Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Ray Tracing 3D Gaussians

Matěj Gargula

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
February 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Gargula Matěj** Personal ID number: **492145**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Ray Tracing 3D Gaussians

Master’s thesis title in Czech:

Zobrazování 3D Gaussiánů pomocí sledování paprsku

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **07.02.2025** Deadline for master's thesis submission: ____________

Assignment valid until: **20.09.2026**

____
Head of department’s signature

____
prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

____
Date of assignment receipt

____
Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Gargula Matěj** Personal ID number: **492145**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Ray Tracing 3D Gaussians

Master’s thesis title in Czech:

Zobrazování 3D Gaussiánů pomocí sledování paprsku

Guidelines:

Review methods for scene reconstruction from photographs suitable for subsequent photorealistic synthesis of new views. Focus on the 3D Gaussian splatting [1] and evaluate its available implementation.
Implement a ray tracing-based rendering of 3D Gaussians using a selected framework such as NVIDIA OptiX.
Perform a thorough evaluation of the implementation and identify its bottlenecks. Propose optimizations that will allow efficient ray tracing of very large models with 3D Gaussians. Consider using a hierarchical representation of 3D Gaussians as intersection primitives.
Explore the possibility of incorporating the ray tracing-based rendering into the process of optimizing the parameters of 3D Gaussians using the existing framework [1].
Evaluate the results on at least three different datasets. Compare the results with the splatting-based implementation. Discuss the quality of the rendered images as well as the rendering speed.

Bibliography / sources:

[1] Kerbl, B., Kopanas, G., Leimkühler, T., Drettakis, G. (2023). 3D Gaussian splatting for real-time radiance field rendering. ACM Transactions on Graphics (TOG), 42(4), 1-14.
[2] Kerbl, B., Meuleman, A., Kopanas, G., Wimmer, M., Lanvin, A., Drettakis, G. (2024). A hierarchical 3D Gaussian representation for real-time rendering of very large datasets. ACM Transactions on Graphics (TOG), 43(4), 1-15.
[3] Müller, T., Evans, A., Schied, C. and Keller, A. (2022). Instant neural graphics primitives with a multiresolution hash encoding. ACM Transactions on Graphics (TOG), 41(4), 1-15.
[4] Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., Hedman, P. (2022). MIP-NERF 360: Unbounded anti-aliased neural radiance fields. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 5470-5479.
[5] Moenne-Loccoz, N., Mirzaei, A., Perel, O., de Lutio, R., Martinez Esturo, J., State, G., Fidler, S., Sharp, N., Gojcic, Z. (2024). 3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes. ACM Transactions on Graphics (TOG), 43(6), 1-19.
[6] Gao, J., Gu, C., Lin, Y., Li, Z., Zhu, H., Cao, X., Zhang, L., Yao, Y. (2024). Relightable 3D Gaussians: Realistic point cloud relighting with BRDF decomposition and ray tracing. In Proceedings of European Conference on Computer Vision, 73-89.

Acknowledgements

Declaration

All of the declarations are mentioned on the following page titled Declaration.

Prague, February 7, 2025

Abstract

Rendering techniques have made significant advances in recent decades, transitioning from traditional rasterization to sophisticated volumetric representations. Among these, 3D Gaussian Splatting (3DGS) has emerged as a promising method, enabling real-time rendering of high-fidelity radiance fields by representing scenes as collections of anisotropic Gaussian primitives or particles. This technique could also be expanded upon with the use of modern Raytracing approaches for even greater results.

Keywords: Point-based rendering, 3D Gaussian, Volumetric rendering, Real-time rendering, Scene reconstruction, Rasterization, Raytracing, Spherical harmonical functions, Parameter optimization, Novel View Synthesis

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Praha 2,
Karlovo náměstí,
E-421

Abstrakt

Techniky vykreslování zaznamenaly v posledních desetiletích významný pokrok, přechází od tradiční rastrizace k sofistikovaným objemovým reprezentacím. Mezi nimi se vynořila slibná metoda 3D Gaussian Splatting (3DGS), která umožňuje vykreslování vysoce kvalitních radiančních polí reprezentací scén jako kolekcí anizotropních Gaussiánských primitiv nebo částic a to v reálném čase. Tuto techniku je možné i rozšířit pomocí moderních technik sledování paprsků pro ještě lepši výsledky.

Klíčová slova: vykreslování založené na bodech, 3D Gauss, Volumetrické vykreslování, vykreslování v reálném čase, rekonstrukce scény, Rasterizace, Sledování paprsků, Sférické harmonické funkce, Optimalizace parametrů, Novel View Synthesis

Překlad názvu: Zobrazování 3D Gaussiánů pomocí sledování paprsku

Contents

Figures

Tables

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Gargula Matěj
Personal number:	492145
Programme name:	Open Informatics

declare that I have elaborated the master's thesis entitled

Ray Tracing 3D Gaussians

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I did not use any artificial intelligence tools during the preparation and writing of my thesis. I am aware of the consequences if manifestly undeclared use of such tools is determined in the elaboration of any part of my thesis.

In Prague on 20.05.2025

Bc. Matěj Gargula
....
student's signature

Chapter 1

Introduction

Over the years, many different rendering techniques have been discovered, each best suited for a different task. In scene reconstruction, we have many options to achieve the most realistic scene possible.

To reconstruct a scene, we first need an appropriate representation. The most common representation is the polygonal mesh, which can effectively represent individual objects within the scene and is also easy to render via rasterization or raytracing. Point-based approaches have also been used in the past, but have often suffered a lot of problems. The biggest problem is that, in a point-based representation, we are often missing a lot of information, which creates holes in the reconstructed scenes. Later, these approaches were expanded with splatting methods that were able to remove the hole problems[7].

Rendering techniques have made significant advances in recent decades, transitioning from traditional rasterization to sophisticated volumetric representations. Among these, 3D Gaussian Splatting (3DGS) has emerged as a promising method, enabling real-time rendering of high-fidelity radiance fields by representing scenes as collections of anisotropic Gaussian primitives or particles. This approach offers substantial benefits, including its ability to optimize computational efficiency while maintaining high visual quality, even for complex, unstructured 3D scenes [7].

One of the key innovations of Gaussian Splatting is its adaptability to diverse rendering scenarios, from real-time applications to photorealistic visualizations, as shown in the example rendered scene in figure 1.1. Unlike traditional methods that rely on grid-based voxel structures, Gaussian splatting employs a differentiable representation optimized for scene accuracy and efficiency. Recent advances, such as the integration of ray tracing with the use of acceleration structures (mainly the bounding volume hierarchies or BVH), have further improved its effectivity, enabling advanced visual effects such as reflections, shadows, and depth of field [7] [2] [5].

The aim of this project is to explore the methodology of the Gaussian Splatting methods and their combination with the raytracing algorithm, with the main focus on the real-time rendering aspect.

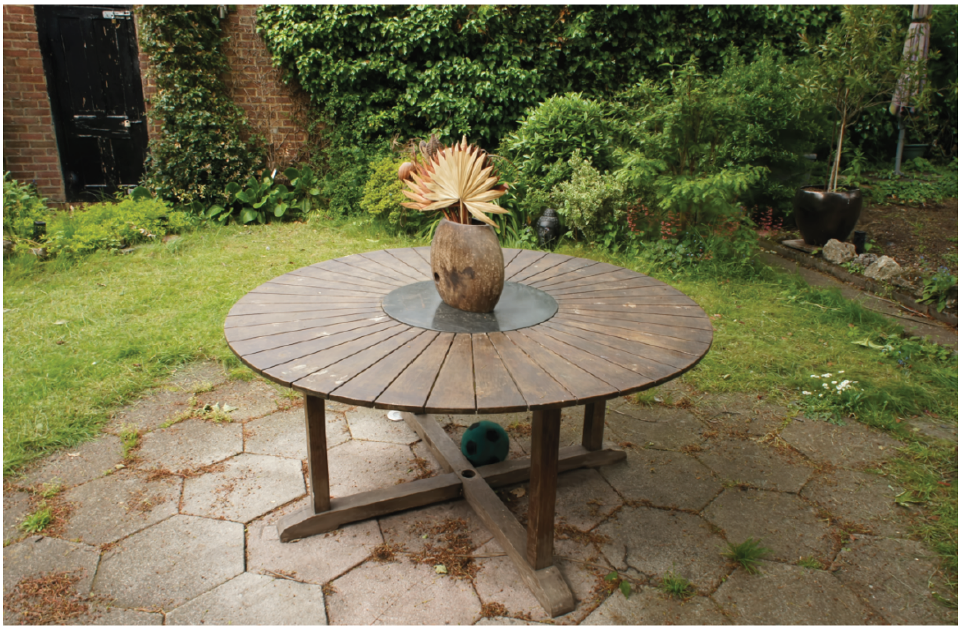


Figure 1.1: Example of a rendered 3D Gaussian scene. [1].

1.1 Differentiable Point-Based Rendering

Differentiable point-based rendering is a framework in which scenes are represented as collections of discrete points or particles, enabling image generation that supports gradient propagation for parameter optimization. This method integrates the principles of point-based rendering with the ability to compute and backpropagate gradients, making it particularly useful for tasks such as Gaussian Splatting [1] [8].

The process begins with representing the scene as a point cloud, where each point is parameterized by attributes like position, color, opacity, and potentially view-dependent properties like spherical harmonic coefficients or different parameters that simulate properties of certain materials. Points are traditionally rendered by projecting them onto the image plane (rasterization) or tracing rays through the scene (ray tracing). The key to differentiable rendering is to ensure that all pipeline stages are smooth and continuous, allowing the computation of gradients with respect to point parameters [1] [5].

To ensure differentiability, functions such as Gaussian kernels or radial basis functions are used to model how each point contributes to the rendered image. These kernels ensure that small changes in parameters lead to smooth variations in the output image, which is essential for gradient-based optimization. Loss functions measuring the discrepancies between rendered and ground-truth images guide the optimization process, enabling the refinement of scene parameters [8] [5].

1.2 Novel View Synthesis

Novel View Synthesis (NVS) is a transformative field within computer vision and graphics, aimed at generating new views of a scene from a set of existing images or 3D data. It has become a crucial tool in various applications, including virtual reality, augmented reality, and 3D content creation. At its core, NVS involves creating realistic renderings of a scene from viewpoints that were not part of the original data, leveraging advances in deep learning and 3D reconstruction.

A prominent approach within NVS is Neural Radiance Fields (NeRF) [9], which represents a scene as a continuous volumetric scene representation. NeRF employs a neural network to predict the color and density at any point in 3D space, allowing for photorealistic novel views with high levels of detail and realism. However, traditional NeRF methods can be computationally expensive and time-consuming, limiting their real-time application [9].

To address this challenge, newer algorithms like MIP-NeRF 360 [10] and Instant Neural Graphics Primitives (Instant NGP) [11] have emerged. MIP-NeRF 360 improves upon NeRF by enabling more efficient rendering of 360-degree scenes, optimizing both the quality and speed of view synthesis. Instant NGP, on the other hand, accelerates NeRF-based methods through novel techniques such as neural networks optimized for graphics primitives, dramatically improving performance without compromising visual quality. These advances have pushed the boundaries of high-fidelity real-time scene rendering, making them key technologies in the future of immersive media and computer graphics [10] [11].

In addition to these approaches, experiments with 3D Gaussians emerged and showed great results in the field of NVS. These experiments introduced rasterization-based projects like 3DGS or the open-source Gsplat library [4], but also other projects that are leveraging the use of raytracing algorithms to generate even better visual results (but at a cost of performance). These methods include projects like Raygauss [6], 3DGRT [5] or Relightable 3D Gaussian [3].

The uses of the 3D Gaussian representation are still being explored in great detail, and we can expect that they can be set as a standard for future NVS methods.

Chapter 2

Gaussian Scene Training

This chapter will focus on discussing the process of training a Gaussian scene. The following sections are mainly focused on discussing important techniques and methods that are used to train and generate a Gaussian scene from the input image dataset.

2.1 Spherical Harmonics and Spherical Gaussian

Spherical harmonics are mathematical functions that provide a basis for representing angular variations on a sphere. These functions are defined on the unit sphere in three-dimensional space and are extensively used in physics, computer graphics, and signal processing due to their efficiency in encoding directional data. Each spherical harmonic Y_m^ℓ is defined by two parameters: degree ℓ and order m. The degree ℓ determines the level of detail captured by the function, while the order m specifies its symmetry on the azimuthal axis. Together, these parameters allow spherical harmonics to represent angular variations ranging from low-frequency, broad features to high-frequency, intricate details [12].

The mathematical form of a spherical harmonic function is described in equation 2.1 [12].

$$Y_m^\ell(\theta, \phi) = N_m^\ell P_m^\ell(\cos\theta) e^{im\phi} \tag{2.1}$$

where θ is the polar angle, ϕ is the azimuthal angle, and N_m^ℓ is a normalization factor that ensures orthonormality. The term $P_m^\ell(\cos\theta)$ represents the associated Legendre polynomial, which encodes the angular dependence in the polar direction. The exponential term $e^{im\phi}$ introduces variation along the azimuthal angle.

A key property of spherical harmonics is their ability to represent functions on a sphere with a finite number of coefficients. By increasing the degree ℓ, spherical harmonics can capture progressively finer details of angular variation. For example, lower degrees represent smooth, diffuse variations, while higher degrees capture sharper, more detailed features. This hierarchical structure makes spherical harmonics particularly efficient for applications requiring compact and accurate representations of directional data [12] [8].

Spherical harmonics are also orthonormal, meaning that they form a complete basis for square-integrable functions defined on a sphere. Any such function can be expressed as a weighted sum of spherical harmonics, where the weights correspond to the projection of the function onto the harmonic basis. This property underpins their utility in a wide range of applications, including global illumination, wave modeling, and signal decomposition.

Spherical harmonics are commonly used in rendering to model directional quantities such as radiance or reflectance. Their compactness and efficiency enable for the accurate representation of angular information while minimizing storage and computational overhead. The smoothness of the spherical harmonics further facilitates interpolation, making them robust in applications involving sparse or noisy data [8] [5] [1].

In practice, spherical harmonics are very good for modeling low frequencies, but they can struggle to represent higher frequencies. Because of that, they are sometimes used in combination with spherical Gaussians.

Spherical Gaussians are an alternative, analytic basis for representing directional variation on the unit sphere, particularly well suited to capturing sharp, view-dependent effects (for example, specular highlights) with very few parameters. Because of that, they are perfect for modeling higher frequencies of an image. Each spherical Gaussian lobe is defined by:

- mean direction p_{lj} (a unit vector on the sphere)
- sharpness λ_{lj} (a scalar controlling how tightly the lobe is peaked around p_{lj})
- an amplitude k_{lj} (a color vector giving its strength)

For any viewing direction d (also a unit vector), the contribution of one lobe is calculated as shown in equation 2.2.

$$e_l j(d) = k_{lj} e^{\lambda_{lj}(d \cdot p_{lj}) - 1)} \tag{2.2}$$

By adding several such lobes (indexed by j), you obtain the high-frequency component of the radiance for a surface or volume primitive, as shown in the equation 2.3.

$$L_{high} = \sum_j e_l j(d) \tag{2.3}$$

Because increasing λ makes the lobe narrower, spherical Gaussians can represent arbitrarily sharp features, but remain analytic and inexpensive to evaluate and integrate.

2.2 Gaussian particle

A Gaussian particle is a mathematical primitive used to represent the spatial and visual properties of a 3D scene. It is an anisotropic volumetric object defined by a set of parameters that describe its position, scale, orientation, transparency, and color. Gaussian particles are the foundation of the 3D Gaussian splatting method.

Each Gaussian particle is mainly defined by its kernel function. The kernel function of a particle describes its contribution to any point $\mathbf{x} \in \mathbb{R}^3$ as described in equation 2.4, where $\rho(\mathbf{x})$ is the kernel function of a particle, $\boldsymbol{\mu} \in \mathbb{R}^3$ describes the center (or mean) of the Gaussian in 3D space, and $\boldsymbol{\Sigma} \in \mathbb{R}^{3\times3}$ is a covariance matrix that defines the shape, scale and orientation of the particle [1] [5].

$$\rho(\mathbf{x}) = e^{-(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{2.4}$$

The covariance matrix $\boldsymbol{\Sigma}$ is represented as described in equation 2.5 [1] [5], where $\mathbf{R}$ is a rotation matrix that defines the orientation of particles in the 3D space and $\mathbf{S}$ is a scaling matrix that describes the size of the Gaussian along its principal axes. This representation ensures that the matrix $\boldsymbol{\Sigma}$ is semidefinite, which is an important property during the scene optimization described in the following sections.

$$\boldsymbol{\Sigma} = \mathbf{R}\mathbf{S}\mathbf{S}^\top\mathbf{R}^\top \tag{2.5}$$

To use the Gaussian particle in rendering, we require additional properties. Most commonly, this includes opacity and radiance function [5].

The opacity σ is a parameter that defines the transparency of the Gaussian particle and determines how much light is transmitted by the particle. High values (close to 1) of opacity σ make the particle behave in an opaque way, meaning that it absorbs or scatters more light. Lower values (closer to 0) indicate that the particle is nearly transparent, allowing most of the light to pass through. The opacity parameter σ also directly affects the transmittance function used in volume rendering, which will be discussed in more detail in the following sections.

The radiance function $\phi_\beta(\mathbf{d})$ models the view-dependent color of the Gaussian particle, accounting for directional lighting effects. The radiance function is defined in equation 2.6 [8], where:

- ℓ is the degree of the spherical harmonic function
- $\mathbf{m}$ is the order of the spherical harmonic function
- $Y_m^\ell(d)$ are the spherical harmonic functions of degree ℓ and order $\mathbf{m}$
- β_m^ℓ are the spherical harmonic coefficients of degree ℓ and order $\mathbf{m}$
- $\mathbf{f}$ is a sigmoid function applied to normalize the radiance, ensuring it remains within a valid range

  With these additional parameters, we are able to represent the view-dependent color of the Gaussian particle. Using spherical harmonics, we are able to simulate a range of different lighting effects, such as shadows, highlights, and reflections.

$$\phi_\beta(\mathbf{d}) = f\left(\sum_{\ell=0}^{\ell_{\max}} \sum_{m=-\ell}^{\ell} \beta_m^\ell Y_m^\ell(\mathbf{d})\right) \tag{2.6}$$

To effectively store the parameters of the Gaussian particle (namely: position(mean), covariance matrix, opacity, and radiance parameters), we can use the following parameters [1] [5] [16].

- Position μ as a 3D vector
- Scale $\mathbf{s}$ as a 3D vector. This will be used to create the scale matrix $\mathbf{S}$.
- Rotation $\mathbf{q}$ stored as a unit quaternion (4D vector). With the rotation and scale information, we can reconstruct the covariance matrix Σ
- opacity σ as a float
- Spherical harmonic coefficients β_m^ℓ as an array of real numbers. The size of the array depends mainly on how many degrees of spherical harmonics we wish to represent. The most common is to store coefficients up to degree $\ell = 3$, which totals 49 individual numbers.
- (Optionally) $\mathbf{p}$ lobe axis of the spherical gaussians
- (Optionally) λ sharpness of the spherical gaussians
- (Optionally) $\mathbf{k}$ color vector of the spherical gaussian
- (Optionally) normal vector N

The parameters p, λ and k are only required if we want to use the spherical Gaussians for better modeling of the higher frequencies, and the normal vector N can be stored if we plan to calculate real-time lighting effects during the rendering process [3] [16].

2.3 Training Overview

The main training approaches have been introduced in the original Gaussian Splatting project (3DGS) and the methods used are also used with the raytracing approaches.

Gaussian Splatting is a form of differentiable point-based rendering. Before we have a nice representation of the scene with our Gaussian particles, we must initialize the scene formed by the particles and perform training. The training method can be split into several phases: Initialization, Adaptive Density Control, Rendering, and Optimization. Excluding the initialization phase, each phase is repeated as shown in figure 2.1 to iteratively refine the representation of the scene given by the set of 3D Gaussians [1], where the optimization phase is executed during the backward flow of the gradients. The high-level overview of when and how each individual phase is executed is also shown in the algorithm 1.

The following subsections discuss each of the important phases in more detail.

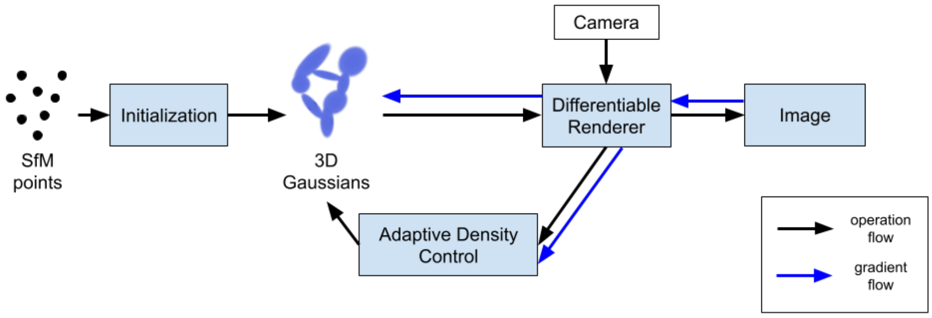


Figure 2.1: Gaussian Splatting method schematic.

How this pipeline functions is also described by the algorithm (1)

Algorithm 1 High-level overview of the Gaussian Splatting training pipeline

```
function TrainGaussianSplatting
    $Points = GetSfMPointCloud()$
    $Gaussians = InitializeGaussians(Points)$
    $iter = 0$

    while $notconverged$ do
        $Camera, Image = SelectRandomTrainingView()$
        $TrainingImage = Render(Gaussians, Camera)$
        $L = Loss(TrainingImage, Image)$
        $Gaussians = OptimizeWithSGD(\Delta L)$

        if $DoAdaptiveDensityControl(iter)$ then
            for all $Gaussian(\mu, \Sigma, c, \sigma) \in Gaussians$ do
                if $\sigma < \epsilon$ or $IsTooLarge(\mu, \Sigma)$ then
                    $RemoveGaussian(Gaussian)$
                end if
                if $CheckOverReconstruction()$ then
                    $SplitGaussian(Gaussian)$
                end if
                if $CheckUnderReconstruction()$ then
                    $CloneGaussian(Gaussian)$
                end if
            end for
        end if

        $iter+ = 1$

        Perform some operation
    end while
end function
```

2.3.1 Initialization phase

In this phase, we mainly process the input images and create an initial set of 3D Gaussians. Here, the Gaussians serve as the primitives for the scene representation. This initial set will optimized in the following phases to ensure it represents the given scene well.

The input images are first processed with Structure-from-Motion (SfM) [1]. The SfM is used here to generate the initial sparse point cloud and a set of camera configurations for each given image. This initial set of points provides a course representation of the given scene. Each point within the point cloud represents the positions (mean) of individual 3D Gaussians. This ensures that the Gaussians are roughly aligned with the visible geometry detected in the scene.

Next, each point in the generated sparse cloud is transformed into a

Gaussian particle with parameters that were discussed in the previous section on Gaussian Particles. To transform a point from the SfM set, we must first know how to choose the parameters. Excluding the position of the particle, we can generate each of the parameters randomly and still get a good result [1]. We can take a different approach if we do not want to start with only random values. Now, each parameter can initialized in a number of ways, but a popular way sets individual parameters as follows [8] [1] [5].

- The position (mean) μ is directly mapped from the positions of the SfM points. During mapping, we can also filter out outlier points from the point set to improve the initial set of Gaussian particles
- Scale $\mathbf{s}$ can be initialized with the information about neighboring points in the point cloud. For each particle, we compute the distances to the 3 closest neighbors in the point cloud for each point. We then use these three distance values to set the scales of the Gaussian particle
- rotation $\mathbf{q}$ is initialized to identity, assuming no orientation preference at the start.
- opacity σ can be initialized with a simple default value, which is often in the range of 0.5 to 1. Since the opacity is restricted by the sigmoid function, the opacity is constrained to values between 0 and 1.
- The spherical harmonic coefficients β_m^ℓ can still utilize data from the point cloud. For the zero-order coefficients, we can use the color of the points (if they are included) as a starting point. We can either initialize the higher-order coefficients with zero values or generate random numbers drawn from the standard normal distribution.

2.3.2 Rendering

The rendering phase is responsible for transforming the 3D Gaussian-based representation of a scene into a 2D image, accurately capturing the spatial and visual properties of the input data. This phase evaluates the contribution of each point in the scene to the pixels of the final image, using methods that ensure an accurate representation of the scene's geometry, appearance, and visibility. The rendering process is the most important phase in the pipeline since its output is used to generate the gradients that are used to improve the scene iteratively [1] [5].

Rendering involves computing how each point influences the image based on its spatial attributes (position, scale, and rotation) and appearance attributes (radiance, opacity). Two primary rendering techniques are commonly employed:

- Rasterization: Points are projected onto the 2D image plane, where their contributions to pixel values are determined through compositing techniques such as alpha blending.

- Ray tracing: Rays are cast through the 3D scene from the camera's perspective, and each ray interacts with the points to compute their contributions by treating each particle as a volume that must be sampled. Because of that, we need each individual sample to be sorted by depth.

The rendering phase also requires a camera configuration as a parameter. This is because we must align the output image with the ground-truth image to calculate the loss function, and we also require the correct projection matrix to perform the 2D projection.

Also, a very important step during the rendering of the scene is to identify which Gaussian particles have influenced each pixel in the final rendered output. Because of that, we need to maintain a list that will represent which pixels are influenced by which Gaussians. A different approach is to perform a second simpler backward pass, identifying which Gaussians are associated with each pixel. These lists are important during the optimization step when we optimize the parameters of the individual Gaussians.

The following chapter expands on the rendering techniques that can be used with the Gaussian splatting method.

2.3.3 Optimization

The optimization phase refines the Gaussian representation by iteratively comparing the rendering output to the ground truth. Errors caused by ambiguities in projecting 3D geometry to 2D can result in misaligned or redundant Gaussians, requiring particle addition, removal, or repositioning. The addition and removal of the particles is processed during the Adaptive Density Control phase. Covariance parameters are essential for achieving a compact representation, as large homogeneous regions can be efficiently represented using fewer anisotropic Gaussians [1].

Stochastic Gradient Descent (SGD) is used to adjust the parameters of the 3D Gaussians. Gradients of a loss function, which measures the difference between the rendered and target images, are computed for parameters such as position, covariance, opacity, and spherical harmonic coefficients for radiance. SGD processes mini-batches of rays or pixels, making it computationally efficient for large datasets. GPU-accelerated frameworks can be utilized to calculate gradients in parallel, speeding up the process. An exponential learning rate decay is applied to make large updates early in the optimization and smaller and more precise updates as it progresses [1].

The opacity parameter σ is limited to the range [0,1) using a sigmoid activation function, which ensures smooth gradient calculations. Covariance scales are constrained with an exponential activation function to keep them positive. The initial covariance matrices are set as isotropic Gaussians, with axes equal to the mean distance to the three closest points [1] [5].

The loss function, defined in equation 2.7, combines the loss of L_1 for the differences at the pixel level and L_{D-SSIM} for the structural similarity. The weight between these terms is controlled by the parameter λ , typically set to $\lambda = 0.2$ [1] [4].

$$L = (1 - \lambda)L_1 + \lambda L_{\text{D-SSIM}}, \tag{2.7}$$

After we evaluate the loss function by comparing the rendered image with the ground-truth image, we can calculate the loss gradients for each pixel. We then use this gradient and perform backward pass, propagating the loss gradients and calculating the color and parameter gradients for each particle used in the point cloud. The backward pass can be processed only on depth-sorted particles that have been stored during the forward rendering phase. Another way is to render the scene again, but during the rendering process we can calculate (or incrementally accumulate) gradients for each particle that participates in the final image. This approach seems to be more common and is mainly used in raytracing approaches [5] [6].

When we have the gradients calculated, we can optimize the parameters of the particles that have participated in the rendering process with the SGD [1].

2.3.4 Adaptive Density Control

Adaptive Density Control is a process used in Gaussian Splatting to dynamically adjust the number and distribution of Gaussians during optimization. Starting with the initial set of Gaussian Particles from the Initialization phase, the method applies several techniques to control the density of the particles: Densification, Splitting, and Pruning [1].

Densification is applied to regions with insufficient Gaussian representation, often due to under-reconstruction, shown in Figure 2.2. Gaussians in these areas are cloned by duplicating their parameters and moving the clones in the direction of the positional gradient. This process improves the coverage of fine details and missing geometry. Cloned Gaussians retain the size and other attributes of the original particle.

The splitting is performed in overreconstructed regions shown in Figure 2.2, where large Gaussians cover significant portions of the scene. Each large Gaussian particle is replaced by two smaller particles, and the scale of the original Gaussian is divided by a factor of 1.6, determined experimentally [1]. The positions of the new Gaussians are sampled using the original Gaussian as a probability density function. This process maintains the total density of the representation while improving its granularity [1].

To remove redundant particles, pruning is performed on Gaussian particles with negligible opacity (meaning the σ is under a specific threshold). Additionally, large Gaussians that overlap excessively with others are removed to maintain an optimal density in both world and view space. This step ensures computational efficiency and prevents excessive memory usage during the rendering phase [1].

This phase does not have to be executed in every iteration of the training, but rather every i-th iteration is preferable, where, for example, $i = 100$ [1].

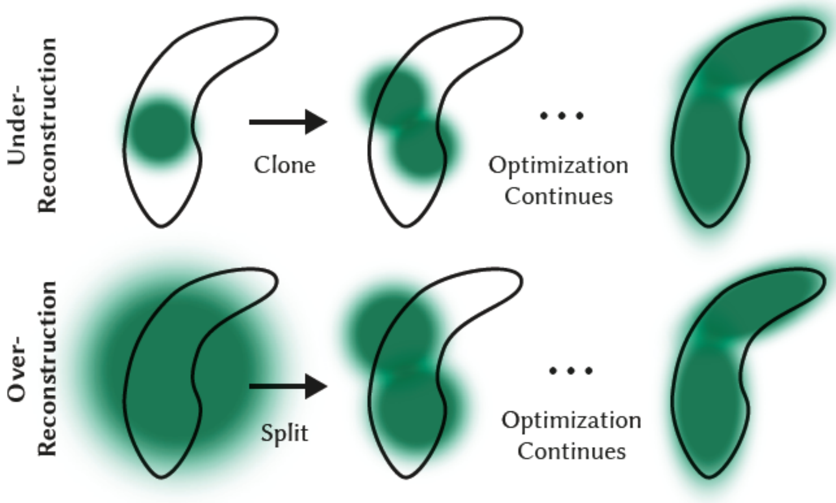


Figure 2.2: This figure shows how the Adaptive Density Control functions. If we detect a under-reconstruction, we clone particles to cover more area and continue to optimize the scene. If over-reconstruction is detected we split particles to cover less area and again continue with the optimization. [1]

Chapter 3

Rendering 3D Gaussians

Figure 3.1 shows a more detailed view of the training pipeline. In it we can see in what sections we perform the rendering passes and what are their respective outputs.

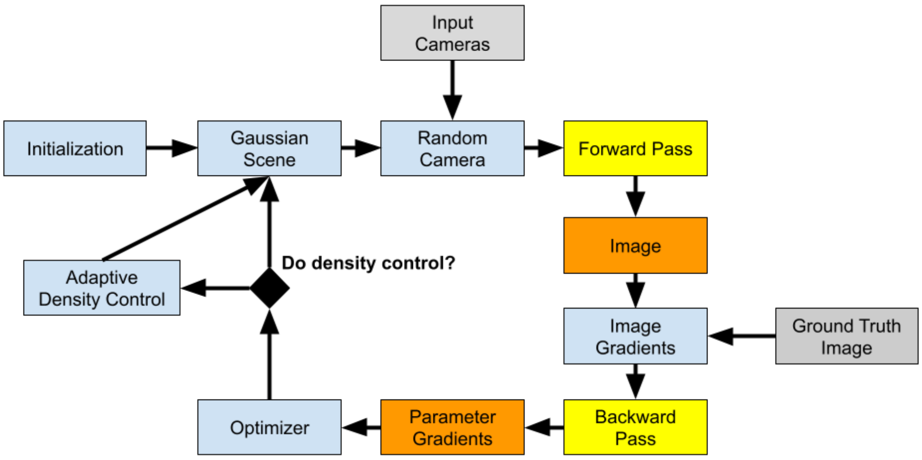


Figure 3.1: This figure shows a more detailed scheme of the training pipeline. The yellow boxes symbolize the phases of the pipeline where rendering takes place. The orange boxes then show the what is the output of the given rendering pass(For forward it is a color image and Bacwards pass calculates a list of particle gradients).

This chapter will mainly focus on the differential rendering techniques used to visualize the scene represented by a set of 3D Gaussian particles in real time. This will expand the topic of rendering discussed briefly in the previous chapter. The following sections will explore Rasterization, Raytracing, and acceleration of the rendering computation. Note that this will focus mainly on the forward pass of the rendering algorithms as the backward pass should always render the scene in the same way, with the difference of calculating the gradients for the individual particle parameters.

3.1 Radiance Color Evaluation

Calculating radiance from spherical harmonics can be a very expensive operation. Because of that, we want to be able to calculate the radiance color in an efficient way by pre-computing some of the values that are required by the computation as shown in equation 2.6 [8] [12]. An efficient way to evaluate the radiance function is shown in the algorithm 2. The constant variables C_i are precomputed explicit spherical harmonic polynomial basis functions. These can be either precomputed or, to save time, even set as constants in implementations [1] [5] [8]. The input of the $GetRadianceColor$ function is the degree of spherical harmonics that we want to compute, the coefficients of the spherical harmonics β_l^m, here called shc, and the direction from which we want to evaluate the radiance function. The coefficients shs are represented as a set of sixteen 3D vectors, where each element of the vector is the coefficient for a single color channel in the RGB representation. The algorithm shown is capable of evaluating spherical harmonics up to a degree of 3, commonly the maximum degree that is used during rendering [1] [5].

Additionally, we can also use the spherical Gaussians to model higher frequencies (such as specular reflections). This can be achieved very easily by computing the radiance for the lower frequencies L_{low} with spherical harmonics and then combining them with the calculated radiance for the higher frequencies with a spherical Gaussian L_high. To obtain the final radiance value, we can simply combine the lower and higher radiance calculations L_{low} and L_{high} as shown in equation 3.1 ([17] [6]).

$$L = L_{low} + L_{high} \tag{3.1}$$

Note that if we choose to use this approach of combining lower and higher frequency calculations we will have store more data for each particle to represent the spherical gaussian lobe-axis, sharpness and color amplitude.

Algorithm 2 Evaluation of the SH Radiance Function

```
function GetRadianceColor(degree, $shc$, dir)
    $(x, y, z) = dir$
    $Y_0^0 = C_0[0]$
    $RadianceColor = Y_0^0 * shc[0]$

    if $degree > 0$ then
        $Y_1^{-1} = C_1[0] * y$
        $Y_1^0 = C_1[0] * z$
        $Y_1^1 = C_1[0] * x$
        $RadianceColor+ = Y_1^{-1} * shc[1]$
        $RadianceColor+ = Y_1^0 * shc[2]$
        $RadianceColor+ = Y_1^1 * shc[3]$
        if $degree > 1$ then
            $xx, yy, zz = x * x, y * y, z * z$
            $xy, yz, xz = x * y, y * z, x * z$
            $Y_2^{-2} = C_2[0] * xy$
            $Y_2^{-1} = C_2[1] * yz$
            $Y_2^0 = C_2[2] * (2zz - xx - yy)$
            $Y_2^1 = C_2[3] * xz$
            $Y_2^2 = C_2[4] * (xx - yy)$
            $RadianceColor+ = Y_2^{-2} * shc[4]$
            $RadianceColor+ = Y_2^{-1} * shc[5]$
            $RadianceColor+ = Y_2^0 * shc[6]$
            $RadianceColor+ = Y_2^1 * shc[7]$
            $RadianceColor+ = Y_2^2 * shc[8]$
            if $degree > 2$ then
                $Y_3^{-3} = C_3[0] * y * (3xx - yy)$
                $Y_3^{-2} = C_3[1] * z * xy$
                $Y_3^{-1} = C_3[2] * y * (4zz - xx - yy)$
                $Y_3^0 = C_3[3] * z * (2zz - 3xx - 3yy)$
                $Y_3^1 = C_3[4] * x * (4zz - xx - yy)$
                $Y_3^2 = C_3[5] * z * (xx - yy)$
                $Y_3^3 = C_3[6] * x * (xx - 3yy)$
                $RadianceColor+ = Y_3^{-3} * shc[9]$
                $RadianceColor+ = Y_3^{-2} * shc[10]$
                $RadianceColor+ = Y_3^{-1} * shc[11]$
                $RadianceColor+ = Y_3^0 * shc[12]$
                $RadianceColor+ = Y_3^1 * shc[13]$
                $RadianceColor+ = Y_3^2 * shc[14]$
                $RadianceColor+ = Y_3^3 * shc[15]$
            end if
        end if
    end if

    return $max(RadianceColor, 0)$
end function
```

3.2 Differential Rasterization

Rasterization in Gaussian Splatting is the process of projecting 3D Gaussian primitives onto a 2D image plane and blending their contributions to generate a rendered image. As discussed in the previous chapter, each Gaussian particle is represented by its position, scale, rotation, opacity, and radiance, which are transformed into screen space during rasterization. Unlike traditional rendering techniques that rely on discrete geometry, Gaussian Splatting uses continuous volumetric primitives, enabling smooth and differentiable rendering. This process is important in creating visually accurate images and supporting optimization tasks in 3D scene reconstruction. By handling anisotropic shapes and overlapping contributions, the rasterization method ensures that the spatial and visual properties of the scene are preserved in the final output.

The first step in the rasterization of Gaussian particles is the projection from 3D representation to 2D. Thankfully, the particles can be easily projected into the 2D screen space. We start by projecting the particle's 3D position (mean) and obtaining 2D screen coordinates using the camera projection matrix. This projection is shown in equation 3.2, where $\mu \in \mathbb{R}^3$ is the position of the particles, $\mu' \in \mathbb{R}^3$ is the transformed position on the screen space (in homogeneous coordinates) and $W \in \mathbb{R}^{4\times 4}$ is the projection matrix of the camera created with the combination of the projection perspective and the viewing matrix [1] [4] [8].

$$\mu' = W\mu \tag{3.2}$$

Next, we are also required to transform the covariance matrix, which defines the particle's anisotropic shape by retaining information about the particle's size and orientation. The transformation of the covariance matrix is shown in equation 3.3, where $W' \in \mathbb{R}^{3\times 3}$ is the submatrix without translation of the camera projection matrix, $J \in \mathbb{R}^{2\times 3}$ is the Jacobian of the Affine projection, $\Sigma \in \mathbb{R}^{3\times 3}$ is the original covariance matrix, and $\Sigma' \in \mathbb{R}^{2\times 2}$ is the newly transformed covariance matrix that defines the elliptical spread of the Gaussian in the 2D screen space. [1]

$$\Sigma' = JW'\Sigma W'^{\top} J^{\top} \tag{3.3}$$

Once we have the position, along with the covariance matrix, transformed into the 2D screen space, we can calculate the influence of a particle for any given point x with the Gaussian kernel. The kernel use is similar to the one shown in equation 2.4, but uses the transformed parameters of the Gaussian particle. The modified kernel function ρ' is shown in equation 3.4 with the transformed parameters μ' and Σ' [1].

$$\rho'(x) = e^{-\frac{1}{2}(x-\mu')^{\top}\Sigma'^{-1}(x-\mu')} \tag{3.4}$$

To test whether a pixel can be influenced by the Gaussian particle, we can employ different tactics.

Firstly, we want to perform frustum culling to remove any unnecessary particles that have an insignificant influence on the screen's pixels.

To filter out pixels that the particle cannot influence in a significant way, we can utilize a bounding volume, such as a simple AABB, and project it to the screen space. Now, only pixels within the bounding volume's boundaries are processed further.

Next, the covariance matrix defines the elliptical spread of the particle. With that, we can check to see if a pixel lies in the spread of the particle. If the pixel is outside the particle's spread, we can disregard it because its impact on the image would be insignificant. We can also specify a confidence interval (for example, 99%) to define an effective range of Gaussian [1], again disregarding any pixels that would be outside this interval.

The next step in the rasterization is the evaluation of the color of a pixel. Since the average scene contains a lot of different Gaussian particles, we need an efficient way to calculate the pixel output color with the influence of multiple particles. This is done using the alpha blending technique shown in equation 3.5, where N is the number of Gaussian particles that influence the pixel, α_i is the blending value of the i-th Gaussian and c_i is the Gaussian radiance color. The blending opacity value of the i-th Gaussian is calculated using the particle's kernel function and the set opacity parameter σ. This calculation is shown in equation 3.6.

$$C(x) = \sum_{i=1}^{N} \alpha_i c_i \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{3.5}$$

$$\alpha_i = \rho(x)' * \sigma \tag{3.6}$$

During alpha-blending, we accumulate the opacity α of the pixel. If the accumulated opacity exceeds one, we can terminate the calculation since it is the largest possible value.

We must process the individual Gaussians that affect a pixel in the correct depth order to use the alpha blending technique. Because of that, we have to presort the Gaussians by depth before performing the alpha-blending step. Also, as mentioned in the previous chapter, we need to record each pixel that this Gaussian influences for the optimization pass. However, we can skip this step if we only want to render an image from a certain view that will not participate in the training pipeline. The final output of the alpha-blending step is the resulting color of the image [1] [2].

3.2.1 GPU Differential Rasterization

To achieve real-time rendering results, an optimized GPU implementation is required. So far, the most common way to optimize the Gaussian Splatting rasterization process is to implement a tile-based rasterizer [1], [4].

Firstly, the screen is separated into small tiles (typically 16x16 [1] [4]), and a thread block is executed for each tile on the screen. Then, the culling of particles is performed against the frustum and individual tiles. For culling, the Gaussians are represented by a simple box that is stretched to cover the extent of the Gaussian. We also often reject particles that are too close to the near plane or to the far plane of the camera (By checking the position/mean of a Gaussian). Then, we instantiate each Gaussian according to the number of tiles they overlap and assign each instance a key that contains their distance to the camera and the ID of a tile. Then, for each tile, we want to sort the individual Gaussians by depth [4]. This can be done by any fast GPU sorting algorithm, such as a GPU version of a radix sort [1]. This approximation provides a much faster version of sorting than sorting Gaussians per screen pixel and does not generate significant errors or artifacts during the training process [1]. Then, each tile loads its packet of Gaussians into the shared memory for faster memory access and starts processing the Gaussians for each pixel front-to-back. During processing, the alpha blending process (as shown in equation 3.5) is executed. This process can be stopped early if the saturation (opacity) α of a pixel reaches $\alpha = 1$. After this process, the rasterization is complete, and the final color is stored in the frame buffer [1] [4]. A high-level overview of this method is shown in algorithm 3.

Algorithm 3 High-level overview of the Tile-based rasterization pipeline [1]

```
function RASTERIZE(Gaussians, View, Width, Height)
    $Image = InitializeBlankImage(Width, Height)$
    $Tiles = Create16x16Tiles(Width, Height)$

    $Gaussians = CullGaussians(Gaussians, View)$
    $ProjectedGaussians = ProjectTo2D(Gaussians, View)$

    $TileGaussians, Keys = InstantiateTileGaussians(ProjectedGaussians)$

    $SortByDepthGlobally(Keys)$

    for all $Tile \in Tiles$ do
        for all $Pixel \in Tile$ do
            $I[Pixel] = BlendInOrder(Pixel, TileGaussians, Keys)$
        end for
    end for
    return $Image$
end function
```

3.3 Differential Ray Tracing

A raytracing approach requires a different set of techniques than rasterization, but can simplify some of the rasterization issues. It is also important to note that raytracing is generally slower than rasterization [5], but it is able to produce better results with sometimes smaller scene size.

With a raytracing approach, we calculate the radiance for a given pixel by solving the radiative transfer equation 3.7 [6], which models the variation in radiance as it travels through an infinitely small volume at a given position and direction. $L(x, \omega)$ is the radiance at point x traveling in direction ω, $\sigma(x)$ is the density (or absorption) at position x and $c(x, \omega)$ is the radiance emitted at x in direction ω. The first term of equation 3.7 models the loss of radiance by absorption, and the second term models the emission at the given point and direction.

$$(\omega \cdot \Delta)L(x, \omega) = -\sigma(x)L(x, \omega) + \sigma(x)c(x, \omega) \tag{3.7}$$

By solving the differential equation 3.7 we get equation 3.8. Here we calculate the resulting radiance by integrating the path to x which is defined by separate points y defined in the equation 3.9. The variable $T(x, y)$ is defined in the equation 3.10 and represents the transmittance between the points x and y that is widely used in volumetric rendering [6].

$$L(x, \omega) = \int_0^\infty c(y, \omega)\sigma(y)T(x, y)dy \tag{3.8}$$

$$y = x + t\omega \tag{3.9}$$

$$T(x, y) = e^{-\int_0^t \sigma(x - s\omega)ds} \tag{3.10}$$

Since we want to utilize ray tracing to integrate this radiance, we can parameterize the equation 3.8 with the ray parameter $r(t) = o + t\omega$ as shown in equation 3.11 and 3.12 [6].

$$L(r) = \int_0^\infty c(r(t), \omega)\sigma(r(t))T(t)dt \tag{3.11}$$

$$T(t) = e^{-\int_0^t \sigma(r(s))ds} \tag{3.12}$$

This approach is used as the basis for radiance calculations in modern novel view synthesis algorithms. Different approaches can instead calculate this radiance via a more classical volume rendering approach, which is shown in equations 3.13 and 3.14

$$L(r(t)) = \int_{t_f}^{t_n} T(r(t)) \left(\sum_i (1 - e^{-\sigma_i(r(t))}) c_i(d) \right) dt \tag{3.13}$$

$$T(r(t)) = e^{-\int_t^{t_n} \sum_i \sigma_i(r(t))\, dt} \tag{3.14}$$

In practice, when we wish to evaluate the radiance of a ray r we use a discretized version shown in equations 3.15. This version of the equation allows us to combine individual samples to evaluate the radiance at a given point in a given direction. Note that with this approach, the individual samples must be sorted by depth in a front-to-back order (closest to furthest) [5].

$$L(r) = \sum_{i=0}^{N}(c_i(r)\sigma_i(r)\prod_{j=1}^{i-1}(1-\sigma_j(r)) \tag{3.15}$$

In some cases, we might want to utilize an approach similar to ray marching and sample the scene with a uniform step size Δt. This approach has been used in similar algorithms such as NeRF(Neural Radiance Field). In these cases, we can combine the samples differently as shown in the equations 3.16 and 3.17, where σ_i is the density of the i-th particle, Δt is the fixed step size of the sampling process.

$$L(r) = \sum_{i=1}^{N}(1-e^{-\sigma_i \Delta t})c_i T_i \tag{3.16}$$

$$T_i = e^{-\sum_{j=0}^{i-1}\sigma_j \Delta t} \tag{3.17}$$

The overview of the raytracing approach in practice is as follows. We first want to shoot rays from our camera. Next, we have to perform a test to see which particles have been intersected by the ray and store their references in a list. To be able to use these stored particles, we have to determine where the sample point will be located. Then, similarly to the rasterization, we want to sort each individual sample point along the ray by its depth. After that, we evaluate the radiance and density of each Gaussian and blend them in the sorted order (unless they are already sorted). There are multiple different approaches to evaluating the final radiance, as shown in the equations above. One of the main things that also differs is the way to sample the 3D Gaussian particles along the ray, how many times, and at which locations [5] [6].

But to be able to perform any kind of sampling, we need a way to accurately determine when a ray has intersected the 3D Gaussian particle. 3D Gaussians are elliptically shaped. Because of this, we can calculate the intersection of a particle as a ray-ellipsoid intersection. A standard 3D ellipsoid can be defined by its position P, scale matrix S, and a rotation matrix R. Then we can use the world-space ray representation $r(t) = o_w + t * \omega_w$ with a defined minimum and maximum possible length of the ray, so $t \in [t_{min}, t_{max}]$. To calculate the exact intersection we must first map the ray into the "unit-sphere" frame of the ellipsoid. To perform this mapping we first construct the matrix M shown in equation 3.18. We can then use this matrix to transform the ray into the local frame, where the ellipsoid becomes a unit sphere $||x|| \leq 1$ and the ray becomes $r(t)_l = o_l + t\omega_l$. This is achieved by calculating the o_l and ω_l, which are shown in equations 3.19 and 3.20 respectively [6].

$$M = R^T S^{-1} \tag{3.18}$$

$$o_l = M(o_w - \mu_w) \tag{3.19}$$

$$\omega_l = \frac{M\omega_w}{||M\omega_w||} \tag{3.20}$$

With this, we can calculate and evaluate the intersection. There are two common approaches. The easiest and most straightforward way to calculate the intersection distance t is to solve equation 3.21, which can be rearranged to equation 3.22, which is a simple quadratic equation. If we solve the equation for t we can get two, one, or no intersection points.

$$||o_l + t\omega_l||^2 = 1 \tag{3.21}$$

$$t^2 + 2(o_l\omega_l)t + (||o_l||^2 - 1) = 0 \tag{3.22}$$

The second common approach is more numerically stable. Here we can project the unit sphere center (transformed ellipsoid) onto the ray to get the scalar b as shown in equation 3.23, where b is the signed distance from the ellipsoid-space ray origin to the point of closest approach on the infinite line defined by our ray. Then we calculate the scalar c (shown in equation 3.24), which shows how far outside ($c > 0$) or inside ($c < 0$) the ray origin lies relative to the sphere. Then we calculate the actual projection point p, shown in equation 3.25 that minimizes the distance to the center of the sphere. Then we calculate the discriminant D (shown in equation 3.26). With the discriminant calculated, we can determine if the ray has hit the unit sphere or not. If $||p||^2 > 1$, the ray missed, and if $||p||^2 \leq 1$, then the ray has hit the sphere, and we can calculate the distances t_1 and t_2 to the intersection points. The calculation for t_1 and t_2 is shown in equations 3.27 and 3.28. Note that the distances t_1 and t_2 are multiplied by $\frac{1}{||\omega_l||}$ to transform them back from the local frame.

$$b = -o_l\omega_l \tag{3.23}$$

$$c = ||o||^2 - 1 \tag{3.24}$$

$$p = o_l + b\omega_l \tag{3.25}$$

$$D = 1 - ||p||^2 \tag{3.26}$$

$$t_1 = (\frac{c}{b + sign(b)\sqrt{D}})\frac{1}{||\omega_l||} \tag{3.27}$$

$$t_2 = (b + sign(b)\sqrt{D})\frac{1}{||\omega_l||} \tag{3.28}$$

With the t_1 and t_2 calculated, we can check if $t_{1,2} \in [t_{min}, t_{max}]$ and report if an intersection has occurred along with the $t_{1,2}$ values or if the ray missed the ellipsoid [6].

The following sections discuss three different approaches and how a scene should be initialized to be able to collect samples efficiently.

3.3.1 Initialization of the Scene

Just like in the rasterization step, we start by taking the camera configuration of one of the training views as input for the rendering. With this input, we can set up our camera to determine the origin and direction of our rays. If we would like to simply view the trained scene, we can use any custom camera configuration to view the scene from any angle [1] [5] [4] [6].

Then, an initialization of our scene has to take place. We already have 3D Gaussians represented as volumetric particles, which have been discussed in the previous chapter, but we require a simple way to represent the boundaries of the particles. This is done by encapsulating each particle with a Bounding Primitive. There is a wide range of primitive types that can be used. Some examples of common primitives are shown in figure 3.2 [5] [3]. Each of the bounding primitives comes with a trade-off between the speed of intersection computation and the tightness of the boundaries, where spheres and axis-aligned bounding boxes (AABB) offer very fast intersection test but do not offer good tightness around the particle, causing it to take into account unnecessary intersection which not have a significant impact on the rendered image. On the other hand, custom triangle meshes such as the icosahedron or the stretched icosahedron, offer much better tightness and particle representation but require more costly intersection tests [5].

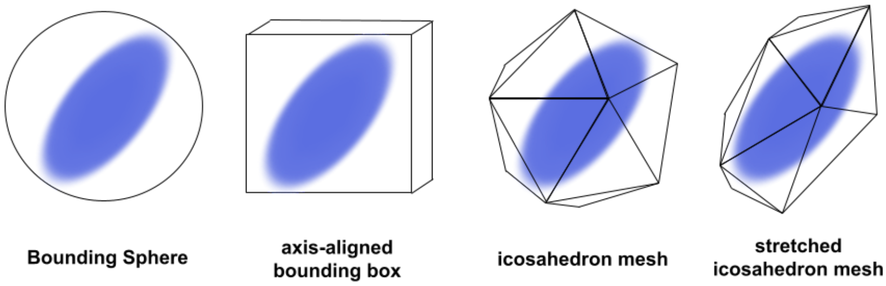


Figure 3.2: Example of Bounding Primitives.

With the boundary representation of the particle, we can build an acceleration structure on our scene. The acceleration structure is a key component in the raytracing approach if we wish to achieve a real-time rendering with a large number of particles. The most commonly used acceleration structure is the bounding-volume hierarchy (BVH), which builds a hierarchy of bounding volumes over the particles. This enables us to quickly find particles that can intersect a ray and filter out particles that will never be intersected. After this test we can use the exact intersection calculation.

3.3.2 GPU Raytracing

To produce real-time rendering results, we want to utilize GPU hardware acceleration. Modern GPUs already offer hardware acceleration options, which are designed to be used with the raytracing method. This hardware acceleration can be used more easily with frameworks such as NVIDIA OptiX or Vulkan. These frameworks enable us to create rendering pipelines, which often contain several programmable entry points. We can create our own custom shader program, which will often be executed at these or similar entry points: [5]

- **ray-gen program**: At this entry point, we initialize our scene and camera configurations, create the initial rays, and start their traversal of the scene
- **intersection program**: This program is called during the traversal of the scene to precisely compute the intersections with the scene objects
- **any-hit program**: This program is called every time a scene object hit is recorded, and we may choose to stop the traversal of the scene or ignore the intersection to continue traversing.
- **closest-hit program**: Unlike Any-hit, this program is called only at the end of the traversal to process the closest hit recorded.
- **miss program**: Here, we can decide what should be done at the end of a traversal when no hit has been recorded.

This type of pipeline provides a very powerful set of tools for the raytracing method and is highly optimized for use in rendering opaque objects. But this also means that the pipeline performs the best when there is a low number of recorded hits in the scene during the traversal. Sadly, this is not the case when we want to render a large set of transparent volumetric particles [5] [3] [6].

To fully utilize the GPU acceleration support, we will mainly focus on the ray-gen and any-hit programs (optionally, closest-hit programs), as they are the most important parts of rendering transparent objects. In the ray-gen program, we want to start traversing the scene and gather all intersections along the ray in the any-hit program to compute their color and contributions to the final output [5] [6].

3.3.3 Uniform Slab Size Volume Raytracing

The method of using raytracing on a scene composed of 3D Gaussian particles has multiple different approaches. In this approach, we can see the Gaussian particles as emissive and absorbing entities. A Gaussian scene is then simply a collection of these entities. These entities can then be defined by their density (absorption) σ and their emission c. The density σ shown in equation 3.29 is defined as a product of the maximum possible density σ^{max} and a basis function $\psi(x)$, where $0 < \psi(x) < 1$. The basis function ψ in this context is a weighting function that defines how each primitive contributes to its density. For this reason, we can use the Gaussian particle kernel function ρ (shown in equation 2.4) as the basis function in the density calculation.

$$\sigma(x) = \sigma^{max}\psi(x) \tag{3.29}$$

The emission c of the entity depends only on the direction of emission d and not on the position x as shown in equation 3.30. With this representation, we have a defined collection of entities whose appearance changes depending on the viewing direction. This allows us to use the Gaussian particle's radiance function $\phi(d)$ (equation 2.6) as a way to compute the radiance emitted by the entity since the radiance function

$$c(x, d) = c(d) \tag{3.30}$$

To utilize this aspect of the entity representation, we can use a modified version of the radiative transfer equation (shown in equation 3.7) for volumes consisting of N independent entities with absorption and emission properties. This differential equation is shown in equation 3.31, where $(d \cdot \Delta)L(x, d)$ is the rate of change of radiance, σ_i is the density, and c_i is the emission of the i-th entity [6].

$$(d \cdot \Delta)L(x, d) = -(\sum_{i=1}^{N} \sigma_i(x))L(x, d) + \sum_{i=1}^{N} \sigma_i(x)c_i \tag{3.31}$$

If we compare equation 3.31 to the original equation 3.7, we can define a volume with a global density function $\sigma_g(x)$ (equation 3.32) and the global emission function $C(x, d)$. This formulation gives us a way to calculate the radiance at a certain position as the weighted sum of the densities and emissions of the entities [6].

$$\sigma_g(x) = \sum_{i=1}^{N} \sigma_i(x) = \sum_{i=1}^{N} \sigma_i^{max}\psi_i(x) \tag{3.32}$$

$$C(x, d) = \frac{\sum_{i=1}^{N} c_i(d)\sigma_i^{max}\psi_i(x)}{\sum_{i=1}^{N} \sigma_i^{max}\psi_i(x)} \tag{3.33}$$

In practice, we have a set of our Gaussian particles that can be used with the representation used in equations 3.32 and 3.33. But to be able to evaluate these equations we first need to decide where are we going to sample them. The most straightforward way is to utilize the volumetric ray marching approach and set a fixed step size to sample the intersected particles along the ray in uniform intervals which is shown in Figure 3.3 with the slabs containing six sample points each [6].

Individual Gaussian scenes can be very large, and it would not be very efficient to evaluate the particle density for each particle intersected by the ray because more distant particles would have minimal impact on the final

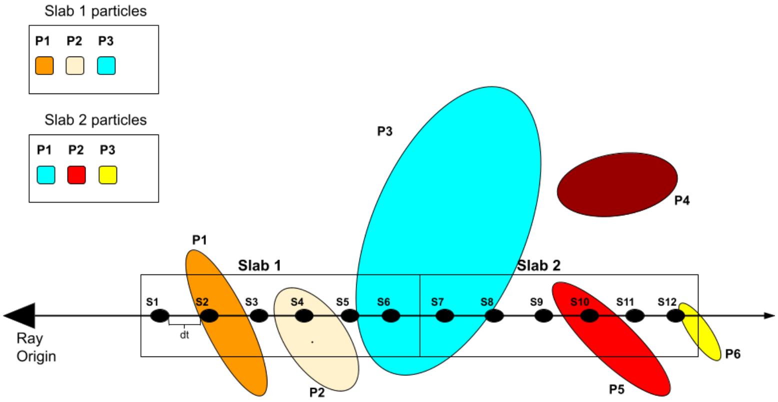


Figure 3.3: This is a scheme of how the particles are gathered with the Uniform Slab Size method.

ray radiance contribution. Because of that, another approach is suggested, which is called slab-tracing [13].

In this approach, we trace the ray in smaller intervals called slabs. Each slab has the same number of sample points, and we evaluated only the particles that intersect the slab. If the slab does not contain any particles, it is skipped, and the calculation moves to the next slab. In this way, we calculate the particle response only for the Gaussian, which will have a significant impact on the final radiance of the ray [6].

The uniform slab method is as follows. We start by shooting rays from the camera position outward. Before we start tracing the rays, we can calculate the bounding box of the whole scene. If a ray does not intersect the boundaries of the scene, we can terminate it early. We trace these rays in small segments that are the slab intervals. The size of the slab ΔS is shown in equation 3.34, where Δt is the step size (or the distance between the individual samples) and $N_{samples}$ is the number of samples per slab. This gives us two parameters to control the sampling process [6].

$$\Delta S = \Delta t N_{samples}) \tag{3.34}$$

In each slab, we trace a segment of the original ray with the size of our slab. During the tracing of the segment, we store all unsorted references of the intersected particles into a hit buffer. After we acquired the refrences to all particles within the slab we accumulate the radiance C_s and density σ_s for each of the sample points $s_i (i \in [1, N_s])$ as shown in equations 3.32 and 3.33. The position of the calculation to the position of the sample point s_i is shown in equations 3.35 and 3.36, where o_{ray} is the origin of the ray, d_{ray} is the direction of the ray, and t_{min} is the distance to the closest boundary of the scene.

$$s_i = s_{i-1} + d_{ray}\Delta t \tag{3.35}$$

$$s_0 = o_{ray} + d_{ray}t_{min} \tag{3.36}$$

After processing all of the sample points within the slab, the accumulated radiance and density are then stored in an individual buffer. With these buffers calculated, we can start combining their accumulated values into a final ray radiance, as shown in the equations 3.16. Along with the radiance, the transmittance value also has to be updated. We can set a parameter for controlling the minimum transmittance threshold. If our transmittance falls below this threshold, we can perform an early exit. With this, we have calculated the final radiance of the ray which can be outputted on the screen [6].

This process can be easily implemented with GPU support in mind. The only thing that is required for this algorithm is to implement a ray-gen and an any-hit program. This ray-gen program will focus on the main calculations for the radiance and density accumulation, along with the composition of the samples. An example of this implementation of this algorithm can be seen in algorithm 5. The any-hit program in this algorithm is very simple. The only thing that is required is a large hit buffer for storing references to the intersected particles, which has to be allocated on the GPU and keep track of the number of intersected particles. This hit buffer can only store the ID of an intersected particle. In this way, the hit buffer does not need to be unnecessarily larger than it has to be. Since we have pre-allocated the buffer before we start the allocation, we also have to set a limit on how many particles we can process in a single slab. After that, the any-hit program only has to store the reference/ID of an intersected particle and increment the number of intersected particles. An example of this simple any-hit program is shown in algorithm 4 [6].

This approach leads to higher quality images, but requires a lot of sampling which impacts the time required to render an image. But even with a large number of samples, this approach still produces real-time rendering results.

Algorithm 4 Algorithm for the Uniform Slab tracing Any-Hit program [6]

```
function Any-Hit
    //The number of Intersected particles can be stored as a ray payload
    $N_{intersected} = GetNumberOfParticles()$
    $ID_{particle} = GetIntersectedParticleIndex()$

    //Stop the traversal if we reached max number of particles
    if $N_{intersected} + 1 \geq N_{max}$ then
        $StopTraversal()$
        $return$
    end if

    $Hitbuffer[N_{intersected}] = ID_{particle}$

    $UpdateNumberOfParticles(N_{intersected} + 1)$
end function
```

3.3.4 Naive Closest Hit Raytracing

This subsection will focus on a different approach compared to the uniform slab tracing method. Instead of accumulating the radiance and density for a single sample point from multiple different particles, we can sample each intersected particle only once. The sampled position should be the position along the ray with the highest possible response (density) of the particle. For this, we need to find the distance t_{max}, which is the distance along the ray to the point with the highest particle density. If we define our ray as $r(t) = o + t\omega$, where o is the origin of the ray and ω is the direction of the ray, we can calculate the maximum response as shown in equation **??**. In the exact terms of our Gaussian particle, this can also be expressed as shown in equations 3.38 and 3.37. Equation 3.38 also shows a simplified calculation without the need to calculate the covariance matrix and its inverse matrix using the transformed origin o_g and direction ω_g which are calculated with the inverse scaling matrix S, which is easy to compute, and the transposed (inversed) rotation matrix R, which is also easy to calculate. The calculation of o_g and d_g is shown in equations 3.39 and 3.40 respectively [5].

$$t_{max} = argmax_t(o + t\omega) \tag{3.37}$$

$$t_{max} = \frac{(\mu - o)^T \Sigma^{-1} \omega}{d^T \Sigma^{-1} \omega} = \frac{-o_g^T \omega_g}{\omega_g^T \omega_g} \tag{3.38}$$

$$o_g = S^{-1} R^T (o - \mu) \tag{3.39}$$

$$d_g = S^{-1} R^T \omega \tag{3.40}$$

Algorithm 5 Algorithm for the Uniform Slab tracing Ray-Gen program [6]

```
function RAY-GEN( )
    $o = GetStartingRayOrigin()$
    $d = GetStartingRayDirection()$
    $L = (0.0, 0.0, 0.0)$
    $T = 1.0$
    $\Delta S = \Delta t * N_s$
    $t_{current} = t_{SceneMin}$

    //Traverse the whole scene or trasmittance reached a threashold
    while $t_{current} < t_{SceneMax}$ and $T > T_{min}$ do
        $Slab_{min} = t_{current}$
        $Slab_{max} = t_{current} + \Delta S$

        //Traverse the scene all particles from the slab
        $HitBuffer = TraceRay(o, d, Slab_{min}, Slab_{max})$

        //Accumulate radiance and density for $N_s$ samples
        $RadianceBuffer, DensityBuffer = ProcessSlab(HitBuffer)$

        for all $s \in 1 \ldots N_s$ do
            $C_s = RadianceBuffer[i]$
            $\sigma_s = DensityBuffer[i]$
            $\alpha = 1.0 - exp(-\sigma_s * \Delta t)$
            $L+ = C_s * \alpha * T$
            $T* = 1.0 - alpha$
        end for

        $t_{current}+ = \Delta S$
    end while
    return $L$
end function
```

Now, we are able to evaluate the kernel function of the Gaussian particle. We can focus on compositing the individual contributions of the intersected particles. Since we have parameterized each particle, we calculate the output color of a pixel for each ray using the standard volume rendering approach, shown in equations 3.13 and 3.14 [5] [6].

This volumetric rendering calculation can then be easily approximated using numerical integration to a discretized version shown in equation 3.15 where we assume that σ_i is calculated as shown in equation 3.41 ($\tilde{\sigma}_i$ is the i-th particle density coefficient and ρ is the particle's kernel function).

$$\sigma_i(r(t)) = \tilde{\sigma}_i \rho_i(r(t)) \tag{3.41}$$

Unlike the uniform slab raytracing method, the individual sample points have to be sorted by their depth (front-to-back). As the name of this approach suggests, we can go with the most straightforward way and always evaluate only the closest sample. After processing the closest single sample, we can find another closest sample point and repeat the process as shown in Figure 3.4, where we can see the order of the samples $S1 - S5$, where each sample S counts as one traversal of the scene. By applying these techniques, we are able to render our scene represented by the volumetric Gaussian particles even with a single sample per particle [5].

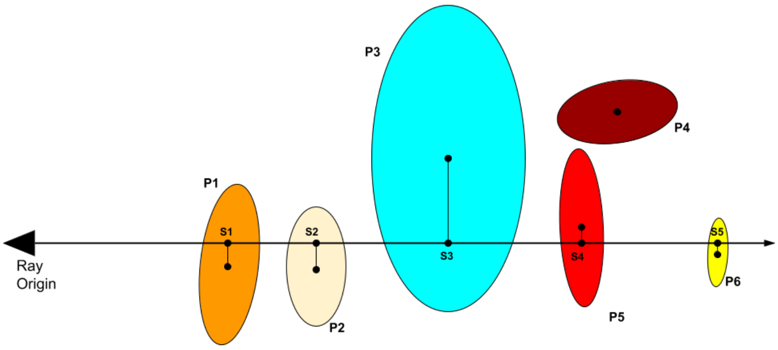


Figure 3.4: This is a scheme of how the sample points are gathered with the Naive closest hit method.

To implement this approach with GPU acceleration support, we can do the following. Since we always want to process only a single sample point that is closest, we can implement this method with only a ray-gen program and a closest-hit program. In this case, the ray-gen program will handle only the scene traversal, as shown in the algorithm 6. Note that since we are not sampling in uniform given intervals we have to keep track of the last hit distance so we are not evaluating a single sample point multiple times. The rest of the calculations can be done in the closest-hit program. We can avoid this issue by setting the starting distance of the ray (or shifting the origin of the ray) by the last hit distance $t_{hit} + \epsilon$, where ϵ is a very small constant. Optionally, we can keep a reference to the closest particle and skip it during the traversal if we do not want to use the constant ϵ. The current ray radiance and transmittance can be stored as a ray payload, which is then accessible in the closest-hit program. We can sometimes find particles whose calculated density is very small, and because of it, will not have a significant impact on the final ray radiance. To avoid unnecessary calculations, we can set a minimum threshold σ_{min}. In this way, we are able to combine the sample points right in the closest-hit program, as shown in the algorithm 7.

The main issue with this approach is that, in order to acquire the closest sample, we have to perform a full scene traversal. This operation is very expensive, and for this purpose the next subsection will explore a different approach which is built on the same ideas which is trying to minimize this issue.

Algorithm 6 Algorithm for the Naive Closest hit Ray-Gen program [5]

```
function Ray-gen
    $o = GetStartingRayOrigin()$
    $d = GetStartingRayDirection()$
    $L = (0.0, 0.0, 0.0)$
    $T = 1.0$
    $t_{current} = t_{SceneMin}$
    //Traverse the whole scene or trasmittance reached a threashold
    while $t_{current} < t_{SceneMax}$ and $T > T_{min}$ do
        $Payload = [T, L]$
        $T, L, t_{hit} = TraceRay(o, d, t_{current}, t_{SceneMax}, Payload)$
        $t_{current} = t_{hit} + \epsilon$
    end while
    return $L, T$
end function
```

Algorithm 7 Algorithm for the Closest-Hit program [5]

```
function Closest-Hit
    $o, d = GetRay()$
    $t_{hit}, particle = GetHit()$
    $T, L = GetPayload()$

    $\sigma_{hit} = CalculateParticleDensity(o, d, particle)$
    if $\sigma_{hit} > \sigma_{min}$ then
        $L_{hit} = EvaluateRadianceColor(d, particle)$
        $L = L + T * \sigma_{hit} * L_{hit}$
        $T = T * (1 - \sigma_{hit})$
    end if
    $UpdatePayload(L, T)$
end function
```

3.3.5 Adaptive k-Closest Hit Raytracing

This approach can be seen as an improved version of the previous closed-hit naive raytracing approach, as they both share the main ideas.

Here, just like in the previous approach, we want to sample each particle only once at the position with the highest density (which also lies on the ray). The calculations for the distance to the point of the highest density t_{max} are the same as shown in equation 3.38. We also use the same method for combining the samples as the naive approach (equation 3.15). The main difference in this approach is the way of gathering hits. At the end of the previous subsection, we have established that scene traversal is an expensive operation, and its use should be minimized. Just like in the naive approach, we are required to gather samples in a sorted order of front-to-back. To minimize the traversal of the scene, we would like to gather multiple samples [5].

To do this, we can use an any-hit program instead of the closest-hit program and simply sort our samples during the sample-gathering process. We can define a hit buffer where we will store the first closest k hits in a sorted way, where k is the size of the hit buffer. Figure 3.5 shows how the samples are collected for $k = 3$. We can see that the samples are being collected in dynamically sized slabs (we see that the size of Slab 1 is not the same as the size of Slab 2). We can use the insertion sort algorithm to sort the individual hits as they are gathered. Note that our hit buffer can be set to be very large, but in that case we are required to pre-allocate this memory somewhere on the GPU and accessing this memory can be costly or we have a very small buffer which can be stored as the ray payload, in which case it does not have to be pre-allocated and can be accessed more efficiently [5]. Also, note that if we choose the smaller buffer version, we will have to traverse the scene more times.

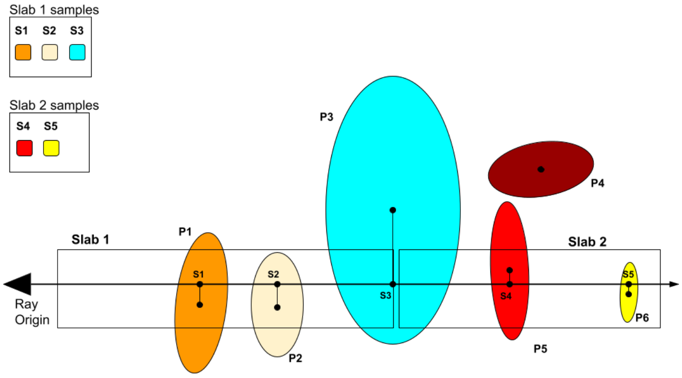


Figure 3.5: This is a scheme of how the sample points are gathered with the adaptive k-closest hit method.

The following text will consider the use of a smaller buffer with the option to store the buffer as a ray payload.

After we have gathered the individual hits, we can process each hit in the ray-gen program to calculate the contribution to the output radiance (same as in the naive method) and continue traversing from the last stored hit. Similarly to the naive method, we can use the ϵ constant trick to avoid sampling particles multiple times.

The algorithms 8 and 9 show how the ray generation and any-hit programs can be implemented, respectively.

Algorithm 8 Algorithm for the Ray-Gen program [5]

```
function Ray-gen
    $o = GetStartingRayOrigin()$
    $d = GetStartingRayDirection()$
    $L = (0.0, 0.0, 0.0)$
    $T = 1.0$
    $t_{current} = t_{SceneMin}$
    $Hitbuffer[k]$

    //Traverse the whole scene or trasmittance reached a threashold
    while $t_{current} < t_{SceneMax}$ and $T > T_{min}$ do
        $Payload = Hitbuffer$
        $HitBuffer = TraceRay(o, d, t_{current}, t_{SceneMax}, Payload)$

        for all $Hit(t_{hit}, Particle) \in HitBuffer$ do
            $\sigma_{hit} = CalculateParticleDensity(o, d, particle)$
            if $\sigma_{hit} > \sigma_{min}$ then
                $L_{hit} = EvaluateRadianceColor(d, particle)$
                $L = L + T * \sigma_{hit} * L_{hit}$
                $T = T * (1 - \sigma_{hit})$
            end if

            $t_{current} = t_{hit}$
        end for
        $t_{current} += \epsilon$
    end while
    return $L$
end function
```

Algorithm 9 Algorithm for the Any-Hit program [5]

```
function ANY-HIT($SortedHitBuffer, t_{hit}, particle, k$)
    $HitBuffer = GetPayload()$
    $hit = (t_{hit}, particle)$

    //Add the hit into the HitBuffer with insertion sort
    for all $i \in 0 \ldots k-1$ do
        if $hit.t_{hit} < HitBuffer[i].t_{hit}$ then
            $swap(HitBuffer[i], hit)$
        end if
    end for

    //Prevent traversal from stopping for the first k closest hits
    if $t_{hit} < HitBuffer[k-1].t_{hit}$ then
        $IgnoreHit()$
    end if
    //Otherwise stop the traversal
    if $t_{hit} > HitBuffer[k-1].t_{hit}$ then
        $StopTraversal()$
    end if
    $UpdatePayload(HitBuffer)$
end function
```

Chapter 4

Custom Differential Gaussian Tracer

This chapter is dedicated to the implementation of a custom Gaussian renderer. This Renderer is planned to be able to render a scene represented by Gaussian particles. The renderer will be implemented in CUDA C/C++ with the NVIDIA OptiX framework (explained in the following sections) for the main rendering logic and Python for the higher-level logic. To simplify the package management the Anaconda 3 distribution will be used in this project as well (also explained in the following section).

4.1 Anaconda

Anaconda is an open source, comprehensive distribution of Python and R programming languages specifically engineered to meet the demands of data-intensive disciplines such as data science, machine learning, and scientific computing. The distribution unifies a curated selection of more than 1,500 precompiled packages, including very popular libraries such as NumPy, pandas, SciPy, and Matplotlib, with a robust environment and package manager named Conda (Anaconda, Inc., 2024). By bundling these components into a single installer, Anaconda eliminates the need for end users to compile libraries from source or resolve intricate dependency trees manually, thereby accelerating project setup and reducing configuration errors across Windows, macOS, and Linux platforms.

At the heart of the Anaconda ecosystem lies Conda, a cross-platform tool that performs dual roles as both a package manager and an environment manager. The Conda binary distribution model ensures that scientific and numerical libraries are executed with consistent performance and compatibility, regardless of the user's operating system. Through Conda environments, researchers and engineers can encapsulate specific versions of the interpreter and its associated libraries, thus isolating the dependencies of one project from those of another. This isolation is instrumental in reproducibility: An environment specification can be exported to a YAML file and shared with collaborators, who can then reconstruct an identical computational context with a single command (Anaconda Documentation, 2024).

Beyond command line operations, Anaconda includes Navigator, which is a graphical user interface that abstracts the environment and package administration into a point-and-click experience. Navigator provides immediate access to development tools such as JupyterLab, Jupyter Notebook, Spyder, Visual Studio Code, and RStudio, all of which are staples in academic and industrial research settings.

4.2 NVidia OptiX Framework

NVIDIA OptiX is a general-purpose ray-tracing engine developed by NVIDIA that enables the efficient deployment of complex ray-based algorithms on modern Graphics Processing Units (GPUs) 11. It provides a flexible application programming interface (API) designed to handle key tasks such as ray traversal, primitive intersection, and material shading. By capitalizing on the inherent parallelism of GPUs - and, starting from Turing architecture, specialized RT cores - OptiX significantly reduces the complexity associated with implementing high-performance ray tracing [18].

4.2.1 Pipeline

In contrast to fixed-function pipelines with a few programmable stages commonly utilized in rasterization hardware, OptiX adopts a more programmable and customizable pipeline model. As discussed in the previous chapters, users can define a number of specific programs to customize the pipeline to suit their needs. These programs include:

- ray-generation: defines how and where rays are emitted
- closest-hit: dictates how the closest geometry intersections are processed after a traversal
- any-hit: dictates how any geometry intersections are processed during a traversal
- miss: defines a behavior when no geometry is intersected
- intersection: dictates how to calculate the intersection for custom primitives

Starting with OptiX version 7, developers can also enjoy low-level control over GPU memory, thread scheduling, and data flow, allowing integration with custom rendering pipelines and data structures [18]. However, these features will not be discussed here.

4.2.2 Usage and Integration

Beyond classical photorealistic rendering, OptiX proves valuable in a range of applications, including scientific visualization, real-time simulations, and advanced geometric or volumetric computations. Thanks to its programmable stages, it can accommodate custom intersection routines for procedurally defined surfaces (e.g., fractals, implicit surfaces) and volumetric shapes (e.g., Gaussian fields or smoke volumes). It also supports multihit ray queries, thanks to the any-hit program, which is frequently required in volume rendering, order-independent transparency, or global illumination methods. Each ray can also store and carry a payload of data, which can be modified during the traversal of the scene. Each ray can store up to 32 unsigned integer values that represent the payload [19]. Because of that, some values (for example, float) have to be converted into an unsigned integer to store them as a payload.

In this work, OptiX is used for:

- Accelerated Ray Traversal: By constructing a BVH for scene objects, OptiX handles the traversal of the scene and identifies intersection candidates between rays and volumetric or geometric primitives.

- Custom Intersection Programs: Optix offers the option of implementing a custom intersection logic. This logic is implemented specifically for 3D Gaussians, treating each Gaussian as either a bounded volume or an implicit shape. This allows us to compute intersections on the GPU without manually coding the BVH traversal.

- Custom Shading and Composition: Through programmable shading stages, the contributions of each Gaussian to color and opacity can be accumulated, enabling layered or alpha-blended visualization of overlapping Gaussian fields. This can be handled in any-hit or closest-hit programs or even in the main ray-generation program.

Because OptiX abstractly manages the complexities of BVH construction and traversal, we can concentrate on the domain-specific tasks (e.g., computing Gaussian contributions and combining them via alpha blending) rather than low-level performance optimizations.

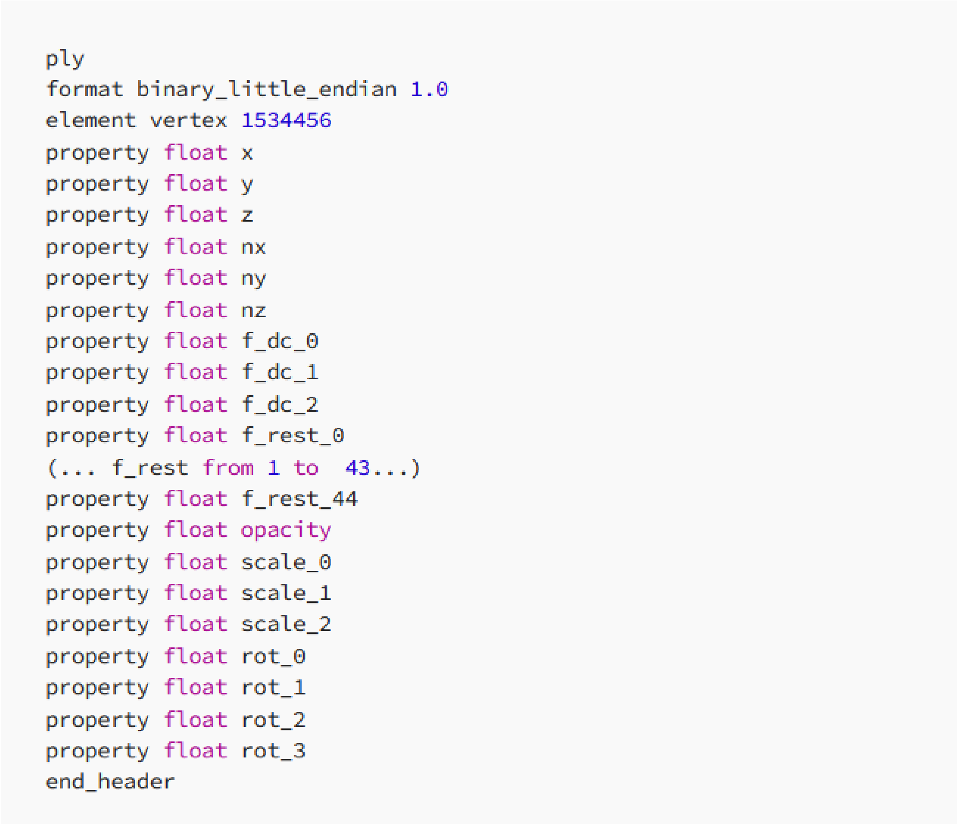
Sadly, the Optix framework is not very publicly popular, and it is difficult to find many public study materials on how to operate with the OptiX framework. The only exception is the programming guide document [18], API reference document [19] created by NVIDIA, and a handful of sample projects which are accessible to anyone who downloaded and installed Optix.

OptiX's core API is exposed as a traditional C interface with C++ compatible headers, and the SDK itself is written in C and C++. As a result, any host application that drives OptiX must be written in C or C++ and compiled with NVIDIA's NVCC driver (or a system compiler invoked through NVCC), linking to the OptiX and CUDA runtime libraries. On the device side, every programmable stage of the ray-tracing pipeline is written as standard CUDA C/C++ code. This code is then compiled into a PTX (Parallel Thread Execution) intermediate representation of the implemented CUDA kernels, which are then used by the Optix API during the launch of the ray-tracing pipeline.

4.3 Implementation

4.3.1 Data Loading

Before we can render a pre-trained scene, we have to figure out how to load it and save it. The most common way to store a Gaussian scene is in PLY format. This format is most commonly used to store polygonal or point-cloud data. Since the individual Gaussian particles do not share any relation with each other, we work with the Gaussian scene as a point cloud. Every PLY file has two main parts: a header and a binary body. The header of the PLY file is stored in ASCII format and is readable by humans. It contains information on the data layout of the binary body and the binary format in which the body data is stored. This means that the header defines what properties are stored for a single entry/point and also other information, such as the number of points stored in the file (this is important for the loading process) or how many types of entry are present (for Gaussian scene, only one type of entry is present). The way a header for a Gaussian scene can look is shown in Figure 4.1. The body is where the actual data are stored in binary format as a list of entries/points.



```
ply
format binary_little_endian 1.0
element vertex 1534456
property float x
property float y
property float z
property float nx
property float ny
property float nz
property float f_dc_0
property float f_dc_1
property float f_dc_2
property float f_rest_0
(... f_rest from 1 to  43...)
property float f_rest_44
property float opacity
property float scale_0
property float scale_1
property float scale_2
property float rot_0
property float rot_1
property float rot_2
property float rot_3
end_header
```

Figure 4.1: An example of a header in PLY file containing a Gaussian scene [16].

So, in order to read and load data from a PLY file, we need to define a structure that matches the structure of the data stored in the header. Then, initialize a list with the same number of entries and load the binary data into this list of structures. With the defined structure, saving the scene is very simple. We simply write the header so it matches the same structure used for loading, and then save the whole scene in a binary format.

Sometimes, there could be issues with the saving or loading process of the scene. It is good to check that some parameters are loaded correctly. Because of this, it is a good practice to check that the quaternions are normalized (so they represent only rotation) and that other parameters like the density parameter are within reasonable bounds.

4.3.2 Gaussian Viewer

To show how the 3D Gaussian raytracing can be implemented, I first created a custom raytracing viewer for pre-trained Gaussian scenes. The raytracer was built with the Nvidia Optix framework for GPU acceleration. Because of this, the main core of the application is written in CUDA C/C++ code. This means that the main functions are written as OptiX kernel functions (these define the ray-gen, any-hit programs, etc.). The rest of the application, which means the higher-level logic, is handled by a collection of Python scripts. This is a common practice as it is often seen in other projects which focus on the same topic of raytracing 3D Gaussian particles [5] [6] [1]. However, this approach poses a smaller issue. The Optix framework does not support the Python language. Luckily, there are Python packages that serve as a wrapper for the Optix framework. The one used in my project is called simply Python-optix. With this, I was able to call Optix's specific functionalities from a Python script. One small disadvantage of this approach is the fact that the Python scripts are using JIT (Just In Time) compilation. Because of this the python-optix package also compiles the Optix kernels when they should be used, often even if the code did not change. Note that this issue does not have to be present if a different Optix wrapper package is used.

To implement the viewer for the 3D Gaussian scene, we only require the forward pass to gather the radiance for each individual ray (pixel). I have chosen to base my implementation on the uniform slab tracing method, which has been discussed in the previous chapter. I have spotted a large disadvantage of the approach, which is the large hit buffer required for the hit gathering. My implementation does not use a hit buffer for the forward pass. Instead of gathering each individual particle ID in any-hit program and then processing all of the calculations in the ray-gen program, as shown in the previous chapter, I moved the process of radiance and density accumulation to the any-hit shader. Firstly, I set the accumulation buffers as a ray payload. This payload is easily accessible in the any-hit program and enables us to accumulate the radiance and density right in the any-hit program. This approach has several advantages. Data stored in the ray payload is optimized for quick memory access compared to normally allocated data on the GPU. Since the hit buffer is no longer required for the forward pass, there is no limit to the number of particles that can be sampled during the radiance and

Algorithm 10 Algorithm for the Forward pass Ray-Gen program [6]

```
function RAY-GEN( )
    $o = GetStartingRayOrigin()$
    $d = GetStartingRayDirection()$
    $L = (0.0, 0.0, 0.0)$
    $T = 1.0$
    $\Delta S = \Delta t * N_s$
    $t_{current} = t_{SceneMin}$

    //Traverse the whole scene or trasmittance reached a threashold
    while $t_{current} < t_{SceneMax}$ and $T > T_{min}$ do
        $Slab_{min} = t_{current}$
        $Slab_{max} = t_{current} + \Delta S$

        $Payload = [RadianceBuffer, DensityBuffer]$
        //Traverse the scene all particles from the slab
        $Payload = TraceRay(o, d, Slab_{min}, Slab_{max}, Payload)$

        $[RadianceBuffer, DensityBuffer] = Payload$

        for all $s \in 1 \ldots N_s$ do
            $C_s = RadianceBuffer[i]$
            $\sigma_s = DensityBuffer[i]$
            $\alpha = 1.0 - exp(-\sigma_s * \Delta t)$
            $L+ = C_s * \alpha * T$
            $T* = 1.0 - alpha$
        end for

        $t_{current} + = \Delta S$
    end while
    return $L$
end function
```

density accumulation process. However, this also introduces a disadvantage. We are limited by the small memory of the ray payload. In the case of Optix, the payload can store only up to 32 unsigned integers (we are also able to easily store and retrieve floating point data, as mentioned before). For a single sample, we need four values: three for color (one for each color channel) and one for density. This means that we are only able to store up to eight samples per slab. My implementation uses seven samples per slab and reserves the remaining four values for other data, such as the number of particles intersected. This modified any-hit program is shown in the algorithm 11.

With some of the calculations moved to the any-hit program, the ray-gen program is then much simpler. The ray-gen program can now handle only the scene traversal logic and the composition of the accumulated samples to

Algorithm 11 Algorithm for the Forward pass tracing Any-Hit program [6]

```
function Any-Hit
    //The number of Intersected particles can be stored as a ray payload
    $N_{intersected} = GetNumberOfParticles()$
    $ID_{particle} = GetIntersectedParticleIndex()$

    //Stop the traversal if we reached max number of particles
    if $N_{intersected} + 1 \geq N_{max}$ then
        $StopTraversal()$
        $return$
    end if

    $Hitbuffer[N_{intersected}] = ID_{particle}$

    $UpdateNumberOfParticles(N_{intersected} + 1)$
end function
```

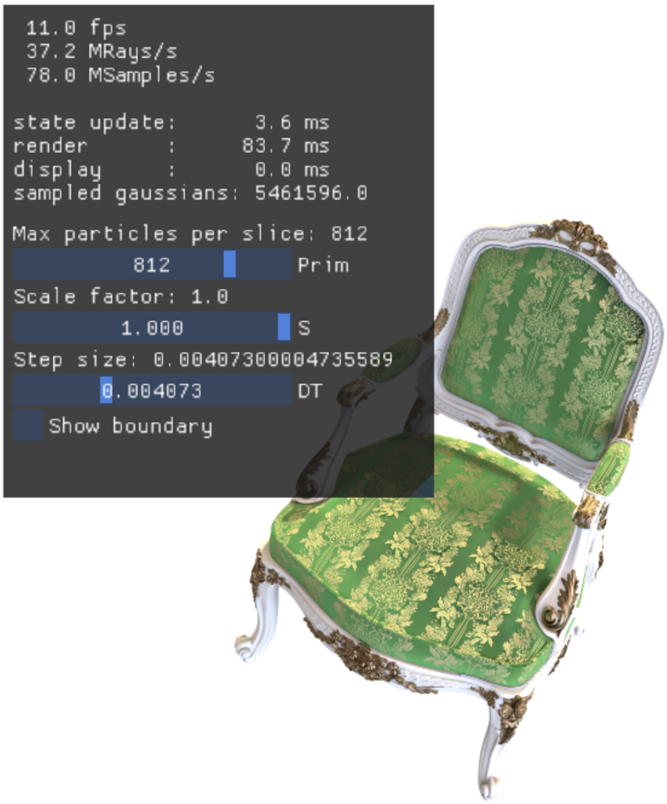
the final ray radiance and transmittance. The simplified ray-gen program is shown in the algorithm 10.

To show some of the statistics of the rendering process, I added a simple GUI created with the IMGUI library [21]. This library is very commonly used for rendering demos and viewers, since it is simple to set up, can be used with almost any framework, and provides a simple way of displaying text and taking input from the user.

With the simple GUI setup, I added a simple text box that shows the time required for rendering, displays the rendered image, and displays the number of frames per second (FPS). Along with it, I am also calculating the number of mega-rays per second (MRay/s) and the number of mega-samples per second(MSample/s).

The GUI also provides the option to tweak some of the parameters required in the rendering process, such as the step size and the maximum number of particles per ray. An example of how the Gaussian Viewer window looks is shown in Figure 4.2

To be able to visualize the individual, I added the option to show the elliptical boundary of the particles and added a scale coefficient so that the individual particles are better visible. An example of how this visualization can look is shown in Figure 4.3. In this way, the viewer can visualize how the Gaussian scene is represented.



```
 11.0 fps
 37.2 MRays/s
 78.0 MSamples/s

state update:      3.6 ms
render      :     83.7 ms
display     :      0.0 ms
sampled gaussians: 5461596.0

Max particles per slice: 812
          812             Prim
Scale factor: 1.0
         1.000            S
Step size: 0.004073000004735589
       0.004073           DT
   Show boundary
```

Figure 4.2: The window of the custom Gaussian Viewer.

4.3.3 Gaussian Tracer

To be able to use the custom tracer in the scene training process, I required a rendering pipeline that can perform the forward pass, which produces the rendered image, and the backward pass, which is used for the gradient backpropagation in the training process. The forward pass was already implemented for the Gaussian viewer. I duplicated the forward pass implemented and removed the unnecessary code that was used in the visualization process. In this way, I could have used separate logic for the training pipeline and the visualization pipeline.

Next, we had to implement the backward pass. The backward pass is always at its core similar to the forward pass, but instead of producing an image, we want to calculate the gradients for every intersected particle in the scene from the gradient color calculated from the image generated by the forward pass and the ground truth image. We still have to trace the scene in the same way as in the forward pass, but the tracer now requires a reference to the particles to calculate the gradients of their individual parameters:

position, density, scale, rotation, and color parameters(coefficients of the spherical harmonic functions).

Firstly, we perform the same operation as in the forward pass. This includes sampling multiple particles and then accumulating their radiance and density in the radiance and density buffers, but we also have to store the references to the intersected particles. This means that we require a hit buffer that will store these references (the buffer can store the particle references in any order). We process these particles in the ray-gen program, where we calculate the gradients for each sample the particle has participated in and accumulate them in a gradient buffer, which has the same number of entries as there are particles. Since this code is used on the GPU, we have to use an *atomicAdd*() function to prevent race conditions. The pseudocode for the backward pass can be seen in algorithm 12.

When the backward pass is finished it outputs the gradient buffer which is then used by the optimizer to improve the scene visual quality.

Algorithm 12 Algorithm for the Backward pass Ray-Gen program [6]

```
function RAY-GEN( )
    $P = GetPointCloud()$
    $\Delta P = InitPointCloudGradients()$
    $o = GetStartingRayOrigin()$
    $d = GetStartingRayDirection()$
    $L = GetFwdRadiance()$
    $\Delta L = GetLossColor()$
    $T = 1.0$
    $\Delta S = \Delta t * N_s$
    $t_{current} = t_{SceneMin}$

    //Traverse the whole scene or trasmittance reached a threashold
    while $t_{current} < t_{SceneMax}$ and $T > T_{min}$ do
        $Slab_{min} = t_{current}$
        $Slab_{max} = t_{current} + \Delta S$

        //Traverse the scene all particles from the slab
        $HitBuffer, Payload = TraceRay(o, d, Slab_{min}, Slab_{max}, Payload)$

        $RadianceBuffer, DensityBuffer = Payload$

        for all $i \in 1 \ldots N_{HitBuffer}$ do
            $floatT_{aux} = T$
            $L_{aux} = L$
            $\Delta p_{hit} = InitializeGradientAccum$
            for all $j \in 1 \ldots N_s$ do
                $C_s = RadianceBuffer[i]$
                $\sigma_s = DensityBuffer[i]$

                $\Delta P_{hit} += AccumColorAndParameterGradients(L, \Delta L, C_s, \sigma_s)$

                $\alpha = 1.0 - exp(-\sigma_s * \Delta t)$
                $L_{aux} += C_s * \alpha * T$
                $T_{aux} *= 1.0 - alpha$
            end for

            $AtomicAdd(\Delta P, \Delta P_{hit})$

            if $i == i_{last}$ then
                $T = T_{aux}$
                $L = L_{aux}$
            end if
        end for

        $t_{current} += \Delta S$
    end while
    return $\Delta P$
end function
```

4.3.4 Scene Training

To be able to use the custom tracer with the forward and backward pass implemented, I would need a training framework that would handle the process of optimization and overall the process of generating (or training) a scene.

For this project, I have used the Raygauss project training framework [6]. The project was also written mainly in Python and CUDA C/C++ and uses the same packages for managing CUDA and Optix functions from a Python script.

I have forked the Raygauss project and replaced their raytracer with my implementation of the forward and backward passes. Then I had to integrate my raytracer to the existing scripts used during the training process and removed scripts which were not useful for my raytracer implementation.

To be able to use the forward and backward passes in quick succession, two different Optix pipelines are needed: one for the forward pass and one for the backward pass. In this way, we can separate the calculations for each pass. When the forward pass is finished, a gradient image is calculated from the forward output and the ground truth image, which is then passed straight to the backward pipeline. The trainer then checks if it should perform Adaptive Density Control. Lastly the BVH is updated with the new parameters of the particle point cloud. A high level overview of this training process can be seen in algorithm 13.

Algorithm 13 A high-level overview of the training process used in the raygauss framework [6]

```
function TRAIN( )
    $P = GetSFMPointCloud(GTImages)$
    $BVH = InitBVH(P)$
    $PipelineFwd, PipelineBwd = InitPipelines(PointCloud, BVH)$

    $i = 0$
    $i_{max} = MaxIterations$

    while $i < i_{max}$ do
        $View, GTImage = SampleTrainView()$
        $Image = PipelineFwd.Launch(View, P, BVH)$
        $LossImage = Loss(Image, GTImage)$
        $\Delta L = PipelineBwd.Launch(View, P, BVH, LossImage)$
        $P = AdamOptimizer(\Delta L)$

        if $DoAdaptiveDensityControl(i)$ then
            $P = AdaptiveDensityControl(P)$
        end if

        $BVH = UpdateBVH(P)$
    end while

    return $P$
end function
```

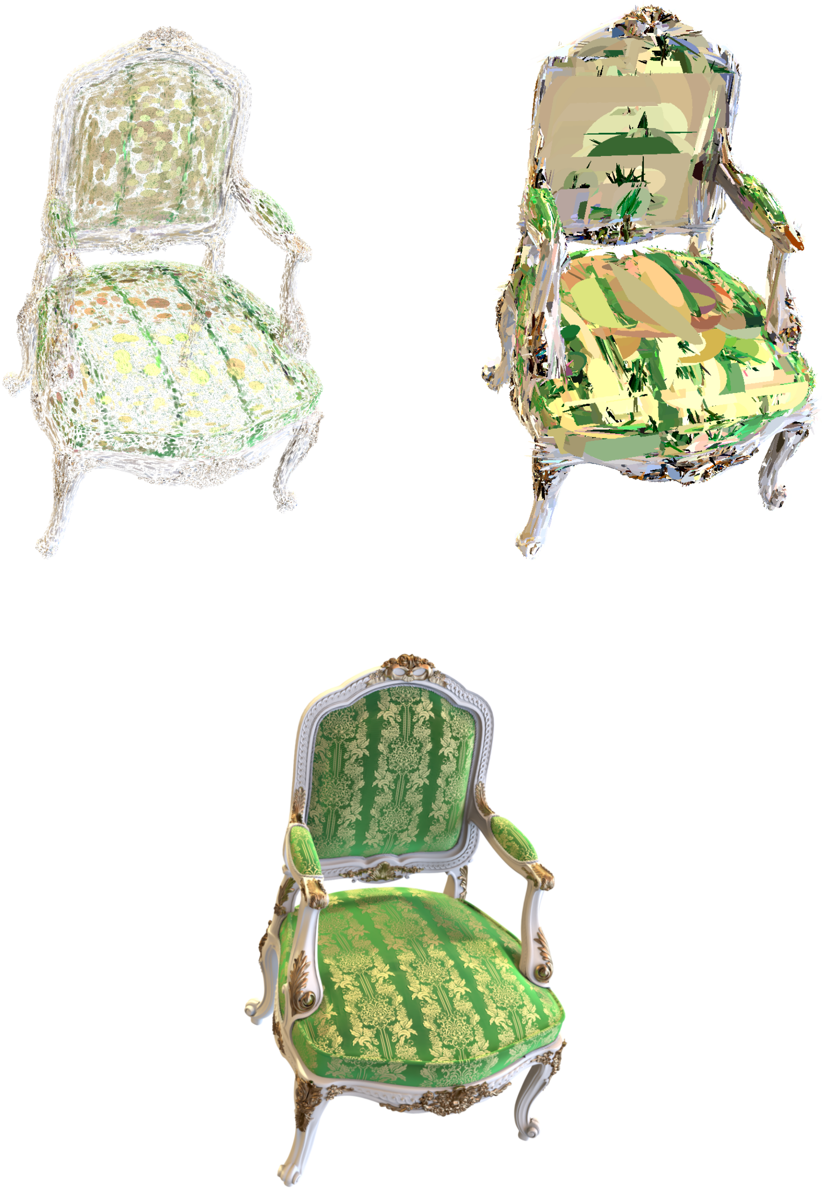


Figure 4.3: This figure shows how the 3D Gaussian particles can be visualized in a way that's visible to the human eye. The bottom image is the standard rendered image. The top left image has particles shrunk with a scale coefficient. The top right image shows the boundaries of the visible Gaussians.

Chapter 5

Results

This section showcases some of the custom-trained scenes that have been produced using the Gaussian Tracer shown in the previous chapter. The section also discusses the performance of the rendering pipeline and the visual quality that can be achieved with the methods used in the chapter discussing the implementation.

The results have been calculated and measured on a computer with:

- GPU: NVIDIA GeForce RTX 3070 Laptop GPU with 16 GB
- CPU: AMD Ryzen 9 5900HS with 3.30 GHz and 8 cores
- RAM: 16 GB
- OS: Windows 10

5.1 Image Quality Metrics

The fidelity of synthesized novel views is most commonly measured with three complementary image-quality metrics: Peak signal-to-noise ratio (PSNR), structural similarity index (SSIM), and learned perceptual image patch similarity (LPIPS) [1] [5] [6].

The trio of PSNR, SSIM, and LPIPS offers complementary insights to the measured data: PSNR is sensitive to absolute pixel deviations, SSIM captures structural distortions, and LPIPS aligns with subjective appearance. Reporting all three of these metrics provides a balanced view of both numerical fidelity and perceptual quality.

This section introduces the formal definition of each score and summarizes the ranges typically reported in Novel View Synthesis projects.

5.1.1 Peak Signal-to-Noise Ratio (PSNR)

PSNR measures the pixel agreement between a reconstructed image K and a reference image I by first computing their mean squared error (MSE). The mathematical definition of MSE is shown in equation 5.1, where m and n are the width and height of the image, respectively.

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{5.1}$$

The PSNR calculation is shown in equation 5.2 .The score is then expressed in decibels (dB).

$$PSNR = 10\log_{10}\Big(\frac{MAX_I^2}{MSE}\Big) = 20\log 10\Big(\frac{MAX_I}{\sqrt{MSE}}\Big), \tag{5.2}$$

with $MAX_I = 255$ for 8-bit RGB data. Higher PSNR indicates lower signal-dependent noise. Scores above ∼32 dB are perceived as sharp and largely artifact-free in full HD resolution [14]. In the context of Novel View Synthesis approaches, the most successful methods can report 30–40 dB on real scenes.

5.1.2 Structural Similarity Index (SSIM)

SSIM was introduced to model perceived changes in luminance, contrast, and local structure. For two image patches x and y, the SSIM is defined as shown in 5.3 , where μ, σ^2, and σ_{xy} denote local means, variances, and covariance, and $C_{1,2}$ are constants that are used to stabilize the calculation of SSIM and prevent division by zero [14].

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{5.3}$$

This calculation can then be used to get a global SSIM score in $[0, 1]$, where the higher score indicates a better (more similar) result, and lower scores indicate visible structural artifacts. Values exceeding 0.95 are considered visually indistinguishable from the reference, whereas scores below 0.75 can reveal obvious structural distortions. Because SSIM incorporates local pattern information, it can correlate better with human vision than purely pixel-based errors such as PSNR.

5.1.3 Learned Perceptual Image Patch Similarity (LPIPS)

LPIPS is a learned perceptual metric that measures the distance between two images in the feature space of a deep convolutional neural network. Given images I and K, activations $\phi_l(\cdot)$ are extracted in multiple layers l and L2-normalised. The final score can then be calculated as shown in equation 5.4.

$$LPIPS(I,K) = \sum lw_l, \|\hat{\phi}l(I) - \hat{\phi}l(K)\|_2^2 \tag{5.4}$$

where the weights w_l are calibrated on large-scale human judgement experiments. LPIPS ranges from 0 (identical) to 1; Lower values indicate greater perceptual similarity. In recent NeRF benchmarks, scores below 0.02 are regarded as essentially indistinguishable from the ground truth, while values above 0.05 denote visible color or texture mismatches [15].

5.2 Created scenes

To be able to train a Gaussian scene, an image dataset is required. I have selected five different image sets. Two of the image sets come from the Mip-NeRF 360 dataset. The remaining three scenes are from the synthetic Blender dataset. These data sets are most commonly used to test algorithms for novel view synthesis. [1] [6] [5].

The scenes Garden and Bicycle are based on realistic images of outdoor areas, which can be difficult to reconstruct. The Scenes Chair, Lego, and Ship each contain single, synthetic, detailed 3D models.

Other Novel View Synthesis algorithms normally showcase their resulting scenes with images after 30,000 iterations of the training process. The resulting scene images after 30,000 iterations from this project are shown in Figure 5.1. As we can see, the resulting images are very realistic, and it can be difficult to distinguish them from realistic photo images.

These scenes can then be viewed using the Gaussian Viewer. During viewing, the user can view the training scene from any angle. It is important to remember that the method used in this project can only accurately reconstruct the scene. In areas that are visible from the input ground-truth images. Figure 5.2 shows the parts of the reconstructed Garden scene that are not documented in the ground-truth images. In these areas, the resulting particle point cloud is very sparse, and we are able to see what the 3D Gaussian particles look like.

To show the training progress of the scene, I generated several images during the training process. These images are shown in Figure 5.3. These images were generated from a training that ran for 2000 iterations. In this collection of images, we can see that the scenes are converging quickly to a photorealistic result. This also shows us that this raytracing approach does not necessarily require 30,000 iterations to get a good enough result, especially for smaller scenes.

5.3 Training Performace

This section discusses the performance and metrics of the training pipeline. To evaluate the visual quality of the trained scenes, I have used the PSNR, SSIM and LPIPS scores which have been explained in the previous sections.

I have evaluated all the trained scenes at the end of their training process. The resulting statistics for each of the trained scenes are shown in Table 5.1. The training was. As we can see, the trained scene and all the trained scenes achieved high visual scores, indicating that they would be difficult to distinguish from the original ground-truth images. The main issue with the trained scenes is the time required to generate them. Training times are very long. But this is an expected result as the raytracing method is much more expensive than, for example, the rasterization approach, which is capable of processing 30,000 iterations in about one hour.

Scene	PSNR ↑	SSIM ↑	LPIPS ↓	# Points	Train Time
Garden	29.78 dB	0.926	0.056	1 636 432	5h 27m
Bicycle	26.10 dB	0.834	0.124	1 615 752	5h 35m
Chair	38.50 dB	0.991	0.008	420 992	3h 15m
Lego	36.90 dB	0.984	0.011	367 179	2h 55m
Ship	31.81 dB	0.913	0.093	467 892	3h 27m

Table 5.1: PSNR, SSIM, LPIPS, point count, and training time for five selected scenes. Garden and Bicycle are outdoor scenes that are typically harder to synthesize, while Chair, Lego, and Ship are single objects that often achieve better scores.

The long training times are then mainly caused by the expensive raytracing rendering. This can also be seen during the visualization of the scenes. Table 5.2 shows the performance scores for each individual scene. For each of the scenes, I have measured the average frames per second (FPS), mega rays per second ($MRay/s$), and mega samples per second ($MSample/s$). The $Mray/s$ have been measured as the number of scene traversals and not the number of "Primary rays" (often equal to the window resolution), which is a more common approach. I have chosen this approach to show how many times the scene has to be traversed. The scenes have been measured on 800x800 resolution. We can see that the viewer still achieves real-time results, but can struggle with the larger outdoor scenes (Garden and Bicycle).

To show the progress of the scene training and how fast the training converges to a realistic result, I have measured the PSNR score during the training process for each of the scenes.

Figure 5.4 shows the outdoor scene training process (Garden and Bicycle). We can see that the PSNR value rises quickly at the start of the training process and then continues to refine the scene more slowly. We can also notice some PSNR dips (Bicycle at iteration 1000). These dips are caused by the adaptive density control process of the point cloud of the particles. During

Scene	FPS	MRays/s	MSamples/s	# Particles
Garden	5.8	145.8	470.5	1 636 432
Bike	4.5	98.1	290.3	1 615 752
Chair	12.1	110.6	187.2	485 210
Lego	13.4	108.5	207.3	367 179
Ship	10.8	133.2	271.6	467 892

Table 5.2: Run-time performance of the five trained scenes. The statistic $MRays/s$ has been calculated as the number of scene traversals (number of processed slabs) per second.

this process, a larger number of new particles can be introduced or removed from the scene, which then has an impact on the visual quality of the scene. Figure 5.5 shows the training progress of the synthetic scenes (Chair, Lego, Ship). We can see that these scenes already start with a reasonably good PSNR score after the initialization. Just like in outdoor scenes, we see the largest improvement right at the beginning of the training process, and it continues to refine the scene more slowly. We can also notice that these scenes converge to a slightly better result than the outdoor scenes. Even with the synthetic scenes, we can still see the dips in the PSNR score at several places. These dips are again caused by the adaptive density control that is taking place.

Next, I compared my measurements of the visual performance of the chosen method with existing projects that also focus on view synthesis and 3D scene reconstruction. I have chosen to compare the results of this project with "older" projects such as NeRF, Mip-NeRF, and Instant-NGP, which do not use 3D Gaussian particles in any way, but share some approaches. Next, I also included the original Gaussian Splatting project (3DGS), which can give us a comparison between the quality of rasterization and raytracing. And finally, I chose two projects that also utilize the raytracing approach: Raygauss and the 3DGRT project. The 3DGRT project implements the approach of the Adaptive k-Closest Hit raytracing method explored in the previous chapter and uses only a single sample per single Gaussian. Tables 5.3, 5.4, and 5.5 show the comparison between individual projects in the PSNR, SSIM, and LPIPS scores, respectively. In these tables, we can see that the 3D Gaussian-based methods perform better than the other methods(NeRF, Mip-NeRF, and Instant-NGP). We can also see that this project has scored similarly to the Raygauss project and is outperforming the other methods, like 3DGS and 3DGRT, with some exceptions (Bicycle).

The raytracing method also has one more improvement compared to the rasterization approach of 3DGS, that is, the scene size. Table 5.6 shows the comparison of the scene sizes between this project and the original Gaussian splatting project. We can see that the chosen raytracing approach is able to generate much smaller scenes ($\approx 3 - 4x$ smaller). This means that the raytracing method can require much less memory than the rasterization approach in practice and still achieve better visual results.

Method	Garden	Bicycle	Chair	Lego	Ship
NeRF [9]	22.70	19.35	34.17	33.31	29.30
Mip-NeRF [10]	–	–	35.65	36.10	31.26
Instant-NGP [11]	25.64	23.69	35.00	36.39	31.10
3DGS [1]	29.58	27.33	35.85	35.87	30.95
RayGauss [6]	29.91	27.21	37.20	37.10	31.95
3DGRT [5]	–	–	35.90	36.20	30.71
This project	29.78	26.10	38.50	36.90	31.81

Table 5.3: Per-scene PSNR (dB) score comparison of the custom implementation with some of the other projects which focus on Novel View Synthesis.

Method	Garden	Bicycle	Chair	Lego	Ship
NeRF [9]	0.653	0.371	0.975	0.968	0.869
Mip-NeRF [10]	–	–	0.983	0.980	0.893
Instant-NGP [11]	–	–	–	–	–
3DGS [1]	0.931	0.871	0.988	0.983	0.893
RayGauss [6]	0.929	0.859	0.990	0.986	0.914
3DGRT [5]	–	–	–	–	–
This project	0.926	0.834	0.991	0.984	0.913

Table 5.4: Per-scene SSIM score comparison of the custom implementation with some of the other projects which focus on Novel View Synthesis.

5.4 Importance of tracer parameters

The training of the scene has several different parameters that have a significant impact on the resulting reconstructed scene.

This section explores the impact of some of the tracer parameters. These parameters include the maximum number of particles per slab N_p and the step size Δt, which defines the distance between individual sample points.

Firstly, I have focused on the impact of the step size parameter Δt on the resulting visual quality of the scene. For this test, I have chosen to work with the PSNR score of the rendered scenes to show the difference in visual quality. I have trained all of the selected scenes: Garden, Bicycle, Chair, Lego and Ship. The scenes have been trained for 2000 iterations and have used the maximum number of particles per slab N_p as $N_p = 1024$. The results of this test can be seen in Table 5.7. The table shows the average PSNR score and the average training times for all scenes. We can see that smaller step sizes Δt can produce visually better scenes. At the same time, we can see that the smaller step sizes also have a large impact on the training time. Because of that, it could be practical to choose a larger step size for better performance at the cost of visual quality. This result of the test has been expected, because smaller step-sizes allow us to sample the Gaussian scene more and because of that we are able to capture more details.

Method	Garden	Bicycle	Chair	Lego	Ship
NeRF [9]	0.360	0.161	0.026	0.031	0.150
Mip-NeRF [10]	–	–	0.018	0.018	0.119
Instant-NGP [11]	–	–	–	–	–
3DGS [1]	0.056	0.121	0.011	0.015	0.118
RayGauss [6]	0.051	0.110	0.009	0.012	0.088
3DGRT [5]	–	–	–	–	–
This project	0.056	0.124	0.008	0.011	0.093

Table 5.5: Per-scene LPIPS score comparison of the custom implementation with some of the other projects which focus on Novel View Synthesis.

# Particles	Garden	Bicycle
3DGS	5 834 784	6 131 954
This project	1 636 432	1 615 752
Absolute difference	4 198 352	4 516 202
3DGS bigger (%)	356,56%	379,51%

Table 5.6: Particle counts for 3DGS versus this project for the outdoor scenes. This table serves as a comparison of output between the methods based on rasterization vs raytracing.

Next, I wanted to test the impact of the number of particles per slab N_p. This parameter allows us to control the amount of particles that can participate in the sampling process for a single slab. Just like in the previous test I have retrained all five scenes with varying N_p parameter on 2000 iterations. I have used $\Delta t = 0.0075$ as the step size during training. Table 5.8 shows the results of this test with the average PSNR scores and the average training times for all scenes. We can see that the parameter N_p appears to have a smaller impact on the visual quality of the trained scenes than the parameter Δt. We see that the visual quality decreases with smaller N_p but only in small amounts. As expected, training time also decreases with smaller N_p. Because of this, it can be good to limit N_p to reasonable values such as 512 or even 256 to increase the rendering performance.

Δt	PSNR	Train time
0.03	20.91 dB	4 m 52 s
0.015	24.62 dB	6 m 02 s
0.0075	26.61 dB	8 m 13 s
0.003 75	26.88 dB	9 m 23 s
0.001 875	26.91 dB	11 m 56 s

Table 5.7: Average PSNR and training time as a function of the integration step Δt.

N_p	PSNR	Train time
1024	26.61 dB	8 m 13 s
512	26.52 dB	8 m 06 s
256	26.41 dB	7 m 30 s
128	26.24 dB	7 m 10 s
64	25.47 dB	6 m 55 s

Table 5.8: Average PSNR and training time as a function of the maximum particles per slice N_p

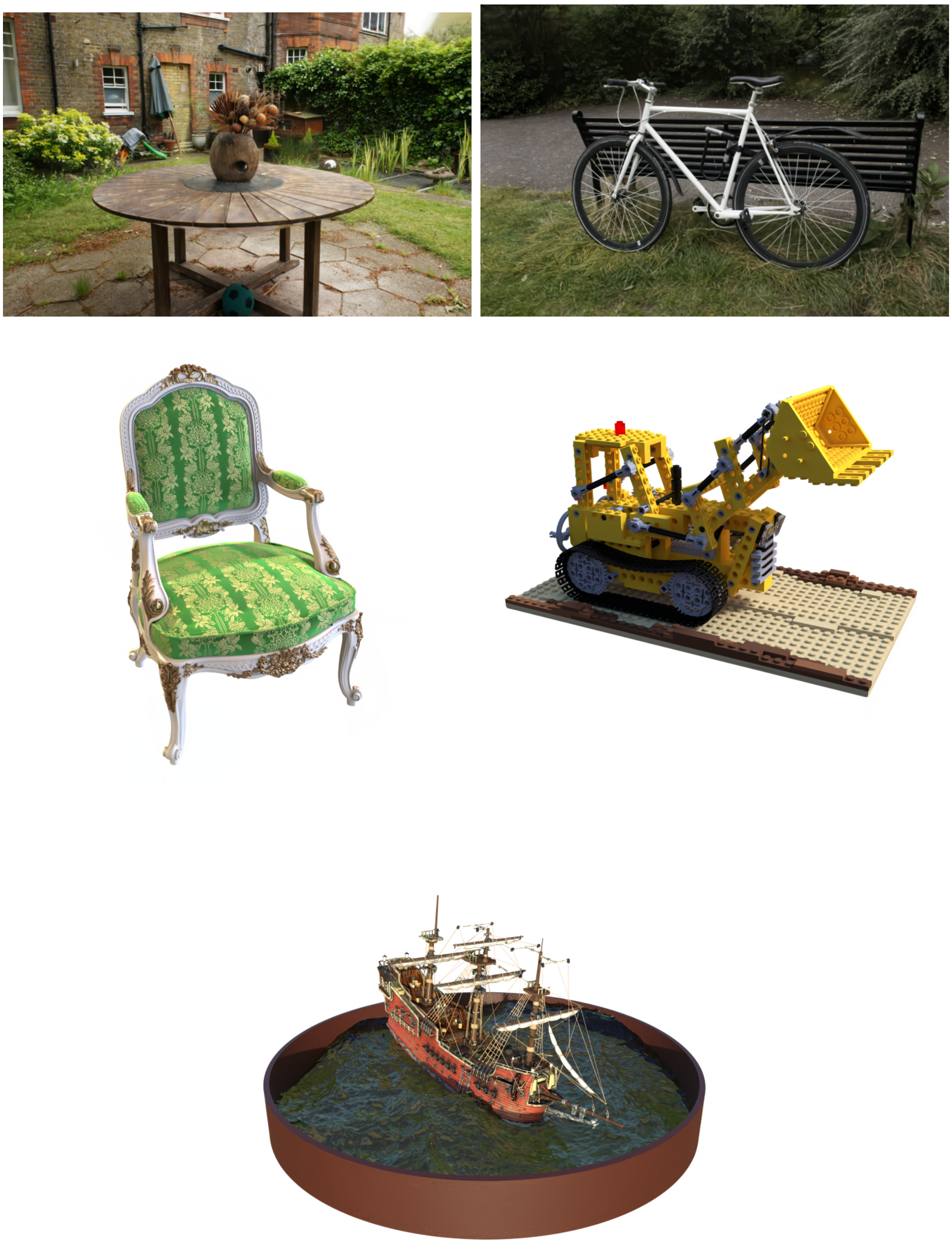


Figure 5.1: This figure shows the rendered images from the trained datasets. The first row shows the scenes from the MIP Nerf dataset (left to right): Garden and Bicycle. The second and third row shows the images of scenes from the Nerf Synthetic dataset (left to right downwards): Chair, Lego, Ship.

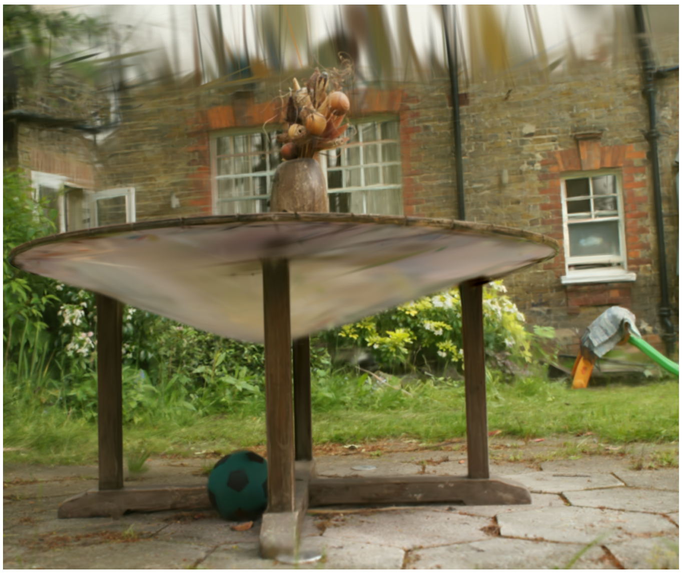


Figure 5.2: Reconstructed scenes are realistically reconstructed only from views which are close to the input ground-truth images. This figure shows parts of the Garden scene from a view which is away from any of the input images.

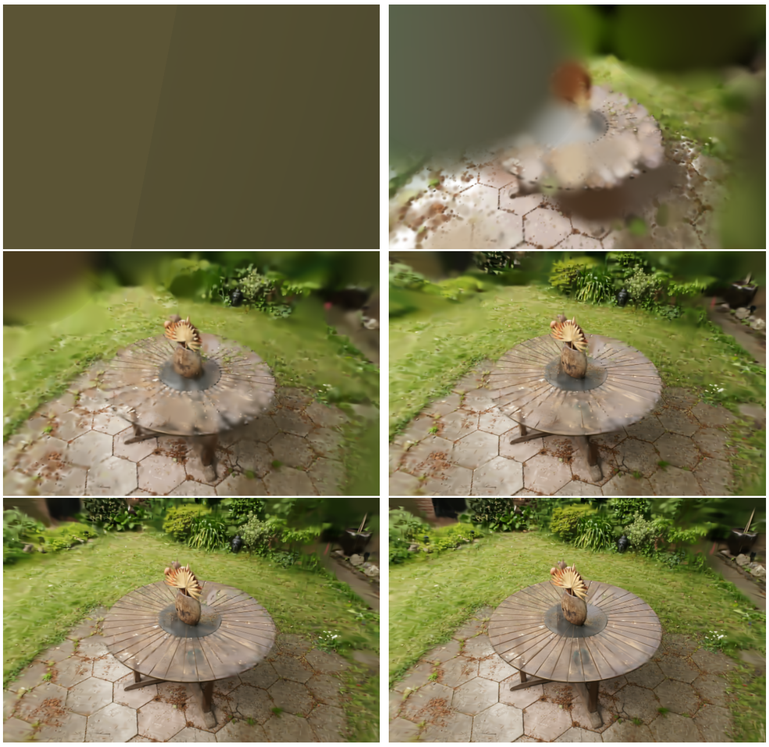


Figure 5.3: This figure shows the progress of the Garden scene training process. From left to right downwards, the images show the garden scene in iterations: 0 (Initial point cloud), 150, 300, 750, 1200, and 2000.

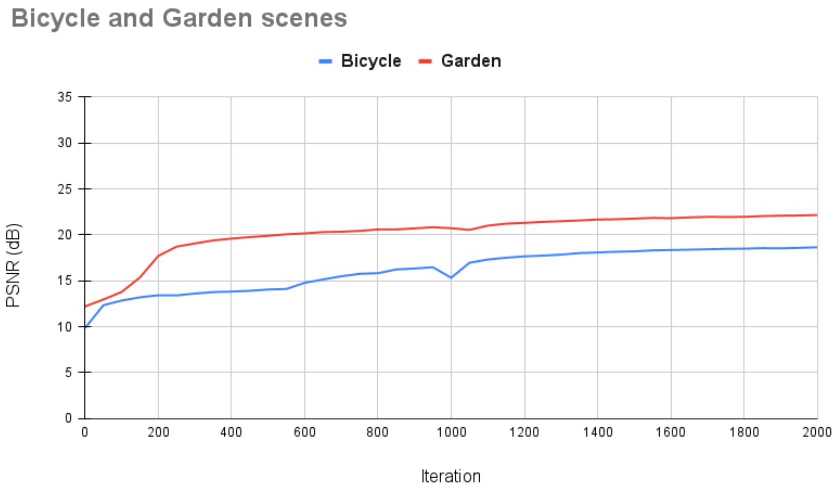


Figure 5.4: This graph shows the growth of the PSNR score during the training process of the outdoor scenes bicycle and Garden.

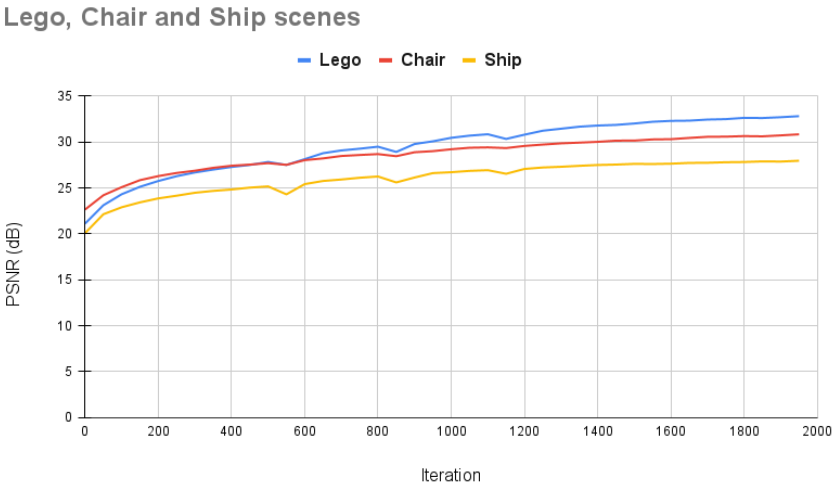


Figure 5.5: This graph shows the growth of the PSNR score during the training process of the synthetic scene Lego, Chair and Ship.

Chapter 6

Conclusion

6.1 What was Achieved

This thesis has been focused on the modern rendering methods used in Novel View Synthesis algorithms, which utilize 3D Gaussian volumetric particles for the scene representation. I have defined commonly used methods for representing the said 3D particles and explained their role in image synthesis. I have also explored the common training pipeline, which is used in many different projects in the same field, and explain each component of the pipeline. I have mainly focused on the rendering component of the training pipeline, where I explored three different approaches which can be used to render a scene compromised of Gaussian particles.

6.2 Advantages and Disadvantages of Raytracing

As was shown in the previous chapter, the raytracing approach for rendering 3D Gaussian has the advantage in quality. Scene trained with a raytracing pipeline (opposed to the rasterizing pipeline, for example) result in better visual scores with often more compact scene sizes. Because of this, the scenes can be stored more easily.

Another advantage of raytracing is the easier ability to combine different representations of visuals. This means that the raytracing pipeline can be expanded to also show 3D models with standard mesh representation inside of the trained Gaussian scenes.

The main disadvantage of the raytracing approach is the performance of the rendering. Raytracing approaches for Novel View synthesis can be 10x or more slower than rasterization.

6.3 Possible improvements

There are many different interesting topics exploring different uses of the 3D Gaussian scene or building on top of the topics explored in this thesis, which would be interesting to explore and document in more detail, such as relightable scenes.

The custom implementation of the rendering pipeline could also be improved. There are many optimizations that can be used during the rendering process that could improve the rendering time, which are not used in this project due to a lack of time. There can also be better ways to utilize the NVIDIA OptiX framework, which would allow even better GPU performance, which I do not know about as an amateur in OptiX because of the lack of study materials on the topic.

The custom implementation could also show some of the advantages of raytracing, such as the ability to easily add custom models to the trained scenes or the ability to simulate different camera effects.

The resulting pipeline could also have been tested in a larger number of scenes, which could show more weaknesses or strengths of the chosen approach.

Bibliography

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuhler, George Drettakis *3D Gaussian Splatting for Real-Time Radiance Field Rendering*, ACM Trans. Graph., 2023.

[2] Jiang, Yingwenqi and Tu, Jiadong and Liu, Yuan and Gao, Xifeng and Long, Xiaoxiao and Wang, Wenping and Ma, Yuexin *GaussianShader: 3D Gaussian Splatting with Shading Functions for Reflective Surfaces*, ArXiv preprint, 2023.

[3] Gao, Jian and Gu, Chun and Lin, Youtian and Zhu, Hao and Cao, Xun and Zhang, Li and Yao, Yao *Relightable 3D Gaussians: Realistic Point Cloud Relighting with BRDF Decomposition and Ray Tracing*, ArXiv, 2023.

[4] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, Angjoo Kanazawa. *Gsplat: An Open-Source Library for Gaussian Splatting*, ArXiv, 2024.

[5] Nicolas Moenne-Loccoz and Ashkan Mirzaei and Or Perel and Riccardo de Lutio and Janick Martinez Esturo and Gavriel State and Sanja Fidler and Nicholas Sharp and Zan Gojcic *3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes*, ACM Transactions on Graphics and SIGGRAPH Asia, 2024.

[6] Hugo Blanc and Jean-Emmanuel Deschaud and Alexis Paljic. *Volumetric Gaussian-Based Ray Casting for Photorealistic Novel View Synthesis*, ArXiv, 2024.

[7] Guikun Chen, and Wenguan Wang *A Survey on 3D Gaussian Splatting*, arXiv, 2024.

[8] Zhang, Qiang and Baek, Seung-Hwan and Rusinkiewicz, Szymon and Heide, Felix. *Differentiable Point-Based Radiance Fields for Efficient View Synthesis*, Association for Computing Machinery, 2022.

[9] Mildenhall, Ben and Srinivasan, Pratul P. and Tancik, Matthew and Barron, Jonathan T. and Ramamoorthi, Ravi and Ng, Ren. *NeRF: representing scenes as neural radiance fields for view synthesis*, ACM pages 99-106. 2021

[10] Barron, Jonathan T. and Mildenhall, Ben and Verbin, Dor and Srinivasan, Pratul P. and Hedman, Peter. *Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields.* CVPR. 2022.

[11] Müller, Thomas and Evans, Alex and Schied, Christoph and Keller, Alexander. *Instant neural graphics primitives with a multiresolution hash encoding.* ACM pages 1-15. 2022

[12] Schönefeld, Volker *Spherical harmonics*, Computer Graphics and Multimedia Group, Technical Note. RWTH Aachen University, Germany, 2005.

[13] Aaron Knoll, R Keith Morley, Ingo Wald, Nick Leaf, and Peter Messmer. *Efficient particle volume splatting in a ray tracer.* Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs. 2019

[14] Alain Horé, Djemel Ziou. *Image quality metrics: PSNR vs. SSIM.* International Conference on Pattern Recognition. 2010

[15] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, Oliver Wang. *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric.* CVPR open access. 2018

[16] Alessio Regalbuto *How We Wrote A GPU-based Gaussian Splats Viewer In Unreal With Niagara*, Medium, 2024. medium.com

[17] Puye. *Introduction to Spherical Gaussians.* Puye's Blog. 2023. puye.blog

[18] *NVIDIA OptiX 8.1 Programming Guide*, NVIDIA, 2024.

[19] *NVIDIA OptiX 8.1 API Reference Manua* NVIDIA, 2024.

[20] Anaconda distribution. anaconda.com

[21] ImGui. github.com/imgui. 2014