as the "Blender dataset", which was first used by Mildenhall et al. [11]. In actuality, this consists of eight different datasets, each composed of 2D images of an object, viewed from different camera angles and positions. The objects are of varying complexity, and all of them are of a bounded nature. In the middle of each scene is one or multiple objects, surrounded by empty space.

3.6 Evaluation Metrics

To evaluate the implementations' performance we have decided on some relevant metrics to use. We evaluate the image quality using three different metrics, Peak Signal-to-Noise Ratio (PSNR), Learned Perceptual Image Patch Similarity (LPIPS), and Structural Similarity Index (SSIM). We also measure the rendering time in order to compute the frames per second (FPS).

The evaluations are made using an evaluation script built into Nerfstudio. It compares 200 ground truth images (which are provided with the dataset) to the images rendered by the NeRF, and gives the average PSNR, LPIPS, and SSIM image quality metrics as well as the average FPS.

3.6.1 Image Quality

While simply looking at the rendered image can give a lot of information, there can be minute details that are easily missed by the naked eye. We have therefore chosen to use the same image quality metrics used by the original NeRF paper, which are PSNR, LPIPS, and SSIM. While each of these methods on their own might not be perfect, as discussed by Zhang et al. [19], together they should give some indication as to the quality of the produced image, especially when comparing between the models themselves. The metrics all use a ground truth image to get a number representing the difference. While a higher value on PSNR and SSIM represents better quality, for LPIPS it is better if the value is lower.

3.6.2 Rendering Speed

The speed of the rendering will be measured in frames per second (FPS). This metric is commonly used in computer graphics as it gives an idea of how smooth the experience is for the user. While movies are generally produced with a refresh rate of 24 FPS, the video game community strives to get as high a number as possible, some not accepting less than 144 FPS, or even 240 FPS in competitive scenarios [15]. When working with FPS it is important to realize that it is not a linear measure, but a reciprocal one [1]. This means that when you, for instance, take the arithmetic mean of FPS values, you get a number that is not the actual mean FPS. Instead, to get a correct value, you have to first compute the frame time, then take the mean of the frame times and lastly convert it back to FPS.

3.7 Technical Challenges

When implementing the different methods described above we encountered many challenges, the biggest of which was memory usage and performance. Initially, we created a PyTorch-based ray tracer, where the GPU-acceleration is conditioned on the usage of vectorized operations. This turned out to be very difficult when tracing multiple rays intersecting different AABBs, as some operations seemed to require using for loops instead. One also had to be careful not to trigger CPU-GPU synchronization, which for instance would occur when resizing a tensor. As the GPU executes operations asynchronously, other operations may be executed in parallel on the CPU. However, if a GPU operation is dependent on the result of an operation executed on the CPU, the operations need to be synchronized, which can be time-consuming. We spent a lot of time optimizing our PyTorch-based ray tracer by doing our best to make operations vectorized and avoiding synchronization.

While our PyTorch-based ray tracer did result in improved performance, the cost of traversing the BVH was unnecessarily high. We noticed that the performance peaked using around 64 bottom-level AABBs, after which it rapidly decreased. It is common for BVHs to be used to encapsulate millions of triangles, so the fact that our implementation peaked with so few AABBs was a clear indication that it was not optimal. We also noticed that we could simply flatten the BVH to a depth of 0, and still get similar performance. This indicated that the performance gains were not due to the usage of a BVH, but rather due to the still significant reduction in the number of samples.

For these reasons, we decided to look into CUDA. We opted to use the *cubvh* project, a ray tracer written in CUDA, and modified it to fit our requirements.

Chapter 4
Results

This chapter presents the results of our experiments. It would be infeasible to provide all our measurements, and we will therefore reduce the dimensionality by focusing on specific variables in each section. In Section 4.2 we focus on determining which model and split method performs the best using a fixed point cloud size and number of clusters, the determination of which is discussed in Section 5.4. In the following sections, we focus only on this split method and model, while instead looking at other variables. Then in Section 4.3 we show the effects of varying the point cloud size and number of clusters for a couple of the evaluated datasets. In Sections 4.4 and 4.5 we demonstrate the effects of varying the point cloud size using a fixed cluster count of 2048. Finally, in Section 4.6 we display close-up renders of each dataset.

4.1 Experimental Setup

The evaluations were conducted on an HP Z2 Tower G9 Workstation Desktop PC with the following specifications:

- CPU: 12th Gen Intel® Core™ i7-12700
- GPU: NVIDIA GeForce RTX 4070 Ti
- RAM: 16 GB
- OS: Ubuntu 22.04.3 LTS

4.2 Split Method and Model Comparisons

Tables 4.1 and 4.2 show a comparison of SAH and median split. The comparison is done using 2048 clusters and 1,000,000 points, but tables with all clusters and point clouds for the Nerfacto model can be found in Appendix A. The different trees were constructed entirely from scratch, including K-Means. We see a slight increase in FPS using SAH and Nerfacto, while no difference is seen using Vanilla-Nerf. No meaningful difference is seen in image quality for the two models.

Data set	PSNR ↑		SSIM ↑		LPIPS ↓		FPS ↑	
	SAH	Median	SAH	Median	SAH	Median	SAH	Median
chair	23.19	23.19	0.852	0.853	0.061	0.061	17.10	16.22
drums	18.23	18.22	0.820	0.820	0.112	0.112	12.86	12.49
ficus	19.48	19.48	0.874	0.874	0.095	0.095	4.34	4.32
hotdog	21.83	21.81	0.865	0.864	0.107	0.108	10.86	10.84
lego	25.17	25.17	0.884	0.884	0.043	0.043	12.61	12.35
mic	21.62	21.62	0.905	0.905	0.053	0.053	18.48	16.87
ship	20.70	20.68	0.737	0.737	0.178	0.178	6.27	6.27
Average	**21.46**	**21.27**	**0.848**	**0.848**	**0.093**	**0.092**	**9.32**	**9.26**
Baseline	**19.20**		**0.834**		**0.172**		**1.27**	

Table 4.1: Comparison of SAH and median split methods with Nerfacto as the model. Baseline uses the default Proposal Sampler.

Data set	PSNR ↑		SSIM ↑		LPIPS ↓		FPS ↑	
	SAH	Median	SAH	Median	SAH	Median	SAH	Median
chair	31.43	31.42	0.956	0.956	0.033	0.033	1.45	1.47
drums	24.25	24.25	0.911	0.911	0.081	0.080	0.86	0.86
ficus	29.32	29.32	0.959	0.959	0.033	0.033	0.47	0.47
hotdog	32.58	32.49	0.952	0.952	0.058	0.059	1.28	1.28
lego	32.09	32.09	0.960	0.960	0.021	0.021	1.00	1.01
materials	28.74	28.73	0.942	0.942	0.039	0.039	0.49	0.49
mic	32.24	32.21	0.979	0.978	0.023	0.023	0.96	0.95
ship	27.89	27.88	0.851	0.851	0.123	0.124	0.83	0.83
Average	**29.82**	**29.80**	**0.939**	**0.939**	**0.051**	**0.052**	**0.80**	**0.80**
Baseline	**30.63**		**0.946**		**0.045**		**0.050**	

Table 4.2: Comparison of SAH and median split methods with Vanilla-Nerf as the model. Baseline uses the default uniform sampler for the coarse network and PDF-sampler for the fine network.

4.3 Graphs

This section shows visualizations of the evaluations on different point clouds and cluster counts. We focus on the results of the Nerfacto model using SAH as the tree construction method for conciseness and due to its superior results as demonstrated in Section 4.2.

4.3.1 Lego

The plots in Figure 4.1 demonstrate that for a majority of the data points, our sampler achieves a slight increase in PSNR and a substantial improvement in FPS, as compared to the original sampler used in Nerfacto. The FPS peak using 2048 clusters and a point cloud of 100,000 points is due to there being too few points for each cluster, resulting in gaps in the rendered object. This can also be seen in the drop in PSNR at that point. Both the FPS and PSNR seem to benefit from a larger number of clusters, as long as the point cloud is of sufficient size. Below the plots are two rendered images of the Lego dataset, the left one for 100,000 points and the right one for 1,000,000 points. In the left image, a noticeable difference in quality can be seen, perhaps most notable around the "floor" of the model. Each rendered image's corresponding points in the plots can be found through the red lines.

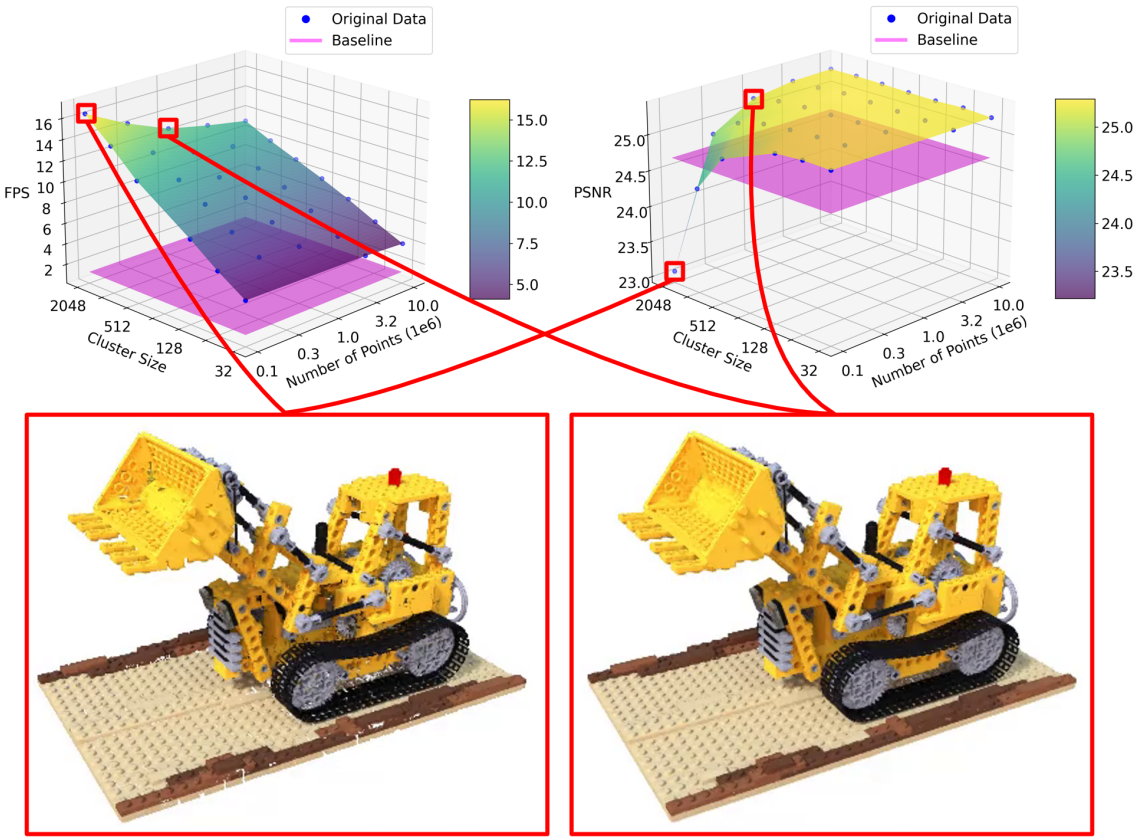


Figure 4.1: FPS and PSNR for the Lego dataset using the Nerfacto model with SAH as the tree construction method, along with rendered images from two of the data points. The baseline in purple is the Nerfacto model using its original sampler.

4.3.2 Ficus

As in the Lego dataset, Figure 4.2 also shows a peak in FPS at the smallest point cloud and cluster count for the Ficus dataset. What is more notable is that the PSNR also peaks at this point, where the Lego dataset started showing artifacts. This seems to indicate that the quality of the point cloud is starting to deteriorate. It should also be noted that the increase in FPS is not as substantial.

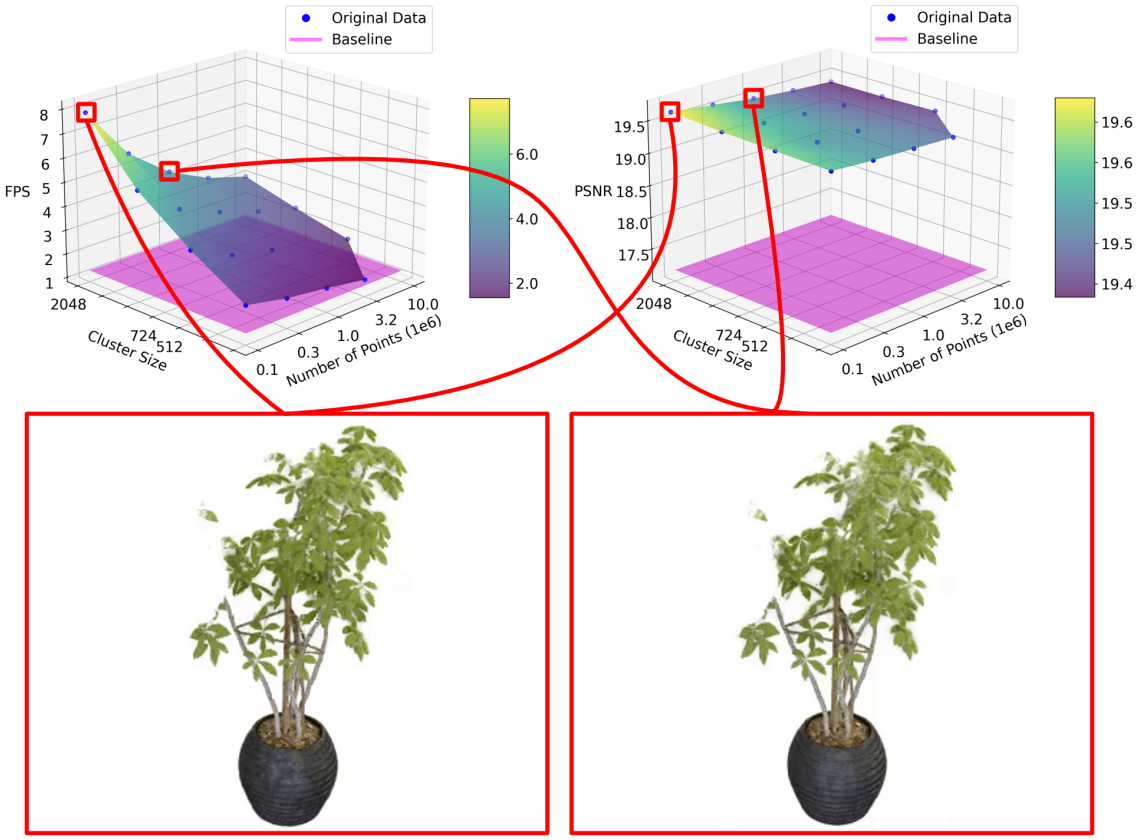


Figure 4.2: FPS and PSNR for the Ficus dataset using the Nerfacto model with SAH as the tree construction method, along with rendered images from two of the data points. The baseline in purple is the Nerfacto model using its original sampler.

4.3.3 Ship

Figure 4.3 demonstrates the simultaneous FPS peak and PSNR dip familiar from Section 4.3.1 at 2048 clusters and 100,000 points. The graph seems to follow the same behavior, where a larger number of clusters is superior as long as the point cloud is sufficiently large. In this dataset, we can however see that the PSNR output from the evaluation of our model is lower than that of the Nerfacto baseline.

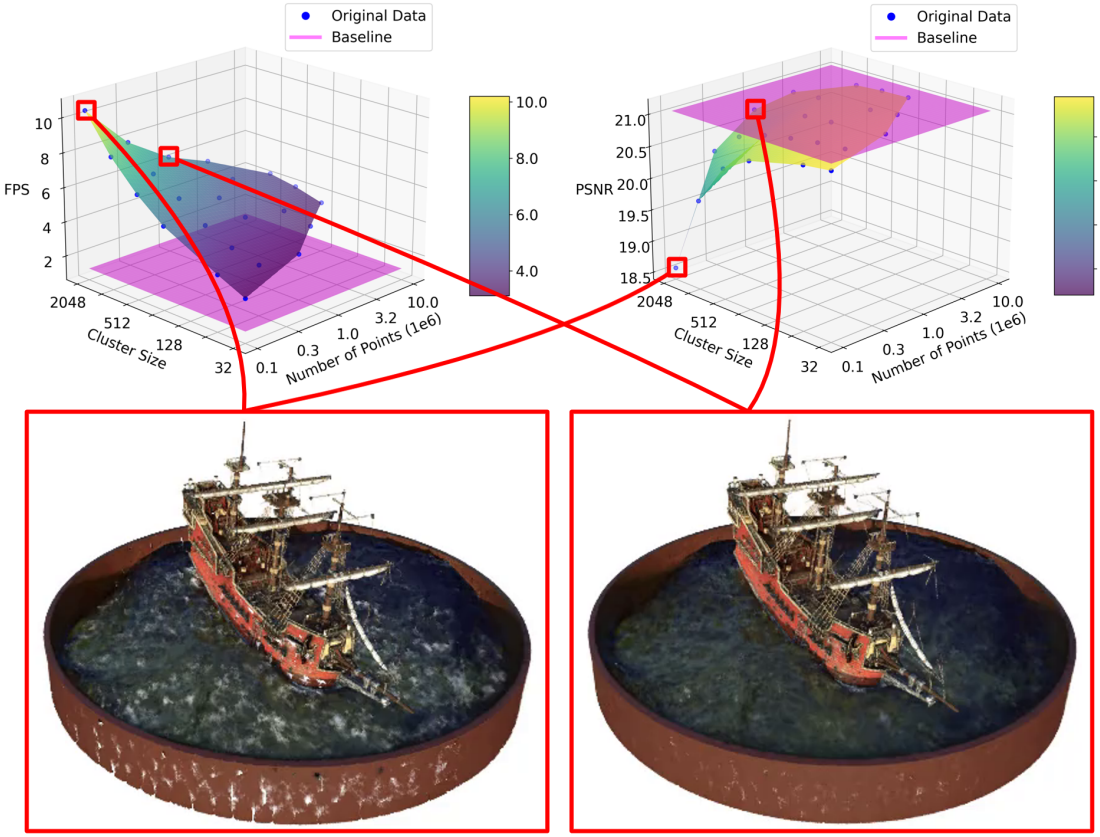


Figure 4.3: FPS and PSNR for the Ship dataset using the Nerfacto model with SAH as the tree construction method, along with rendered images from two of the data points. The baseline in purple is the Nerfacto model using its original sampler.

4.4 Comparison of Point Cloud Sizes

Figure 4.4 demonstrates the average metrics across the different datasets when using a fixed cluster count of 2048 for varying point cloud sizes. We see a rather steep decline in FPS as the number of points increases, while the image quality metrics seem to taper out somewhere around 1,000,000 points. The standard deviation of the frame rate also decreases slightly for larger point clouds. The image quality metrics (PSNR, SSIM, LPIPS) have some slight differences between SAH and median split, in this case mostly due to there being some datasets in which one of the methods does not have an evaluation for a point cloud size, while the other one does.

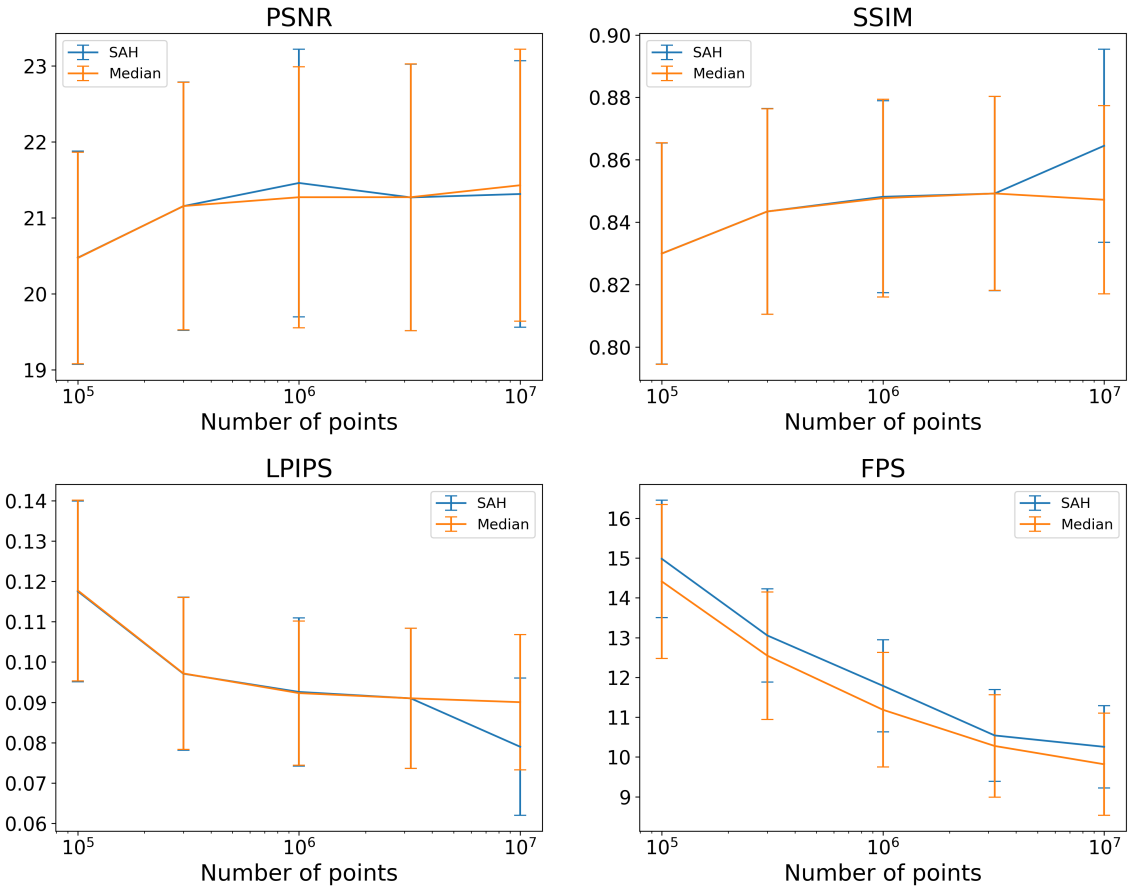


Figure 4.4: Average metrics of the different point cloud sizes for median split and SAH using the Nerfacto model using a cluster count of 2048. The X-axis uses a logarithmic scale. The vertical lines are the standard deviation at each point cloud size.

4.5 Visualization of Leaf Nodes

This section demonstrates the AABBs generated from the clustered point clouds exported from one model trained on the Lego dataset and one trained on the Ficus dataset. The purpose is to visualize the areas where samples will be allocated.

When using a small point cloud consisting of 100,000 points, gaps can be noticed between the AABBs, as seen in Figure 4.5 where some of the gaps are marked with red circles. This is especially noticeable when comparing to the AABBs generated from the larger point cloud created using 1,000,000 points, which is found in Figure 4.6. Notice however that in the latter figure, some slight noise is starting to appear, which is highlighted with the red circles.

In some models, such as the one created from the Ficus dataset, a considerable amount of noise, mainly at the outer parts of the scene, is caught by the point clouds. This noise obstructs the ficus itself, as seen in Figures 4.7 and 4.8. From the Lego dataset, the difference in noise caught by the larger and smaller point clouds is minimal, but when comparing the Ficus dataset figures the swelling is much more obvious.

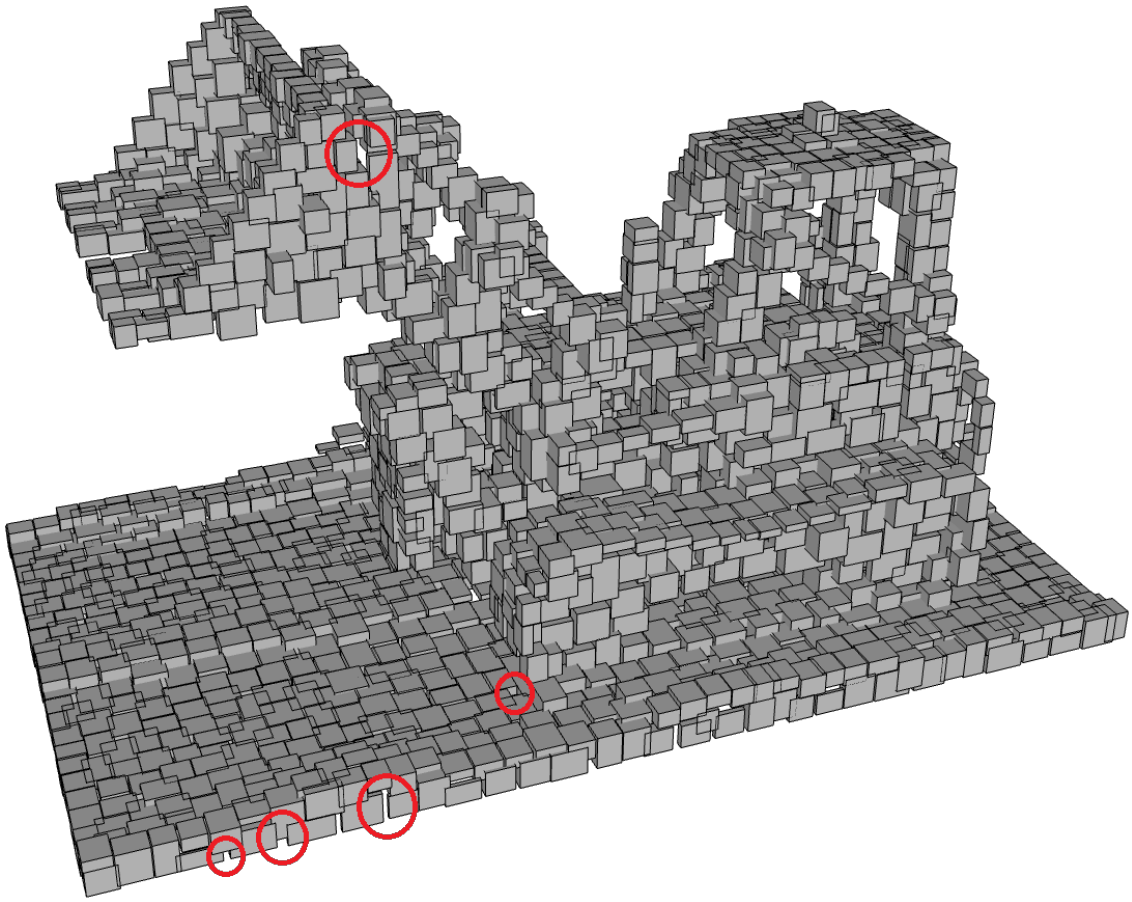


Figure 4.5: The bounding boxes of the leaf nodes constructed after K-Means for the Lego dataset and Nerfacto model. Constructed from 100,000 points and using 2048 clusters. The red circles highlight gaps in the object.

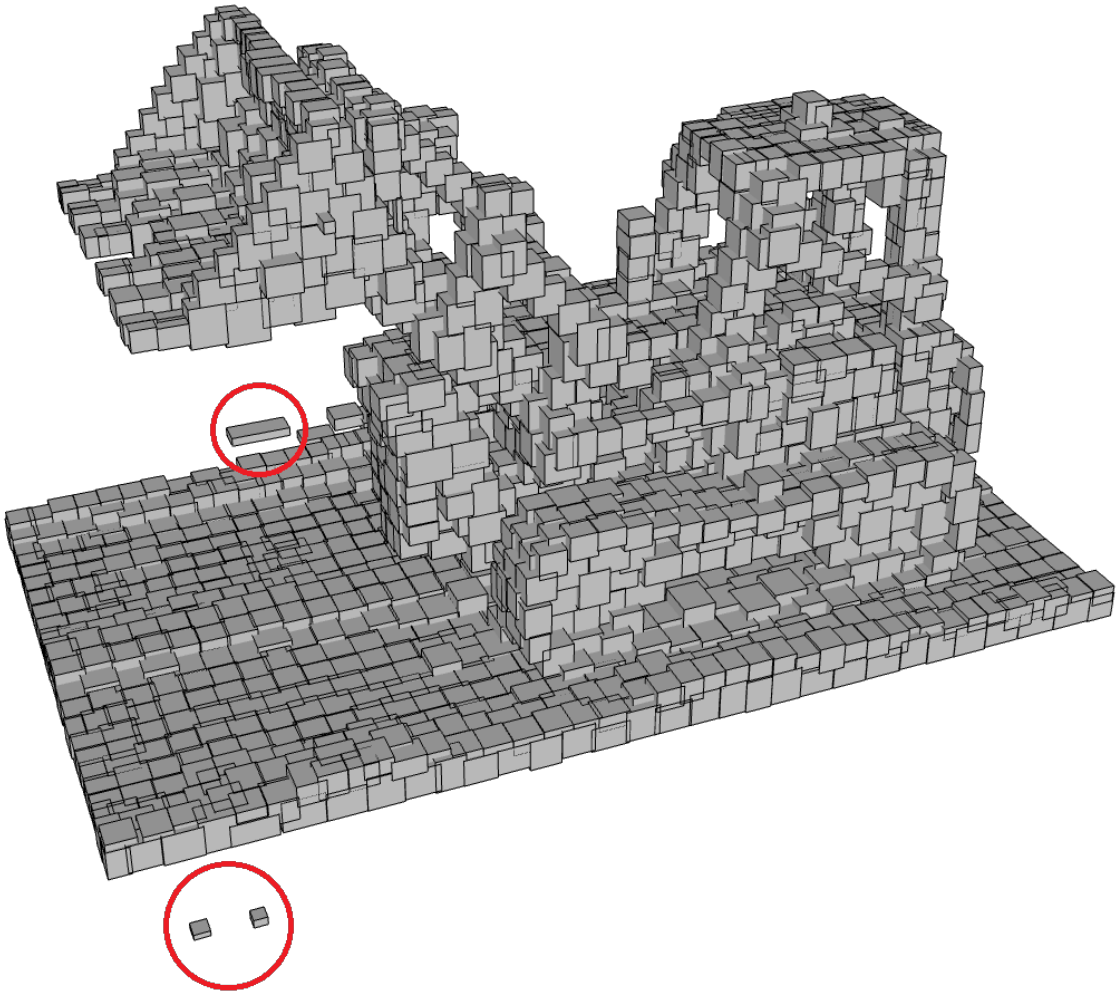


Figure 4.6: The bounding boxes of the leaf nodes constructed after K-Means for the Lego dataset and Nerfacto model. Constructed from 1,000,000 points and using 2048 clusters. The red circles highlight the noise in the object.

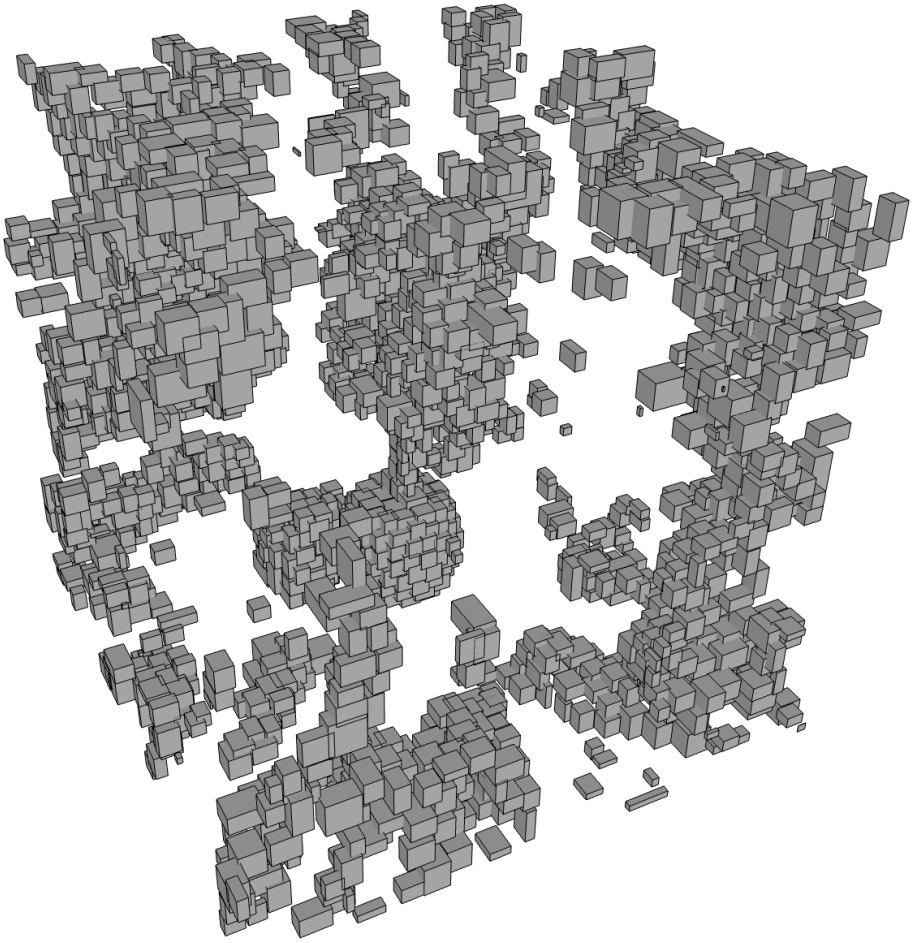


Figure 4.7: The bounding boxes of the leaf nodes constructed after K-Means for the Ficus dataset and Nerfacto model. Constructed from 100,000 points and using 2048 clusters.

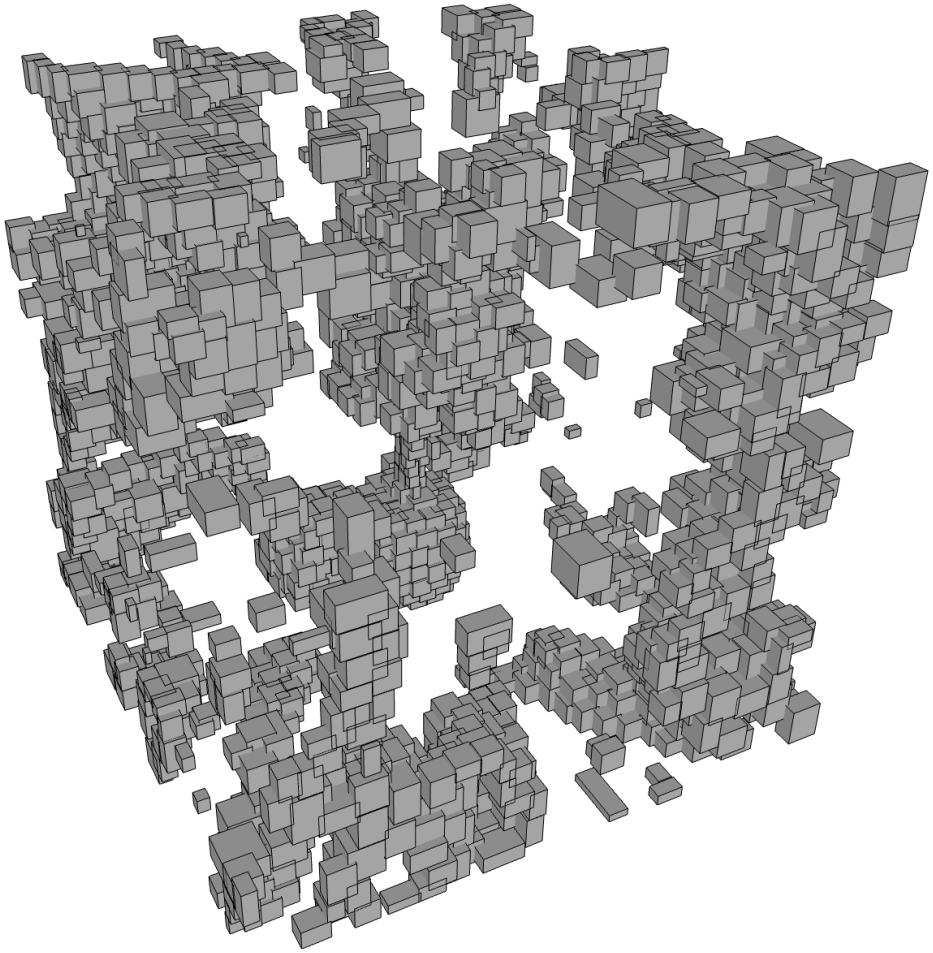


Figure 4.8: The bounding boxes of the leaf nodes constructed after K-Means for the Ficus dataset and Nerfacto model. Constructed from 1,000,000 points and using 2048 clusters.

4.6 Close-up Renders

Figure 4.9 shows close-ups of all datasets. The images in the first two columns are rendered using our sampler using a point cloud size of 1,000,000 points and 2048 clusters, and the second two using the default samplers. Some interesting artifacts appear in the hotdog dataset and the ship dataset. Notice that the structure of the bounding boxes is showing on the plate of the hotdog. Also, some white spots are appearing in the water underneath the ship as well as a white box in the middle of the ship itself. The ficus rendered using the Nerfacto method with the original sampler has quite some noise in the background, which does not appear in the other images. Furthermore, the Vanilla-Nerf method with the original sampler seems to be missing some of the tracks on the Lego dataset. Apart from these artifacts, the overall quality appears to be quite similar across the different models.

In Figure 4.10 we compare the quality of images rendered using 1,000,000 points and 10,000,000 points, both using 2048 clusters. The artifacts mentioned in the previous paragraph are noticeably reduced, however still present to some extent. However, notice the white box in the center of the ship in the Ship dataset becoming larger, using 10,000,000 points and the Vanilla-Nerf model.

Finally, in Figure 4.11 we use a point cloud size of 1,000,000 for all images, and compare the image quality using 512 and 2048 clusters. This does not seem to have any significant effects, if any, on the artifacts.

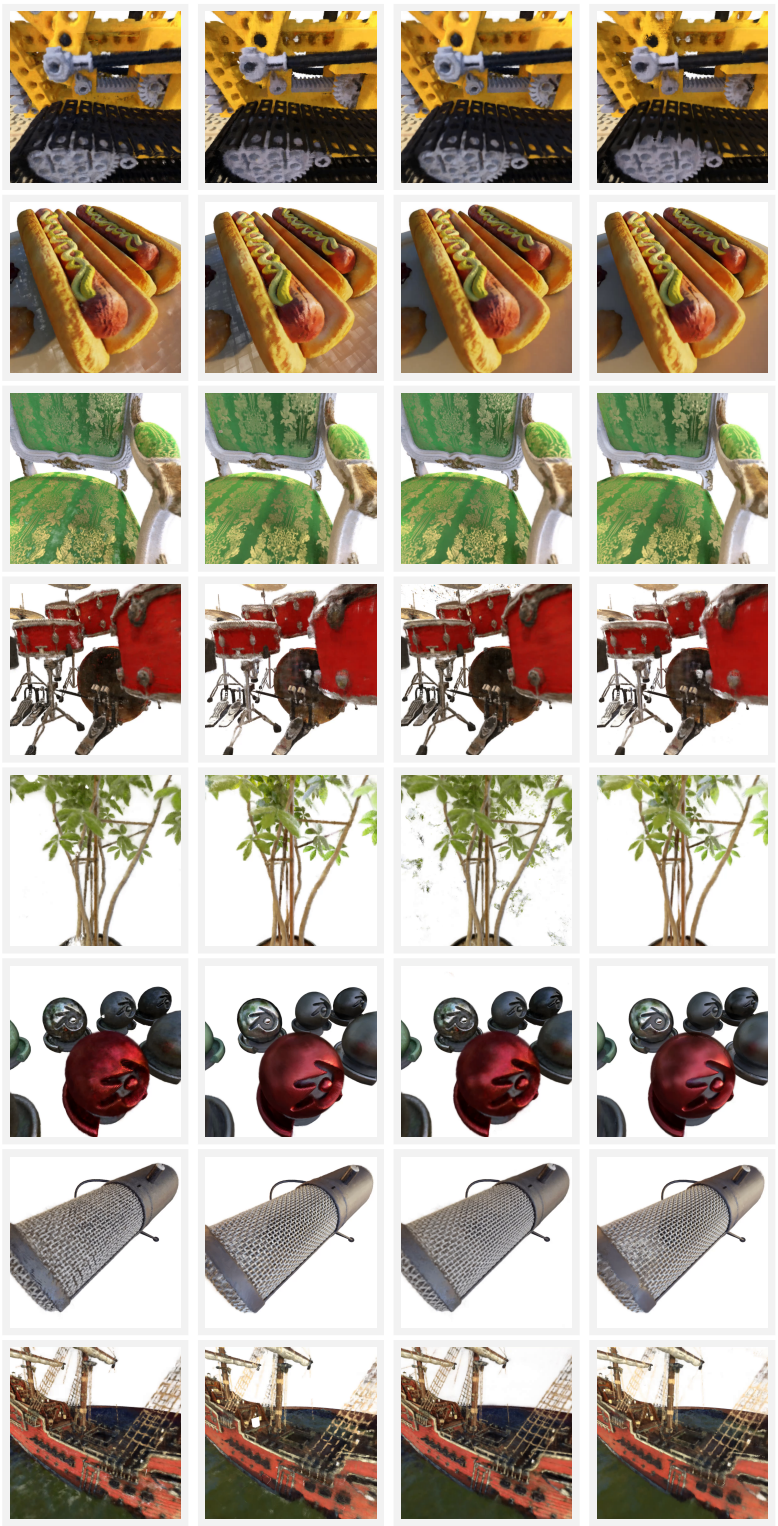


Figure 4.9: Comparison of images rendered using our BVH-based sampler and the original samplers of Nerfacto and Vanilla-Nerf. In the example we use 1,000,000 points and 2048 clusters.

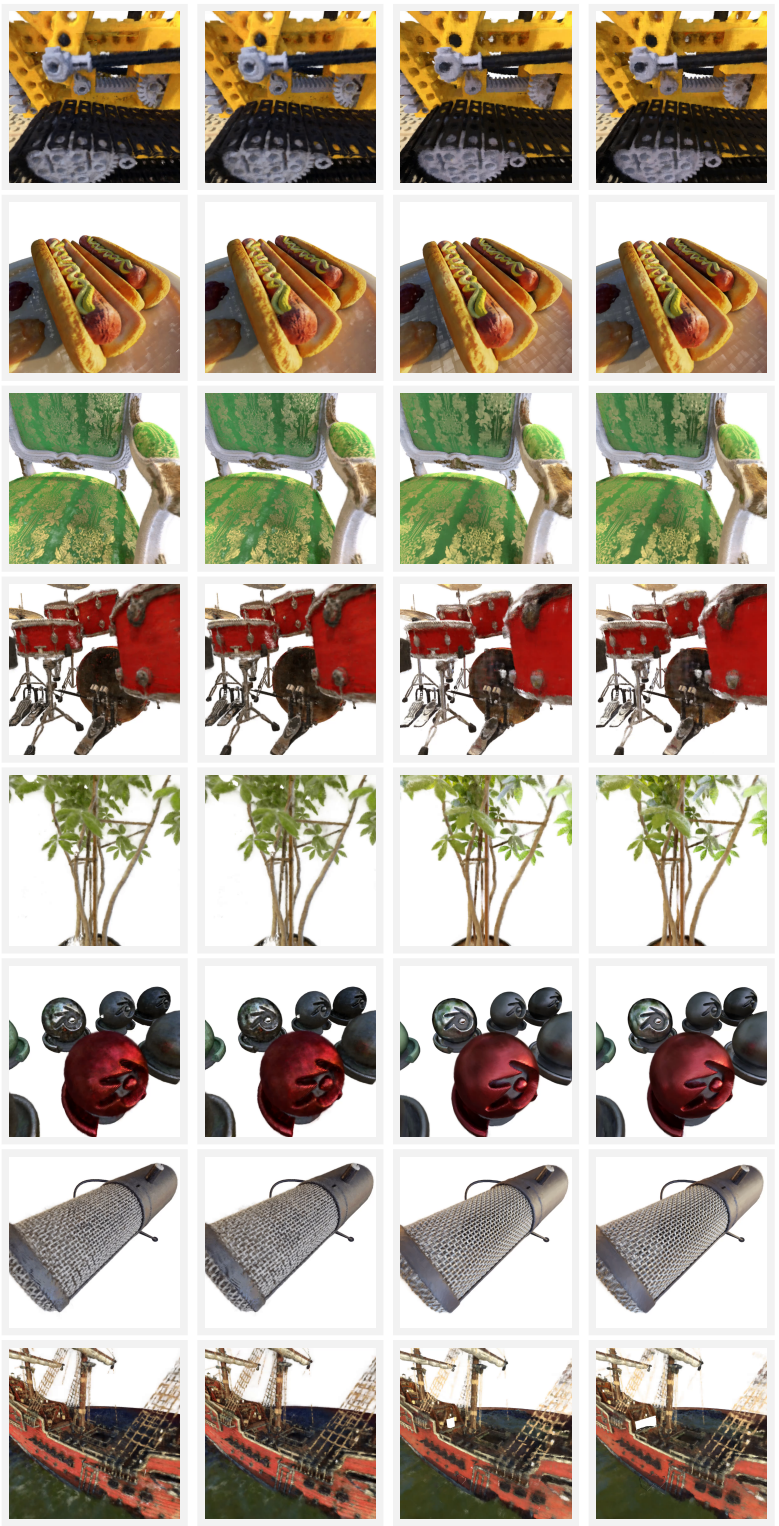


Figure 4.10: Comparison of images rendered using 1,000,000 and 10,000,000 points, all with 2048 clusters.

Figure 4.11: Comparison of images rendered using 512 and 2048 clusters, all with 1,000,000 points.

Chapter 5
Discussion

5.1 Rendering Performance and Quality Impact

Compared to the baseline models, our evaluations show a large improvement in rendering times. On some of the models we also see a noticeable improvement in the image quality metrics, however this should be taken with a large grain of salt. The trained models contain a fair bit of noise, manifesting as floaters at the other parts of the scene. As we specify a bounding box from which to export the point clouds, this seems to effectively cut away a large portion of this noise, which has a tremendous impact on the quality. In the ship model there was more or less no noise present, and there we see that our model has slightly decreased image quality, which is probably closer to the truth. This belief is also strengthened by the close-ups in Section 4.6. When close enough to the objects in the scene, artifacts appear in some of the data sets. This mainly seems to be an issue that occurs when an object has a very thin representation in the NeRF, such as for the plate in the hotdog dataset, or the water surface in the ship dataset. As the samplers used during training place a fixed number of samples along each ray, it is likely that when a ray intersects a very thin surface, there will be a high concentration of samples placed at the intersection point. It is then possible that each sample has a fairly low density, but when they are accumulated during rendering, the densities add up close to one. When we render using our sampler, we will likely reduce the sampling density so drastically that the accumulated density is much lower, resulting in a translucent appearance.

We notice that increasing the point cloud size seems to reduce the artifacts to some extent, although it does not eliminate them. This might be because a larger point cloud more accurately captures where the density is allocated in the NeRFs representation of the scene. This is further proven by the visualization of leaf nodes in Section 4.5 where the ficus leaf nodes, when using the larger point cloud in Figure 4.8, appear to have swollen up compared to using the smaller point cloud in Figure 4.7. We had a theory that decreasing the number

of clusters might alleviate the issue as well, due to the larger bounding boxes, but our results show that this is not the case.

In Section 4.6 we also notice that a white box is appearing on the ship when using the Vanilla-Nerf model. We believe this is caused by K-Means failing to converge to a sufficient result. As explained in Section 3.2.3, we repeat K-Means up to five times in an attempt to catch such results. The strategy we implemented is however not foolproof. When K-Means fails to converge, this usually manifests as one or multiple bounding boxes being exceedingly large, capturing large areas of empty space. The sampling strategy described in Section 3.4 places samples step by step within a bounding box until we hit the specified upper limit. If a bounding box encapsulates a lot of empty space, this limit may be hit before any density is reached, such that all samples are placed in empty space. It is then expected that this would result in a completely white pixel.

5.2 Comparison of Split Methods

We see a slight increase in rendering speed while using SAH as the split method when compared to median split. This does however result in a substantial increase in the construction time of the BVH due to the more complex heuristic function. The slight variations in the quality metrics are only due to the fact that during evaluation, the trees are constructed from scratch, including K-Means clustering. As K-Means is random in its initiation, this may result in different leaf nodes. Technically there should be no difference in quality if the trees were constructed using the same leaf nodes, as it would result in the same sampling positions. It would have been more correct to save the leaf nodes and use the same ones for both split methods, however, this was realized too late in the process.

We believe that the main reason why there is no substantial difference between SAH and median split is due to K-Means resulting in highly uniform clusters, such that the AABBs generated will have similar surface areas. As seen in Figures 4.5 and 4.6, the floor consists of very many uniform AABBs rather than one large AABB, which probably would not result in a substantial increase in the number of samples. Aside from the usage of larger AABBs resulting in fewer intersection tests, this could also lead to SAH creating trees that are better optimized, and thus getting a more substantial difference in FPS performance compared to median split.

5.3 Comparison of Models

Our model which is based on the Nerfacto model in Nerfstudio has much better performance when compared to the one based on Vanilla Nerf. This is expected, as there are several other improvements in Nerfacto apart from the sampler which contributes to its superior performance. The benefit of using Vanilla Nerf is still its better image quality, which is also present using our sampling method.

Notice that while we improved the FPS of the Nerfacto model by a factor of about seven, the Vanilla-Nerfacto model had a speed-up of on average 16 times.

5.4 Variations in Point Cloud Quality

As seen in Sections 4.3 and 4.5, while the Lego dataset seems to only benefit from increasing the point cloud size, the Ficus dataset does not. This likely originates from the fact that the Lego dataset has little to no noise within the bounding box from which the point cloud is exported, while the Ficus dataset has a substantial amount of noise. This noise is further amplified as the point cloud size grows. This explains the behavior we see in Figure 4.2, which is both higher image quality and better rendering times using the smaller point cloud. Our interpretation of this is that noisier models may benefit from smaller point clouds, while models with little to no noise would benefit from much larger ones.

We decided to use the same scene bounding box size for all the datasets when exporting the point cloud. Because the point cloud generation is part of a pre-processing stage, these values could be optimized to fit the models a lot tighter which would result in the noise being cut away and the bounding boxes enclosing these noisy areas therefore never being created. This would probably result in the graphs for models similar to the Ficus one becoming a lot more like the Lego graph and therefore, making our implementation more reliable.

Worth mentioning is that we believe the substantial amount of noise on the Ficus dataset originates from the leaves being very thin, and the NeRF therefore having a hard time pinpointing exactly where the density is located.

5.5 Limitations

Our work focuses on bounded scenes of synthetic datasets. Unbounded scenes rely on concepts such as scene contraction, which we determined early on would present a different challenge, as the exported point clouds would then also be effectively unbounded. Many of the datasets also contain a lot of empty space. This is clearly to our benefit, as it means that we are able to cut away a large portion of samples just due to this fact. The rendering times are closely related to how densely packed the scenes are, so the performance would likely decrease in scenes with a high density of objects, as we only employ uniform sampling within the bounding boxes.

As can be seen in the graphs in Section 4.3, for some datasets there are missing points. This is due to the GPU running out of memory during evaluation. We found it difficult to see any pattern in this behavior and therefore chose to simply omit those points from the results. It is however clear that it occurs more frequently in datasets where the leaf nodes occupy a larger portion of the scene, such as Ficus (see Figures 4.2 and 4.8). This likely results in too many samples for the GPU to process concurrently.

5.6 Future Research

There exist many interesting areas of improvement in our work. This section covers some of the areas where we see a lot of potential.

5.6.1 Unbounded Scenes

As discussed in Section 5.5, unbounded scenes seem to require creative ways of exporting and handling the point clouds. It may be that it is sufficient to only increase the size of the bounding box from which the point cloud is generated, but it remains unclear how one would determine the limits. Many of the interesting use cases for NeRFs, such as for Virtual Reality, are constructed from unbounded scenes, so this is a clear area of future interest.

5.6.2 Different Sampling Schemes

We simply employ uniform sampling within the bounding boxes, which is fairly trivial. This does not take into account the distribution of density within the bounding boxes, such that samples are allocated to areas of greater density. This presents the possibility of further reducing the number of samples and increasing the image quality. It would be especially beneficial for scenes with a high density of objects, in which our method would allocate too many samples for efficient rendering. As proven by Neff et al. [13], a very small number of samples may suffice if they are placed close to the first surface. It would therefore be very interesting to experiment with only computing the first intersection, and placing a smaller number of samples around this position. Although we did some slight experimentation with this approach at a time when we only computed the first intersection, we were not able to achieve sufficient quality as the first "surface" might just be noise.

5.6.3 Tree Construction

We utilize K-Means in order to instantiate the leaf nodes from which we construct the BVH bottom-up. As discussed in Section 5.2, this results in fairly uniformly sized leaf nodes, which might not be optimal, therefore it would be interesting to explore other methods of instantiating the leaf nodes. Furthermore, other heuristic functions for effective splitting can be explored which might allow for more efficient traversal.

5.6.4 Hardware Accelerated Ray Tracing

While the CUDA-based ray tracer utilizes GPU-acceleration, allowing for much faster performance than the initial PyTorch-based ray tracer, CUDA is optimized for a general-purpose acceleration of computations. Many modern GPUs have specific hardware for ray tracing, and even BVH, which we do not utilize in this project. By using a ray tracer that utilizes such hardware, we would likely be able to speed up the rendering further. This poses an excellent opportunity for future study.

Chapter 6
Conclusions

Our work demonstrates the potential for a large improvement in rendering speeds with limited effect on image quality, although some datasets with thin surfaces suffer from artifacts, by utilizing BVHs in order to reduce sampling frequency. The approach does however come with challenges, such as finding the optimal point cloud size, which seems to vary depending on the dataset. In most cases, it seems to be beneficial to use a larger number of clusters during K-Means as long as the point cloud is of sufficient size. This does however present a practical upper limit, as the point cloud quality appears to deteriorate at some point and K-Means becomes exceedingly expensive to execute. The deterioration of the point cloud quality appears to occur at a lower number of points for more densely packed scenes. We conclude that there seems to be a small increase in rendering performance when constructing the BVH using the surface area heuristic when compared to median split. This does however result in a substantial increase in the construction time, so there is a trade-off between the increase in rendering performance and the increase in pre-processing time. We recognize interesting areas of future research, especially that of unbounded scenes, and utilizing ray tracing hardware.

References

[1] Tomas Akenine-Moller, Eric Haines, Sebastien Hillaire, Naty Hoffman, Michał Iwanicki, and Angelo Pesce. *Real-time rendering, Fourth edition*. CRC Press, 2018.

[2] Ashawkey. cubvh. `https://github.com/ashawkey/cubvh`, 2024.

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CoRR*, abs/2111.12077, 2021. URL `https://arxiv.org/abs/2111.12077`.

[4] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., USA, 2004. ISBN 1558607323.

[5] Kyle Gao, Yina Gao, Hongjie He, Dening Lu, Linlin Xu, and Jonathan Li. Nerf: Neural radiance field in 3d vision, a comprehensive review, 2023.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[7] Jbikker. How to build a bvh – part 2: Faster rays, April 2022. URL `https://jacco.ompf2.com/2022/04/18/how-to-build-a-bvh-part-2-faster-rays/`. Accessed: 2024-05-03.

[8] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, aug 1986. ISSN 0097-8930. doi: 10.1145/15886.15916. URL `https://doi.org/10.1145/15886.15916`.

[9] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics*, 6(3):153 – 166, 1990. ISSN 0178-2789. URL `https://api.semanticscholar.org/CorpusID:3341582e`.

[10] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the wild: Neural radiance fields for unconstrained photo collections. *CoRR*, abs/2008.02268, 2020. URL `https://arxiv.org/abs/2008.02268`.

[11] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

[12] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *CoRR*, abs/2201.05989, 2022. URL `https://arxiv.org/abs/2201.05989`.

[13] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H. Mueller, Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, and Markus Steinberger. DONeRF: Towards Real-Time Rendering of Compact Neural Radiance Fields using Depth Oracle Networks. *Computer Graphics Forum*, 40(4), 2021. ISSN 1467-8659. doi: 10.1111/cgf.14340. URL `https://doi.org/10.1111/cgf.14340`.

[14] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*, 2020.

[15] Tony Tamasi. Why does high fps matter for esports?, December 2019. URL `https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/`. Accessed: 2024-05-03.

[16] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Justin Kerr, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, David McAllister, and Angjoo Kanazawa. Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH 2023 Conference Proceedings*, SIGGRAPH '23, 2023.

[17] Krishna Wadhwani and Tamaki Kojima. Squeezenerf: Further factorized fastnerf for memory-efficient inference. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, June 2022. doi: 10.1109/cvprw56347.2022.00307. URL `http://dx.doi.org/10.1109/CVPRW56347.2022.00307`.

[18] Zirui Wang, Shangzhe Wu, Weidi Xie, Min Chen, and Victor Adrian Prisacariu. Nerf-: Neural radiance fields without known camera parameters. *CoRR*, abs/2102.07064, 2021. URL `https://arxiv.org/abs/2102.07064`.

[19] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 586–595, Los Alamitos, CA, USA, jun 2018. IEEE Computer Society. doi: 10.1109/CVPR.2018.00068. URL `https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00068`.

Appendices

Appendix A
Evaluation Tables

The tables in this section present the same data that is shown in the graphs in Section 4.3, along with the LPIPS and SSIM metrics, for each dataset. Each metric is heat-mapped from yellow (best) to blue (worst). The best metrics are highlighted in bold. Missing values are marked as "N/A".

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.4K	32	25.24	0.886	0.041	4.44	25.24	0.886	0.041	4.32
	64	25.24	**0.887**	0.041	6.13	25.25	0.886	0.041	6.10
	128	N/A	N/A	N/A	N/A	25.22	0.886	0.041	7.74
	256	25.13	0.885	0.043	9.89	25.13	0.885	0.043	9.65
	512	24.92	0.882	0.046	11.62	24.94	0.882	0.046	11.12
	1024	24.37	0.875	0.057	13.88	24.35	0.875	0.056	13.48
	2048	23.11	0.858	0.080	**16.38**	23.09	0.859	0.081	**15.60**
316.5K	64	25.27	**0.887**	0.041	6.01	25.27	**0.887**	0.041	5.94
	128	25.26	**0.887**	**0.040**	7.42	25.26	**0.887**	**0.040**	7.23
	256	25.23	0.886	0.041	9.03	25.25	0.886	0.041	8.78
	512	25.20	0.886	0.041	10.53	25.19	0.886	0.041	10.38
	1024	25.09	0.884	0.043	12.37	25.09	0.884	0.043	12.35
	2048	24.80	0.878	0.050	14.32	24.79	0.879	0.050	14.10
1.0M	32	25.26	0.886	0.041	4.37	25.26	0.886	0.041	4.32
	64	25.27	**0.887**	0.041	5.59	25.27	**0.887**	0.041	5.53
	128	25.28	**0.887**	0.041	7.02	N/A	N/A	N/A	N/A
	256	25.27	**0.887**	**0.040**	8.47	25.27	**0.887**	0.041	8.27
	512	25.27	**0.887**	0.041	9.79	25.27	**0.887**	**0.040**	9.51
	1024	25.25	0.886	0.041	11.29	25.24	0.886	0.041	11.09
	2048	25.17	0.884	0.043	12.61	25.17	0.884	0.043	12.35
3.2M	64	25.25	**0.887**	0.041	5.19	25.25	**0.887**	0.041	4.97
	128	25.27	0.886	0.041	6.78	25.27	0.886	0.041	6.51
	256	25.28	**0.887**	0.041	8.16	25.27	**0.887**	0.041	7.79
	512	**25.29**	**0.887**	0.041	9.22	**25.29**	**0.887**	0.041	8.98
	1024	25.27	**0.887**	0.041	10.46	25.28	**0.887**	0.041	10.05
	2048	25.25	0.886	0.041	11.70	25.26	0.886	0.041	11.20
10.0M	32	25.23	0.886	0.042	4.03	25.23	0.886	0.042	3.99
	64	25.26	0.886	0.041	5.26	25.26	0.886	0.041	5.19
	128	25.25	0.886	0.041	6.58	25.25	0.886	0.041	6.50
	256	25.25	0.886	0.041	7.77	25.26	0.886	0.041	7.70
	512	25.27	0.886	0.041	8.80	25.27	0.886	0.041	8.67
	1024	25.27	0.886	0.041	9.84	25.27	0.886	0.041	9.89
	2048	25.27	0.886	0.041	10.54	25.27	0.886	0.042	10.97

Table A.1: Evaluation metrics for the lego dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.0K	32	23.32	**0.856**	**0.057**	7.23	23.32	**0.856**	**0.057**	7.22
	64	23.29	**0.856**	**0.057**	10.33	23.29	**0.856**	**0.057**	10.26
	128	23.28	0.855	**0.057**	12.68	23.28	0.855	0.058	12.75
	256	23.24	0.854	0.059	14.59	23.25	0.854	0.059	14.39
	512	23.18	0.853	0.063	16.13	23.18	0.853	0.062	15.50
	1024	23.00	0.849	0.069	18.05	23.01	0.850	0.069	17.26
	2048	22.62	0.842	0.084	**19.60**	22.61	0.842	0.083	**18.80**
316.2K	32	**23.33**	**0.856**	**0.057**	7.34	**23.33**	**0.856**	**0.057**	7.34
	64	23.30	**0.856**	**0.057**	9.68	23.30	**0.856**	**0.057**	9.56
	128	23.29	0.855	**0.057**	12.33	23.29	0.855	**0.057**	12.01
	256	23.26	0.855	0.058	14.00	23.26	0.855	0.058	13.21
	512	23.22	0.854	0.060	15.43	23.23	0.854	0.060	14.63
	1024	23.19	0.853	0.062	16.89	23.18	0.853	0.061	15.49
	2048	23.12	0.850	0.065	18.35	23.11	0.850	0.065	17.77
1.0M	32	23.31	**0.856**	**0.057**	5.91	23.31	**0.856**	**0.057**	5.89
	64	23.30	**0.856**	**0.057**	9.35	23.30	**0.856**	**0.057**	9.20
	128	23.30	0.855	**0.057**	11.59	23.30	0.855	**0.057**	11.23
	256	23.27	0.855	0.058	13.32	23.27	0.855	0.058	12.63
	512	23.25	0.854	0.058	14.50	23.24	0.854	0.059	14.14
	1024	23.22	0.854	0.060	15.29	23.22	0.854	0.060	15.41
	2048	23.19	0.852	0.061	17.10	23.19	0.853	0.061	16.22
3.2M	32	23.32	**0.856**	**0.057**	5.93	23.32	**0.856**	**0.057**	5.90
	64	23.30	**0.856**	**0.057**	8.22	23.30	**0.856**	**0.057**	8.16
	128	23.30	0.855	**0.057**	10.94	23.30	0.855	**0.057**	10.80
	256	23.28	0.855	0.058	12.39	23.28	0.855	0.058	11.92
	512	23.26	0.855	0.058	13.82	23.26	0.855	0.058	13.33
	1024	23.24	0.854	0.059	14.90	23.24	0.854	0.059	14.56
	2048	23.22	0.853	0.060	15.97	23.21	0.853	0.060	15.30
10.0M	32	23.28	**0.856**	0.058	3.45	23.28	**0.856**	0.058	3.45
	64	23.30	0.855	**0.057**	5.13	23.30	0.855	**0.057**	5.12
	128	23.30	**0.856**	**0.057**	9.76	23.30	**0.856**	**0.057**	9.64
	256	23.28	0.855	0.058	11.60	23.28	0.855	0.058	11.15
	512	23.26	0.855	0.058	13.17	23.26	0.855	0.058	12.88
	1024	23.23	0.854	0.059	14.35	23.23	0.854	0.059	13.57
	2048	23.22	0.854	0.060	15.33	23.23	0.854	0.060	14.57

Table A.2: Evaluation metrics for the chair dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.0K	32	18.27	**0.821**	0.111	1.91	N/A	N/A	N/A	N/A
	64	18.27	**0.821**	0.111	3.03	18.27	**0.821**	0.111	3.07
	128	18.26	**0.821**	**0.110**	4.96	18.26	**0.821**	**0.110**	4.84
	256	18.25	**0.821**	**0.110**	8.23	18.25	**0.821**	0.111	8.10
	512	18.24	**0.821**	0.111	11.54	18.24	**0.821**	0.111	11.17
	1024	18.19	0.818	0.115	14.40	18.19	0.818	0.115	14.29
	2048	18.04	0.810	0.126	**17.08**	18.04	0.810	0.126	**16.35**
315.8K	64	**18.28**	**0.821**	0.111	2.16	N/A	N/A	N/A	N/A
	128	18.26	**0.821**	0.111	4.26	18.26	**0.821**	0.111	4.26
	256	18.26	**0.821**	0.111	7.04	18.26	**0.821**	0.111	6.82
	512	18.25	**0.821**	0.111	9.73	18.25	**0.821**	0.111	9.57
	1024	18.23	0.820	0.111	12.43	18.23	0.820	0.111	11.78
	2048	18.21	0.818	0.113	14.93	18.21	0.818	0.113	14.07
999.2K	64	N/A	N/A	N/A	N/A	**18.28**	**0.821**	0.111	1.94
	128	18.27	**0.821**	0.111	3.17	18.27	**0.821**	0.111	3.57
	256	18.26	**0.821**	0.111	5.90	18.26	**0.821**	0.111	6.12
	512	18.25	**0.821**	0.111	8.57	18.26	**0.821**	0.111	8.43
	1024	18.24	0.820	0.111	11.04	N/A	N/A	N/A	N/A
	2048	18.23	0.820	0.112	12.86	18.22	0.820	0.112	12.49
3.2M	128	18.27	**0.821**	0.112	2.85	18.27	**0.821**	0.112	2.85
	256	18.27	**0.821**	0.111	4.84	18.27	**0.821**	0.111	4.82
	1024	18.24	0.820	0.112	9.38	18.24	0.820	0.112	9.37
	2048	18.23	0.820	0.112	10.84	18.23	0.820	0.112	11.17
10.0M	256	18.26	**0.821**	0.112	4.17	18.26	**0.821**	0.112	3.93
	512	18.25	0.820	0.112	6.19	18.25	0.820	0.112	6.27
	1024	18.24	0.820	0.112	8.31	18.25	0.820	0.112	8.14
	2048	18.22	0.819	0.113	9.64	18.23	0.819	0.113	9.70

Table A.3: Evaluation metrics for the drums dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.5K	256	19.55	0.875	0.093	2.42	19.54	0.875	0.094	2.46
	512	19.57	**0.876**	0.093	3.78	19.55	0.875	0.093	3.69
	1024	19.59	**0.876**	0.091	5.42	19.60	**0.876**	**0.091**	5.43
	2048	**19.63**	**0.876**	**0.090**	**7.86**	**19.64**	**0.876**	**0.091**	**7.69**
316.1K	256	19.50	0.875	0.095	2.05	N/A	N/A	N/A	N/A
	512	19.50	0.875	0.094	2.97	19.50	0.875	0.094	2.97
	1024	19.53	0.875	0.093	4.08	19.52	0.875	0.093	4.16
	2048	19.56	0.875	0.091	5.66	19.55	0.875	**0.091**	5.59
999.4K	256	19.47	0.874	0.096	1.82	N/A	N/A	N/A	N/A
	512	19.47	0.874	0.096	2.56	N/A	N/A	N/A	N/A
	1024	19.48	0.874	0.095	3.38	19.49	0.874	0.095	3.38
	2048	19.48	0.874	0.095	4.34	19.48	0.874	0.095	4.32
3.2M	256	19.44	0.874	0.097	1.54	N/A	N/A	N/A	N/A
	512	N/A	N/A	N/A	N/A	19.44	0.874	0.097	2.16
	1024	19.43	0.873	0.097	2.82	19.43	0.873	0.097	2.80
	2048	19.42	0.873	0.097	3.53	19.42	0.873	0.097	3.53
10.0M	512	19.40	0.873	0.099	1.83	19.40	0.873	0.098	1.85
	1024	19.38	0.873	0.098	2.40	19.40	0.873	0.098	2.44
	2048	19.38	0.873	0.099	3.04	19.37	0.872	0.099	3.02

Table A.4: Evaluation metrics for the ficus dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
101.3K	32	**22.08**	**0.879**	**0.078**	3.70	N/A	N/A	N/A	N/A
	64	22.07	**0.879**	0.079	4.50	**22.07**	**0.879**	**0.079**	4.50
	128	22.04	0.878	0.083	6.07	N/A	N/A	N/A	N/A
	256	22.00	0.876	0.087	7.53	22.01	0.876	0.087	7.43
	512	21.88	0.871	0.100	9.67	21.96	0.871	0.100	9.36
	1024	21.65	0.858	0.124	11.84	21.59	0.857	0.126	11.57
	2048	20.60	0.828	0.161	**14.39**	20.62	0.828	0.160	**14.03**
317.7K	32	21.80	**0.879**	0.080	3.39	N/A	N/A	N/A	N/A
	64	21.85	**0.879**	0.080	4.32	21.85	**0.879**	0.080	4.30
	128	21.83	0.878	0.081	5.40	21.83	0.878	0.082	5.36
	256	21.82	0.877	0.084	6.92	N/A	N/A	N/A	N/A
	512	N/A	N/A	N/A	N/A	21.94	0.875	0.089	8.44
	1024	21.85	0.868	0.104	10.34	21.85	0.868	0.104	10.27
	2048	21.67	0.855	0.122	12.41	21.69	0.854	0.122	12.05
1.0M	128	21.79	0.878	0.081	5.34	21.78	0.878	0.081	5.23
	256	21.77	0.877	0.084	6.53	N/A	N/A	N/A	N/A
	512	21.85	0.876	0.086	7.87	21.83	0.875	0.088	7.92
	1024	21.87	0.872	0.097	9.39	21.88	0.871	0.097	9.27
	2048	21.83	0.865	0.107	10.86	21.81	0.864	0.108	10.84
3.2M	64	N/A	N/A	N/A	N/A	21.68	0.878	0.081	3.93
	128	21.66	0.878	0.082	5.13	21.66	0.878	0.082	5.11
	256	21.66	0.877	0.083	6.26	21.64	0.877	0.083	6.11
	512	21.65	0.876	0.086	7.53	21.68	0.876	0.086	7.49
	1024	21.67	0.873	0.094	8.82	21.68	0.873	0.094	8.74
	2048	21.66	0.868	0.102	10.11	21.69	0.868	0.102	9.92
10.0M	64	N/A	N/A	N/A	N/A	21.61	0.878	0.081	4.03
	128	21.59	0.878	0.082	4.97	21.59	0.878	0.082	4.95
	256	21.59	0.877	0.083	6.16	N/A	N/A	N/A	N/A
	512	21.57	0.876	0.086	7.24	21.58	0.876	0.085	7.20
	1024	21.59	0.873	0.093	8.38	21.59	0.873	0.092	8.27
	2048	21.60	0.870	0.099	9.37	N/A	N/A	N/A	N/A

Table A.5: Evaluation metrics for the hotdog dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.1K	32	20.47	**0.858**	**0.079**	4.36	20.47	**0.858**	**0.079**	4.34
	64	20.41	0.856	**0.079**	6.36	N/A	N/A	N/A	N/A
	128	20.24	0.854	0.081	7.16	20.24	0.854	0.081	7.07
	256	20.25	0.853	0.082	8.87	20.27	0.853	0.082	8.79
	512	20.23	0.850	0.085	10.13	20.23	0.850	0.085	9.98
	1024	20.14	0.846	0.090	11.68	20.13	0.846	0.090	11.56
	2048	20.04	0.838	0.103	**13.89**	20.05	0.838	0.102	**13.03**
316.7K	32	**20.50**	**0.858**	**0.079**	4.48	**20.50**	**0.858**	**0.079**	4.51
	64	20.34	0.855	0.080	5.70	20.34	0.855	0.080	5.66
	128	20.23	0.854	0.081	6.87	20.23	0.854	0.081	6.82
	256	20.22	0.853	0.081	8.21	20.18	0.852	0.082	8.03
	512	20.16	0.850	0.084	9.14	N/A	N/A	N/A	N/A
	1024	20.11	0.848	0.086	10.30	20.11	0.848	0.086	10.02
	2048	20.10	0.846	0.090	11.74	20.11	0.846	0.090	11.43
999.9K	32	**20.50**	**0.858**	**0.079**	4.83	**20.50**	**0.858**	**0.079**	4.82
	64	20.36	0.855	0.080	5.53	20.36	0.855	0.080	5.51
	128	20.21	0.853	0.081	6.65	20.21	0.853	0.081	6.59
	256	20.17	0.852	0.082	7.85	20.15	0.852	0.082	7.56
	512	20.09	0.850	0.083	8.49	20.09	0.850	0.083	8.33
	1024	20.03	0.848	0.085	9.33	20.04	0.848	0.086	9.13
	2048	N/A	N/A	N/A	N/A	20.00	0.846	0.088	10.13
3.2M	32	20.15	0.856	0.081	2.32	20.15	0.856	0.081	2.32
	64	20.40	0.855	0.080	4.65	N/A	N/A	N/A	N/A
	128	20.18	0.853	0.081	6.20	N/A	N/A	N/A	N/A
	256	20.11	0.851	0.082	7.36	20.11	0.851	0.082	7.24
	512	20.07	0.849	0.083	7.91	20.04	0.849	0.083	7.86
	1024	19.96	0.847	0.086	8.67	19.96	0.847	0.086	8.53
	2048	19.92	0.845	0.088	9.47	19.92	0.845	0.088	9.33
10.0M	32	20.44	0.857	0.080	2.07	20.44	0.857	0.080	2.07
	64	20.39	0.855	0.080	4.26	20.39	0.855	0.080	4.25
	128	20.17	0.853	0.081	5.59	20.17	0.853	0.081	5.54
	256	20.10	0.851	0.082	6.96	20.11	0.851	0.082	7.00
	512	20.02	0.849	0.084	7.56	20.01	0.849	0.084	7.47
	1024	19.92	0.846	0.086	8.21	19.92	0.846	0.086	8.07
	2048	19.85	0.845	0.088	8.95	N/A	N/A	N/A	N/A

Table A.6: Evaluation metrics for the materials dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.2K	32	21.71	**0.908**	**0.048**	6.51	21.71	**0.908**	**0.048**	6.45
	64	21.70	**0.908**	0.049	11.00	21.70	**0.908**	0.049	10.71
	128	21.69	0.907	0.049	13.59	21.69	0.907	0.049	13.31
	256	21.65	0.907	0.050	15.90	21.65	0.907	0.050	15.13
	512	21.58	0.906	0.053	17.79	21.58	0.906	0.053	17.05
	1024	21.45	0.903	0.058	19.01	21.46	0.903	0.058	18.26
	2048	21.19	0.897	0.069	**20.25**	21.20	0.898	0.070	**19.48**
316.1K	32	21.73	**0.908**	0.049	4.47	21.73	**0.908**	0.049	4.45
	64	21.72	**0.908**	0.049	7.60	21.72	**0.908**	0.049	7.56
	128	21.69	0.907	0.049	13.15	21.69	0.907	0.049	12.50
	256	21.67	0.907	0.049	15.29	21.67	0.907	0.049	14.77
	512	21.65	0.907	0.050	16.76	21.65	0.907	0.050	15.71
	1024	21.61	0.906	0.052	18.16	21.60	0.906	0.052	17.31
	2048	21.53	0.904	0.055	19.15	21.52	0.903	0.055	17.65
999.9K	32	N/A	N/A	N/A	N/A	**21.74**	**0.908**	0.049	4.95
	64	21.73	**0.908**	0.049	9.89	21.73	**0.908**	0.049	9.70
	128	21.72	**0.908**	0.049	12.45	21.72	**0.908**	0.049	11.97
	256	21.70	0.907	0.049	14.63	21.69	0.907	0.049	13.84
	512	21.68	0.907	0.050	16.00	21.68	0.907	0.050	15.22
	1024	21.65	0.906	0.052	17.27	21.65	0.906	0.052	15.89
	2048	21.62	0.905	0.053	18.48	21.62	0.905	0.053	16.87
3.2M	32	**21.74**	**0.908**	0.049	4.65	**21.74**	**0.908**	0.049	4.63
	64	21.72	**0.908**	0.049	9.26	21.72	**0.908**	0.049	9.18
	128	21.71	0.907	0.049	11.86	21.71	0.907	0.049	11.71
	256	21.70	0.907	0.050	14.01	21.70	0.907	0.049	13.48
	512	21.68	0.907	0.050	15.57	21.69	0.907	0.051	14.94
	1024	21.66	0.906	0.052	16.59	21.66	0.906	0.052	15.42
	2048	21.65	0.906	0.052	17.44	21.65	0.906	0.053	16.53
10.0M	32	**21.74**	**0.908**	0.049	3.31	N/A	N/A	N/A	N/A
	64	21.72	**0.908**	0.049	5.95	21.72	**0.908**	0.049	5.91
	128	21.71	0.907	0.049	11.57	21.71	0.907	0.049	11.00
	256	21.70	0.907	0.050	13.60	21.70	0.907	0.049	13.34
	512	21.69	0.907	0.050	15.02	21.69	0.907	0.050	14.61
	1024	21.67	0.906	0.052	16.13	21.67	0.906	0.052	15.89
	2048	21.66	0.906	0.053	14.92	21.66	0.906	0.052	16.26

Table A.7: Evaluation metrics for the mic dataset.

Points	Clusters	SAH				Median			
		PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
100.5K	32	20.95	**0.756**	**0.165**	3.10	20.95	0.756	**0.165**	3.10
	64	20.90	0.755	0.166	3.82	20.90	0.755	0.166	3.81
	128	N/A	N/A	N/A	N/A	20.82	0.752	0.168	4.45
	256	20.68	0.747	0.174	5.41	20.71	0.747	0.174	5.46
	512	20.43	0.737	0.184	6.65	20.38	0.736	0.184	6.70
	1024	19.79	0.719	0.200	8.27	19.79	0.719	0.200	8.23
	2048	18.57	0.689	0.228	**10.41**	18.52	0.689	0.229	**10.34**
317.4K	32	N/A	N/A	N/A	N/A	**20.96**	**0.757**	**0.165**	2.92
	64	20.93	**0.756**	**0.165**	3.49	20.93	0.756	**0.165**	3.49
	128	20.89	0.754	0.166	3.90	20.89	0.754	0.166	3.91
	256	20.83	0.751	0.169	4.62	20.82	0.750	0.170	4.73
	512	20.74	0.745	0.174	5.64	20.74	0.744	0.174	5.53
	1024	20.59	0.736	0.181	6.54	20.60	0.736	0.180	6.48
	2048	20.23	0.721	0.190	7.88	20.25	0.722	0.190	7.73
1.0M	64	**20.96**	**0.756**	0.166	3.25	**20.96**	0.756	0.166	3.25
	128	N/A	N/A	N/A	N/A	20.93	0.755	0.166	3.71
	256	20.88	0.753	0.168	4.28	20.87	0.752	0.170	4.20
	512	20.85	0.749	0.171	4.88	N/A	N/A	N/A	N/A
	1024	N/A	N/A	N/A	N/A	20.79	0.744	0.175	5.47
	2048	20.70	0.737	0.178	6.27	20.68	0.737	0.178	6.27
3.2M	128	20.93	0.755	0.166	3.48	N/A	N/A	N/A	N/A
	256	20.88	0.753	0.169	3.84	20.89	0.753	0.168	3.85
	512	N/A	N/A	N/A	N/A	20.86	0.750	0.171	4.29
	1024	20.82	0.747	0.173	4.69	20.83	0.747	0.173	4.72
	2048	20.79	0.743	0.174	5.27	20.79	0.743	0.174	5.24
10.0M	128	N/A	N/A	N/A	N/A	20.93	0.755	0.167	3.31
	256	20.89	0.754	0.169	3.53	N/A	N/A	N/A	N/A
	512	20.88	0.751	0.171	3.98	N/A	N/A	N/A	N/A
	1024	20.84	0.749	0.172	4.26	N/A	N/A	N/A	N/A
	2048	N/A	N/A	N/A	N/A	20.82	0.746	0.175	4.38

Table A.8: Evaluation metrics for the ship dataset.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-05-30

EXAMENSARBETE Accelerating NeRF-based Rendering Using Bounding Volume Hierarchies
STUDENTER Jonathan Permfors, Daniel Kärde
HANDLEDARE Rikard Olajos (LTH)
EXAMINATOR Michael Doggett (LTH)

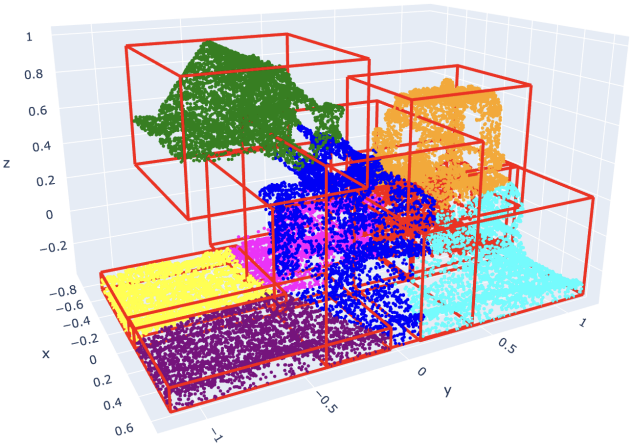
Fotorealistisk grafik blir ännu snabbare

POPULÄRVETENSKAPLIG SAMMANFATTNING **Jonathan Permfors, Daniel Kärde**

Medan datorspel klarar av att rita virtuella 3D-vyer i snabb hastighet, förblir det en utmaning att återskapa 3D-vyer från den verkliga världen. Ny forskning har lyckats uppnå fotorealistiska återskapningar genom att använda artificiell intelligens, men arbete pågår fortfarande för att göra återskapningen snabbare.

Att återskapa 3D-scener från den verkliga världen har flertalet användningsområden, en sådan är artificiell verklighet. Föreställ dig att kliva in i en historisk plats, eller att gå på en virtuell husvisning. Detta har i en lång tid varit en svår uppgift, men kan snart bli mer tillgänglig tack vare en ny teknik som använder sig av artificiell intelligens (AI). De uppnår fotorealism genom att ta 2D-bilder på en vy och träna AI-modellen på dem. Den färdiga AI-modellen kan sedan förutstå färgen vid givna 3D-positioner. Genom att skjuta virtuella strålar genom vyn och förfråga AI:n om färgen vid positioner längs strålen kan nya bilder skapas. Eftersom AI-modellerna är relativt stora blir varje förfrågan kostsam, och det är därför tilltalande att göra detta vid så få positioner som möjligt utan att påverka bildkvaliteten.

Vårt arbete utforskar en metod för att åstadkomma detta genom att kapsla in objekten i vyn i boxar, och enbart förfråga AI-modellen om färg för positioner inom dessa områdena. För att beräkna var strålarna beskär boxarna används en teknik som kallas strålfölning, vilket är ett välutforskat område inom datorgrafik, därmed finns det flertalet tekniker för att påskynda strålfölningen. Vi använder oss av en sådan teknik som arrangerar boxarna i en hierarki. Detta tillåter oss att använda ett mycket större antal boxar och därmed



representera objekten i vyn mer precist, vilket möjliggör ännu färre förfrågningar till AI-modell. Genom denna process uppnår vi en markant snabbare återskapning av vyn, med liten eller ingen effekt på bildkvaliteten i de flesta vyer vi utvärderat.