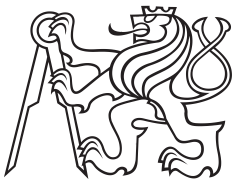
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Ray Tracing 3D Gaussians

Matvii Bunin

Supervisor: Mgr. Jonáš Kulhánek
Field of study: Open Informatics
Subfield: Computer Vision and Image Processing
January 2026

Abstract

This work aims to implement fast ray tracing of 3D Gaussian scenes and demonstrate its advantage over splatting methods in standard, as well as distorted camera rendering setups. To achieve this we analyze the foundations of the current Gaussian splatting and ray based rendering methods to compare them and identify possible improvements. We then design our own implementation of fast ray tracing framework for 3D Gaussians from scratch. This choice was dictated by the absence of any public implementation of our chosen approach at the time when the work was started.

Keywords: gaussian splatting, ray tracing, 3D scene reconstruction, novel view synthesis

Supervisor: Mgr. Jonáš Kulhánek

Abstrakt

Tato práce si klade za cíl implementovat rychlé sledování paprsků (ray tracing) pro 3D gaussovské scény a demonstrovat jeho výhody oproti metodám založeným na splattingu, a to jak při standardním vykreslování kamerou, tak i v režimech se zkreslenou kamerou.

Za tímto účelem analyzujeme principy současných metod gaussovského splattingu a vykreslování založeného na paprscích, porovnáváme je a identifikujeme možné směry zlepšení. Následně navrhujeme a od základu implementujeme vlastní framework pro rychlé sledování paprsků pro 3D gaussovské reprezentace. Toto rozhodnutí bylo motivováno tím, že v době zahájení práce nebyla veřejně dostupná žádná implementace zvoleného přístupu.

Klíčová slova: gaussovský splatting, sledování paprsků, rekonstrukce 3D scény, syntéza nových pohledů

Překlad názvu: Sledování paprsků pro 3D Gaussiány

Contents

1 Introduction 5

2 Background 9

2.1 Gaussian Weighted Average 9
2.2 Surface splatting 10
2.3 Volume rendering 11
2.4 Evaluating radiative transfer equation 13
2.4.1 Quadrature integration 13
2.4.2 Stochastic sampling 14
2.5 Volume splatting 14
2.5.1 3D Gaussians 14
2.5.2 Volume rendering 15
2.6 Direction dependent emission . . 16
2.7 Optimization 17
2.7.1 Image similarity metrics 19
2.8 Camera models 20
2.9 Geometric 3D reconstruction . . . 24
2.10 Hardware accelerated ray tracing 25

3 Related work 27

3.1 Gaussian Splatting 27
3.1.1 Rasterization 27
3.1.2 Rendering formulation 28
3.1.3 Anti-aliasing 30
3.1.4 Density control 31
3.2 Gradient optimization 34
3.3 Ray based methods 39
3.3.1 Ray marching 40
3.3.2 Analytic integration 40
3.3.3 Fast ray tracing 41
3.3.4 Numerical stability 43
3.4 Distorted Cameras in Gaussian Splatting 48
3.4.1 Unscented transform 49
3.4.2 Renderer 50

4 Implementation 51

4.1 Rendering 51
4.1.1 GPU ray tracing 54
4.2 Gradients 55
4.3 Adaptive density control 56
4.4 Evaluation framework 56
4.5 FisheyeGS 57

5 Results and ablations 61

5.1 Densification 61
5.2 Renderer 64

6 Conclusion and future work 73

Bibliography 75

A Qualitative comparison 81

Figures

2.1 Contractive ("barrel") and expansive ("pincushion") distortions respectively (OpenCV [7][40]) 22
2.2 Fisheye (left) vs unfisheyed image [44] 24

3.1 Cloning and splitting (3DGS [10]) 32
3.2 3DGS differentiable rendering pipeline figure from FisheyeGS [33]. The modified parts refer to the ones influenced by changing the camera Jacobian. 37
3.3 a) splatting integration in linearized ray space; b) maximum contributing samples for isotropic Gaussian, corresponding to closest points to the mean, are distributed along a slice of the Gaussian with a quadratic curve; c) anisotropy produced by switching the metric tensor for inverse covariance. 41
3.4 Fitting and adaptive scaling of the icosahedron mesh (3DGRT[37]) ... 45
3.5 The analysis of higher buffer sizes (3DGRT [37]) 48

5.1 Naive gradients on the *ship* scene 62
5.2 Densification comparison for fisheye and undistorted cameras. The former produce much smaller scenes. Note that fisheye and undistorted dataset versions use different initial point clouds. 64
5.3 Fisheye densification with the modified densification criterion produces a larger scene, behaving similarly to the undistorted case. Both graphs show fisheye..... 64
5.4 Laptop brand is reconstructed in fisheye camera case with the sorting buffer size of 512 instead of 1024. 69

Tables

3.1 Position-based depth sorting, pixel-based, and ray marching. [10] 40
3.2 The difference in footprint error of UT and EWA depending on FOV of fisheye cameras and radial distortion.[56] 49
3.3 The frequency histograms show how often the unscented transform is closer to the correct projection than EWA.[56] 50

4.1 The original fisheye image generated during training does not match the reference. The distortion applied during training and evaluation is shown in blue. 58
4.2 The sorting artifact appearing at small z_c best visible with wide FoV, still contributes for cameras with FoV close to 180° 59

5.1 Exponential cloning on *lego* scene corresponds to growth in norms of positional gradients..... 61
5.2 The default setup (left) compared to wide meshes setup. 62
5.3 Clamped densification on *garden* and *bicycle* scenes. 63
5.4 Rendering the pre-trained 3DGS[37] checkpoint..... 65
5.5 Trained scenes with sample based sorting. The wider meshes show very similar results, while significantly degrading performance..... 65
5.6 Performance comparison for the sorting strategy..... 66
5.7 Comparison of our sample based rolling version to 3DGS on pinhole cameras..... 67
5.8 Partial evaluation on Zip-NeRF dataset [19] 67
5.9 The effect of renderer version and camera models..... 69
5.10 The performance on 3DGS scenes. 70
5.11 The performance on 3DGS scenes. 70

5.12 The performance on our trained scenes. 70

A.1 The effect of sample based sorting on fisheye. 81
A.2 The effect of sample based sorting on fisheye. 82
A.3 The effect of sample based sorting on undistorted pinhole. 83

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Bunin Matvii** Personal ID number: **483467**

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

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Computer Vision and Image Processing**

II. Master’s thesis details

Master’s thesis title in English:

Ray-Tracing 3D Gaussians

Master’s thesis title in Czech:

Ray-Tracing 3D Gaussians

Guidelines:

The primary objective of this thesis is to implement the 3D Gaussian Splatting (3DGS) [2] method utilizing ray tracing instead of the traditional rasterization pipeline. This approach aims to enhance 3DGS reconstruction by accommodating a wider range of camera models or the modeling of reflective surfaces through the use of secondary rays.
To achieve this goal, a comprehensive analysis of current methods related to 3D Gaussian Splatting will be conducted, focusing on both rasterization and ray tracing approaches. This analysis will evaluate the advantages and limitations of each method, particularly in terms of rendering quality and computational efficiency. Following this, a ray-tracing-based 3DGS will be developed utilizing the OptiX library [3]. A comparative analysis between this ray-tracing implementation and the original rasterization-based approach will be performed, assessing both speed and rendering quality.
Finally, the proposed method will be evaluated using appropriate scenarios, such as datasets featuring highly distorted camera models. The results will be compared with relevant baseline methods, including the original 3DGS [2] and derived methods [4]. The evaluation will aim to demonstrate the potential advantages and additional use cases of the ray-tracing-based 3DGS implementation over traditional methods.

Bibliography / sources:

[1] Zwicker, Matthias, Hanspeter Pfister, Jeroen van Baar, a Markus Gross. “Surface Splatting.” Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’01), 2001, 371–378. https://doi.org/10.1145/383259.383300.
[2] Kerbl, Bernhard, et al. “3D Gaussian Splatting for Real-Time Radiance Field Rendering.” ACM Transactions on Graphics 42, no. 4 (2023): 1–12. https://doi.org/10.1145/3592433.
[3] Parker, Steven G., et al. “OptiX: A General Purpose Ray Tracing Engine.” ACM Transactions on Graphics 29, no. 4 (2010): 1–13. https://doi.org/10.1145/1778765.1778803.
[4] Liao, Zimu, et al. “Fisheye-GS: Lightweight and Extensible Gaussian Splatting Module for Fisheye Cameras.” arXiv preprint arXiv:2409.04751 (2024).

Name and workplace of master’s thesis supervisor:

Mgr. Jonáš Kulhánek Applied Algebra and Geometry CIIRC

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

Torsten Sattler, Dr. rer. nat. RICAIP CIIRC

Date of master’s thesis assignment: **28.01.2025** Deadline for master's thesis submission: _____

Assignment valid until: **20.09.2026**

prof. Dr. Ing. Jan Kybic
Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

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

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

DECLARATION

I, the undersigned

Student's surname, given name(s): Bunin Matvii
Personal number: 483467
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 used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc Matvii Bunin
.....
student's signature

Chapter 1

Introduction

Novel view synthesis is one of the central topics in computer vision and computer graphics research, which has recently shown significant advances. Given a set of views of a scene, usually photographs of a real world scene, the problem is formulated as synthesizing a set of unseen views. This has a range of applications, including all kinds of immersive and interactive media, such as augmented and virtual reality, free viewpoint video, cinematography, heritage reconstruction, and remote collaboration. It can also be very valuable in the area of robotic simulations, potentially enabling low-cost, highly realistic virtual environments for reliable testing and development of autonomous robotic systems.

Early set of techniques, termed image based rendering (IBR) [49], where the first to approach the problem as using captured images, rather than hand crafted meshes, as primary source of appearance. In the simplest approach, referred to as 3D warping, the novel view can be computed from a provided nearby view using an estimate homography mapping. However, this approach would disregard the occlusion change provided by depth. Subsequent approaches, therefore, addressed the need for designing depth proxies to describe the underlying structure in the form of surfaces, meshes, or point clouds. An effective method to obtain the latter is based on a set of geometric 3D reconstruction techniques named structure from motion (SfM)[51], which based on matching correspondences between nearby images estimates relative camera positions, and positions of corresponding 3D points, jointly optimizing the whole set to produce sparse point clouds, which can be colored using the provided images and rendered directly. A subsequent set of methods called multi view stereo (MVS)[18] focuses on creating dense point clouds based on large image sets, identifying the appropriate image pairs for matching and utilizing per pixel depth maps to generate 3D points, resulting in a more continuous appearance. However, these methods are computationally demanding because of expensive depth estimation. Subsequent work [6] proposed a faster approach using per-pixel disparity maps optimized via random search to obtain correspondences along epipolar lines.

Dense reconstruction methods are still expensive and produce noise in geometry. They are also prone to failure on flat regions due to high sparsity of initial correspondences and on repetitive structures due to ambiguity.

Optimizing for structure, rather than appearance, these methods also do not address view dependent illumination effects. A more recent approach that opened the venue for radiance field reconstruction was introduced by neural radiance fields (NeRF) [36]. These methods address the limitations of dense reconstruction in terms of appearance quality by spatially parameterizing the light field using multi-layer perceptrons (MLP). These representations can be sampled along camera rays and optimized using standard machine learning approaches. However these models can produce very high quality appearance, while being of small size, they are slow in inference due to evaluation of MLPs, and have memory heavy back-propagation due to large amount of ray samples. An alternative set of approaches ([15],[38]) uses volumetric parameterizations with density- and view dependent illumination parameters interpolated on a scene grid, which results in much faster and cheeper radiance field reconstruction. The fundamental limitation of these methods is their focus on appearance, rather than structure. However the structure can be extracted from the radiance fields, its representation is implicit, which reduces potential applicability of these methods.

Radiance field reconstruction has recently been revolutionized by 3D Gaussian Splatting (3DGS)[10], achieving high rendering speed and reconstruction quality. While earlier methods focused on occupancy grids and implicit neural representations, 3DGS proposed an efficient approach to optimizing a cloud of primitives via differentiable rendering, providing a foundation for a whole family of subsequent methods. The learned primitive based representations have proven their versatility, applied for dynamic scenes ([55]), synthesis and editing ([34],[53]), relighting ([17]), and high quality surface estimation ([23]), as well as integration of radiance fields into existing rendering pipelines ([57]). Despite their advantages, Gaussian scenes still lack structure, have higher memory requirements due to per-particle parameters, and are in some ways restricted by the 3DGS rasterization pipeline. There is a significant body of research that addresses each of these issues and more ([1],[56],[33],[14]). In this work, we are mainly focusing on building an alternative rendering pipeline for Gaussian scenes.

The tradeoff of ray tracing against rasterization has long been known in the rendering community as one of slow direct simulation of light paths, allowing for global illumination effects against fast projection and blending of primitives, combined with a set of screen space tricks to achieve convincing results. In application to learned particle scenes, where view dependent reflections are baked per particle, ray tracing can still expand what is possible by including secondary ray sampling (IRGS [20]), directly supporting non-linear camera models, and allowing for integration with classical ray tracing pipelines (3DGRT [37]). The mentioned performance cost can be addressed by utilizing NVIDIA hardware-supported ray tracing infrastructure, allowing for training and inference speed comparable to rasterization.

This work addresses the derivation, implementation, and evaluation of 3D Gaussian ray tracer, comparing it to 3DGS methods. To demonstrate the advantage of ray tracing, we chose a dataset featuring fisheye cameras,

which cannot be used in rasterization pipelines without an approximation. In chapter 2 we discuss theoretical foundations of gaussian splatting and its relationship to volume rendering. We will also introduce the important concepts used in the later chapters, such as stochastic optimization techniques, the mathematical models for distorted cameras, as well as the modern approach to hardware accelerated ray tracing. The chapter 3 will thoroughly analyze the current splatting and ray tracing methods, including mathematical derivations, algorithms, and numerical stability. Finally, in chapter 4 we will introduce the implemented Gaussian ray tracing approach, as well as an adaptation of gaussian splatting pipeline for wide angle distorted fisheye cameras. We will demonstrate parts of our research process, as well as the results achieved by our method in 5.

Chapter 2

Background

This section aims to examine the foundations behind the volume rendering approaches and learning pipelines used in the radiance fields, discussing the assumptions and simplifications used by the current Gaussian and neural rendering methods.

2.1 Gaussian Weighted Average

Splatting approaches stem from the master thesis of Heckbert [21] that focused on sampling textures under non-affine mappings. Since we cannot directly map texels to pixels, the task of rendering textures involves reconstructing a continuous signal from discreet samples, mapping it into the screen space (the process called warping), and then sampling it at pixel positions. This pipeline known as the *resampling filter* expressed in 2.1 requires applying a reconstruction filter r in the texture space and then a low-pass filter h on the warped signal before pixel sampling to ensure that it meets the Nyquist criterion.

$$\rho(\mathrm{x},\mathrm{k}) = \int_{\mathbb{R}^n} h(\mathrm{x}-\mathrm{m}(\mathrm{u}))r(\mathrm{u}-\mathrm{k})\left|\frac{\partial \mathrm{m}}{\partial \mathrm{u}}\right| d\mathrm{u} \tag{2.1}$$

Computing the above integral per pixel requires evaluating the mapping m and its Jacobian at each texel in the kernel footprint. An ideal resampling filter using sinc functions would additionally require evaluating at all texels for each pixel due to infinite impulse response. The ideal solution is therefore deemed impractical, and some simplifications, namely decimation ($r = \delta$) and magnification ($h = \delta$) filters are used, though it is problematic to choose when to apply which in case of 2D Jacobian.

Instead, the work suggested the use of Gaussian filters for both reconstruction and sampling. Gaussians are closed under convolution, projection, and Fourier transform - making them good low pass filters that can be windowed to a finite impulse response without substantial loss in quality. Additionally, we can avoid multiple mapping evaluations in the texture space using a linear approximation at $u_0 = m_{-1}(x_0)$, replacing the mapping function with $\mathrm{m}(u) = x_0 + J_{u_0}(u - u_0)$. For this linear map around u_0 we get a constant Jacobian $\left|\frac{\partial \mathrm{m}}{\partial \mathrm{u}}\right| = |J_{u_0}|$, so substituting $h'(u) = |J_{u_0}|\, h(J_{u_0}u)$, then applying

$J_{u_0}^{-1}(x-y) = \mathrm{m}^{-1}(x) - \mathrm{m}^{-1}(y)$ we get

$$\begin{aligned}\rho(\mathrm{x_0}, \mathrm{k}) &= \int h'(\mathrm{m}^{-1}(\mathrm{x_0}) - \mathrm{u}) r(\mathrm{u} - \mathrm{k}) d\mathrm{u} \\ &= \int h'([\mathrm{m}^{-1}(\mathrm{x_0}) - \mathrm{k}] - \mathrm{u}) r(\mathrm{u}) d(\mathrm{u} + \mathrm{k}) \\ &= (h' \star r)(\mathrm{m}^{-1}(\mathrm{x_0}) - \mathrm{k})\end{aligned}$$

We see that this approximation relies on the filter h', obtained by non-uniform scaling of h. A natural choice for this filter would therefore be one that works independently of the scaling direction, which suggests a circular support in the image space, resulting in an elliptical footprint in the texture space. The approach is called Elliptically Weighted Average (EWA), which covers general profile elliptical footprint anti-aliasing filters.

In case if both filters are 2D Gaussians with variances V_r and V_h we get a resampling filter represented by a single Gaussian.

$$p(\mathrm{x_0}, \mathrm{k}) = G_{J^{-1}V_h J^{-1T} + V_r}(\mathrm{m}^{-1}(\mathrm{x_0}) - \mathrm{k}) \tag{2.2}$$

Since the effective support of an anisotropic Gaussian is an ellipse, this is a special case of an EWA filter with nice properties.

2.2 Surface splatting

While the Heckbert EWA framework that uses source space sampling works well for textured polygons, the approach is not as simple and efficient for rasterizing point clouds, as discussed in a subsequent work by Zwicker et al. [64] . In their setup points are obtained from a laser scanner, each k-th point is assigned a position in 3D, weight coefficients w_k for color channels, and a normal vector n_k. We could then use a 2D surface parameterization in which the texture signal is reconstructed and apply the gaussian resampling filter from 2.2, producing the resampled signal for each point $\mathrm{u_k}$ that contributes to $\mathrm{u} = \mathrm{m}^{-1}(\mathrm{x})$ as

$$g(\mathrm{x}) = \sum_k w_k G_{J^{-1}V_h J^{-1T} + V_r^k}(\mathrm{m}^{-1}(\mathrm{x}) - \mathrm{u_k})$$

The problem here is in determining the point u on a point cloud surface, which is not defined in the first place, and then finding its contributing samples with respect to a local parameterization. This would only make sense in a ray tracing setup.

Instead, the paper proposes to bring the resampling filter to the screen space by warping the reconstruction filter instead of the low-pass filter, resulting in

$$p_k(\mathrm{x}) = (h \star r'_k)(\mathrm{x} - \mathrm{m}(\mathrm{u_k}))$$

The reconstruction filter r_k is defined in an approximation plane with normal n_k and is an isotropic Gaussian scaled depending on distances to nearest

samples. For a linear projection matrix we can derive an inverse projection Jacobian for this plane, which would transform the Gaussian to screen space, obtaining r'_k. The resampling filters can then be rendered, aggregating the normalized contributions from each projected point per pixel. In case of opaque surfaces, we would use simple depth thresholding to determine the visible Gaussian per pixel. To allow for transparency, the authors propose fixed size per-pixel buffers that would be blended front-to-back during rendering. When the buffer is full, we would simply blend a new contribution with the one with the closest depth of those stored.

This filter rendering algorithm, called *surface splatting*, can be applied for each color channel, normals, and store additional shading information per pixel. The authors have also proposed a modification for texture point clouds, storing texture sample coordinates per point, which requires additional texture filtering and optimizing point weight parameters to minimize mapping error. However, this will not play a role in this work, since having per point colors is more suitable for learned point clouds.

The method can be generalized for volume rendering by moving to 3D Gaussians, as will be described in the following sections.

2.3 Volume rendering

Unlike classical rendering techniques that rely on proxies in the form of surface representations, volume rendering aims to directly simulate the effect of light traveling through a medium, the phenomenon described by radiative transfer theory. The survey by Nelson Max [35] of the approximations used for this simulation presents approaches with increasing accuracy depending on the simulated phenomena, namely absorption, emission, volumetric scattering, and shadows. The last two are not widely used by splatting methods, mostly focusing on surface based approaches, while the volumetric rendering model including absorption and emission is fundamental in modern radiance fields.

To see where this rendering model comes from, we start by defining the volume as a density field of small spherical particles with a rendered footprint area A. For each ray with parameter s we then define a density function $\rho(s)$. The fraction of light passing through the medium is called *transmittance*, which can be found as $T = \frac{\Phi_{out}}{\Phi_{in}}$, where $\Phi[W]$ is the *radiant flux* or *radiance* $I[\frac{W}{m^2 \cdot sr}]$ integrated through the area and the solid angle of ray directions. Therefore, to find transmittance we intermediately define a cylindrical beam with a base area E of parallel rays with equal radiance, and consider its slab of depth Δs. Since the rays are perpendicular to the base, we choose $I(w)$ appropriately, so the flux through the base is reduced as

$$\Phi = \int_w \int_B I(w) cos\theta dA dw = \int_B I dA = IE$$

The number of particles in the slab is

$$N = \rho(s)V = \rho(s)E\Delta s$$

We assume that the projections of particles to the base of the beam do not overlap and fully absorb light energy.

$$1 - T = \frac{\Phi_{in} - \Phi_{out}}{\Phi_{in}} = \frac{NA}{E} = \rho(s)A\Delta s = \sigma(s)\Delta s$$

where $\sigma(s)$ is called *extinction function* or simply *opacity*. Then applying $T = \frac{I_{out}}{I_{in}}$ we get

$$\Delta I(s) = -\sigma(s)I_{in}\Delta s$$

This when shrinking the slab gives us the differential equation for absorption

$$\frac{dI}{ds} = -\sigma(s)I$$

We can apply separation of variables, then solve for initial conditions, resulting in

$$I(s) = I(0)e^{-\int_0^s \sigma(t)dt} = I(0)T(s)$$

Here we introduce the transmittance factor (a.k.a. the transparency factor) $T_f(s)$, which we will denote "forward", as it represents the leading transmittance along the light path.

We can now extend this model by adding emission. Assume that each particle emits *radiant exitance* $c(s)[\frac{W}{m^2}]$, contributing to the output flux in the projected area NA as $\Phi'_{out} = \Phi_{out} + c(s)NA$.

$$\frac{\Phi_{in} - \Phi'_{out}}{\Phi_{in}} = \sigma(s)\Delta s - \frac{c(s)\sigma(s)\Delta s}{I_{in}}$$

Similarly to the previous, this after rearrangement results in our target differential equation

$$\frac{dI}{ds} = c(s)\sigma(s) - \sigma(s)I$$

The solution to this is less straightforward but analogous. The key is to use a definite integral $\int_0^{s_{max}}$ with $s_{max} > s$, which effectively accounts for the contribution of light emitted by the particles along the whole ray. This gives us

$$I(s_{max}) = I(0)T_f(s_{max}) + \int_o^{s_{max}} c(s)\sigma(s)T_b(s, s_{max})ds$$

where the residual transmittance is $T_b(s, s_{max}) = e^{-\int_s^{s_{max}} \sigma(t)dt}$.

This form of the *radiative transfer equation* (RTE) is usually presented in the radiance field methods. The factor $I(0)T(s_{max})$, which represents the background contribution, is usually omitted since it does not play any role in real datasets.

Note that the integral is derived in the light direction with s_{max} corresponding to the eye. Evaluating it back-to-front along the camera ray is easier, as we can accumulate the residual transmittance T_b. The actual rendering process can be performed either by ray tracing or rasterization, as we will show later.

2.4 Evaluating radiative transfer equation

Although some simplifying assumptions can be made to compute the transfer integral analytically, as will be presented in the next section, in general we have to rely on numerical integration methods. These can be broadly split into deterministic family, essentially represented by quadrature methods, and non-deterministic Monte Carlo sampling based methods.

2.4.1 Quadrature integration

This family is defined by evaluating a weighted sum of function values at n points in the integration interval.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

The methods differ in order of convergence, defined by the speed at which the approximation error decreases with finer sampling. Newton-Cotes methods take equally spaced samples, subdividing the integration space into intervals of length h. Their approximation error for order p is bounded as

$$\left| I - \hat{I} \right| \leq Ch^p(b-a)\max|f^{(p)}|$$

where $|f^{(p)}| = \sup f^{(p)}$, and C is an independent constant. This shows that higher order methods require smoother functions, which makes them impractical for most rendering applications. It is possible to achieve even faster convergence with Gaussian quadratures, but the smoothness assumption is still analogous. Typically used in volumetric rendering are first- and second order Newton-Cotes variants, namely the rectangle or trapezoid rule.

The standard approach proposed in the classical NeRF[36] follows the rectangle rule. It assumes that color $c(s)$ and opacity $\sigma(s)$ are constant at each interval. Substituting c_i and σ_i for each interval of length δ_i, we get

$$\begin{aligned} I_i &= \int_0^{\delta_i} c_i\sigma_i \exp\left(-\int_s^{\delta_i}\sigma_i dt - \sum_{j>i}\int_0^{\delta_j}\sigma_j dt\right) ds \\ &= c_i e^{-\sum_{j>i}\sigma_i\delta_i}\int_0^{\delta_i}\sigma_i e^{-\sigma_i(\delta_i - s)} \end{aligned}$$

since $\frac{d}{ds}e^{-\sigma_i(\delta_i-s)} = \sigma_i e^{-\sigma_i(\delta_i-s)}$, the integral becomes $[1-\exp(-\sigma_i\delta_i)]$. Summing all intervals, we get the approximation

$$\begin{aligned} I &= \sum_{i=1}^{N} T_i(1-\exp(-\delta_i\sigma_i))c_i \\ T_i &= \exp\left(-\sum_{j=i+1}^{N}\delta_j\sigma_j\right) \end{aligned} \tag{2.3}$$

For all intervals, the length δ_i is chosen equal. Although adaptive subdivision is possible, in learning setups it is replaced by importance sampling. More recent work PL-Nerf [52] notes that assuming opacity constant in each interval leads to artifacts due to different quantization across rays, proposing a linear approximation instead.

2.4.2 Stochastic sampling

Taking deterministic samples per interval, as in classic quadrature, leads to a systematic error that can only be reduced by a finer subdivision. Quadrature also suffers from the curse of dimensionality, making convergence exponentially slower with higher dimensions, which can come from evaluating a function over multiple rays. For these reasons, most ray tracing applications rely on Monte Carlo integration instead, which does suffer these limitations and is agnostic to function smoothness. The estimate is unbiased, although the error as a function of the number of samples is only in $O(\frac{1}{\sqrt{N}})$.

One of the classic techniques for reducing estimate variance is to place samples in equally spaced bins. This technique called *stratified sampling* is structurally the same as the equally spaced quadrature derived in the previous section. The only difference is that, instead of taking samples deterministically, we draw them from a uniform distribution

$$s_i \sim \mathcal{U}\left[0, \delta_i\right]$$

then evaluate $\sigma_i = \sigma(s_i)$ and $c_i = c(s_i)$.

2.5 Volume splatting

A different volume rendering approach has been proposed by Zwicker et.al.[63] as a generalization of their surface splatting framework. Unlike the numerical integration method described previously, it relies on multiple simplifying assumptions to derive an analytic RTE approximation, as will be demonstrated here.

2.5.1 3D Gaussians

The problem is defined as rendering a set of samples of a 3D color and density field, which may be evenly or unevenly spaced. In surface splatting, the samples were considered evenly spaced on the surface, which allowed isotropic reconstruction filters. Here, however, we have to use anisotropic reconstruction filters for the signal to remain smooth when the sample sparsity varies across directions. Then based on the same gaussian EWA reasoning, both the reconstruction filter and screen space isotropic low pass filter are chosen to be Gaussian, producing an elliptical footprint in screen space. In volumetric splatting instead of defining a 2D Gaussian on a surface tangent

plane, we produce it by integrating the 3D resampling filter along one axis.

$$G_\Sigma(\mathbf{x}-\mathbf{p}) = \frac{1}{2\pi\,|\mathbf{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{p})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{p})}$$

$$G_{\hat{\Sigma}}(\hat{\mathbf{x}}-\hat{\mathbf{p}}) = \int G_\Sigma(\mathbf{x}-\mathbf{p})dx_3 \tag{2.4}$$

For splatting the integration axis should be depth, which is achieved by transforming the reconstruction filter to camera space, and then to ray space. The transformation from world to camera is an affine mapping

$$\phi(\mathbf{x}) = \mathbf{W}(\mathbf{x}-\mathbf{C}) \tag{2.5}$$

with Jacobian $\mathbf{W}$ for a world-to-camera matrix $\mathbf{W}$ and camera center $\mathbf{C}$.

The transformation from camera to ray space is defined by

$$\mathbf{m}(\mathbf{x_c}) = \left(\frac{x_1}{x_3}, \frac{x_2}{x_3}, \|(x_1,x_2,x_3)\|\right)^T$$

This is not an affine mapping, so for the filter to remain a Gaussian, the mapping has to be linearized around the center as $\mathbf{m_{p_c}}(\mathbf{x_c}) = \mathbf{p_c}+\mathbf{J_{p_c}}(\mathbf{x_c}-\mathbf{p_c})$, where $J_{p_c} = \frac{\partial \mathbf{m}}{\partial \mathbf{x_c}}(\mathbf{p_c}) = J$ is camera space Jacobian. The desired ray space reconstruction filter is then obtained as

$$G_{\Sigma_w}(\mathbf{m}^{-1}(\phi^{-1}(\mathbf{x}-\mathbf{p}))) = \frac{1}{|\mathbf{W}^{-1}|\,|\mathbf{J}^{-1}|} G_\Sigma(\mathbf{x}-\mathbf{p}) \tag{2.6}$$

$$\mathbf{\Sigma} = \mathbf{J}\mathbf{W}\mathbf{\Sigma_w}\mathbf{W}^\mathbf{T}\mathbf{J}^\mathbf{T} \tag{2.7}$$

Since the depth component of this normalized Gaussian integrates to one, the depth integral 2.4 is a Gaussian with covariance matrix $\hat{\Sigma} = \Sigma_{2\times 2}$ obtained by dropping the last row and column, so $\hat{\mathbf{x}} = [x_1, x_2]$.

$$\int G_\Sigma(\mathbf{x})dx_3 = G_{\hat{\Sigma}}(\hat{\mathbf{x}}) \tag{2.8}$$

Assuming a simplified camera model, this covariance matrix is screen space. Following the EWA splatting approach, the Gaussian is convolved with a low-pass filter giving $\hat{\Sigma}+\Sigma_h$, and then inverted to compute the density, giving what is called *conic matrix*.

Note that this ray space approximation effectively corresponds to coherent rays. As the incoherence increases, either with the scale of the covariance matrix or with the camera distortion, it becomes less precise.

2.5.2 Volume rendering

In gaussian splatting setup, the scene density is parametrized by the weighted sum of reconstruction filters as

$$\sigma(s) = \sum_k g_k r_k(s)$$

substituting this into the RTE gives

$$I(s) = \sum_k \int_0^{s_{max}} c(s) g_k r_k(s) T(s) ds$$
$$T(s) = \prod_j \exp\left(-\int_s^{s_{max}} g_j r_j(t) dt\right)$$

To integrate this, the authors made several simplifying assumptions:

- the reconstruction kernels have local support, do not overlap, and are sorted along the ray;
- emission $c(s) = c_k$ is constant along the ray segment inside the kernel support;
- self-occlusion (transmittance accumulated within the corresponding ray segment) is ignored;
- the contribution of each kernel $g_j r_j(t)$ to the transmittance factor T is small, allowing the exponent to be linearized as $e^x \approx 1 + x$.

Applying these, the integral turns into

$$I = \sum_k c_k g_k q_k \prod_{j>k} (1 - g_j q_j) \tag{2.9}$$

where $q_i = \int r(s) ds$.

To apply the EWA anti-aliasing pre-filter $I \star h$, the convolution has to be pushed through the equation, leading to $q_k \star h$. This, however, requires additional assumptions of both the transmittance factor and color being constant across the kernel footprint. These assumptions still explicitly or implicitly hold in modern gaussian splatting methods, although low pass filtering is performed differently. Notice that the derived equation 2.9 is structurally the same as the one obtained by numeric integration 2.3, as can be shown by substituting

$$\alpha_i = (1 - e^{-\delta_i \sigma_i}) = g_i q_i \tag{2.10}$$

3D Gaussians are therefore essentially treated as defining uneven quadrature regions along the ray. A detailed analysis by A.Celarek et.al.[10] shows that the approximations used here and in subsequent 3DGS[10] are largely mitigated by learning pipelines in big scenes.

2.6 Direction dependent emission

When parameterizing volumetric radiance fields, it is typically assumed that each region can emit differently depending on the view direction. In learning setups, this allows us to capture view-dependent shading effects without actually simulating them.

The problem is formulated as defining an arbitrary color component function on a unit sphere. Assuming that the function is "well behaved", it can be expressed using a spherical harmonic (SH) series ([48],[15],[16]), analogously to such a function on a unit circle being expressed with a Fourier series. Both can be derived from solving the Laplace equation in radial coordinates, giving a set of basis functions.

Classes of complex functions can be studied as generally infinite dimensional closed spaces with a scalar product

$$f \cdot g = \int_{\Omega} f(x)\overline{g(x)}w(x)dx$$

for some chosen measure weight $w(x)$ and domain Ω; these spaces are called Hilbert spaces. Spaces that further satisfy the finite norm $\|f \cdot g\| < \infty$ are called L^2 spaces. For such spaces, it holds, according to the Riesz–Fischer theorem, that for infinite orthonormal bases e_i

$$\|f - \sum_{i}^{N} e_i(f \cdot e_i)\| \to 0$$

for $N \to \infty$. The norm of this approximation $\sum_i^N |f \cdot e_i|^2$ always increases with the number of basis functions, decreasing the L_2 error. Since the basis functions of SH series are orthonormal, we can thus approximate any L^2 function on a unit sphere, the precision increasing with the number of elements. Similarly to the Fourier series, higher degree coefficients will correspond to higher frequencies, so finite series will correspond to band-limiting.

A spherical harmonic basis function of degree $l \in \mathbb{Z}$ and order $m \in \{-l...l\}$ is defined as

$$Y_l^m(\theta, \phi) = N_l^m P_l^m(\cos\theta)e^{im\phi}$$

where $P_l^m(x)$ is an associated Legendre polynomial, N_l^m is the normalization constant. The function is then expressed as a linear combination over all possible orders up to the maximum degree l_{max} with coefficients β_l^m

$$c_\beta(\theta, \phi) = \sum_{\ell=0}^{\ell_{\max}} \sum_{m=-\ell}^{\ell} \beta_\ell^m Y_\ell^m(\theta, \phi) \tag{2.11}$$

In implementations, the function is moved to Euclidean coordinates with $\|\mathbf{d}\| = 1$, and the coefficients N_l^m are precomputed. Since the function $c_\beta(\mathbf{d})$ is meant to represent light energy, in learning setups it has to be remapped to positive values, which is achievable, since the function is even and has a bounded response for bounded coefficients.

2.7 Optimization

The problem of reconstructing a 3D radiance field representation based on a set of camera views is formulated as novel view synthesis. Assuming no

depth supervision, the setting can be summarized as solving

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \mathcal{D}(I_C(\Theta), \hat{I_C}) \tag{2.12}$$

where camera image $I_C = \bigoplus_{r \in C} I_r$ for an incoming ray radiance $I_r : \Theta \to \mathbb{R}_{\geq 0}$ as a function of some scene parameterization Θ for a set of camera rays C. The loss is defined per image due to the use of structural image metrics $\mathcal{D}$. Restricting to per-pixel metrics, the loss can also be defined as

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \mathop{\mathbb{E}}_{r \in C} \|I_r - \hat{I}_r\|_p \tag{2.13}$$

To solve this given a set of training views $\mathcal{C}$ with corresponding ground truth images, we have to apply an iterative method. In doing so, we expect the converged result to get closer to the optimal solution on a set of novel views, thereby exhibiting generalization. The parameter space is very high dimensional in general, but in our case we assume $\mathcal{L}(\Theta)$ to be differentiable, which makes the problem approachable via first-order methods, which proved their efficiency in deep learning settings.

Common gradient descent (GD) starts with some initialized parameters and iteratively takes the optimal proximal step with respect to the local linear approximation, namely $-\alpha \nabla_\theta \mathcal{L}(\theta_t)$. Assuming that the loss gradient is bounded by the Lipschitz condition $\|\nabla\mathcal{L}(\theta) - \nabla\mathcal{L}(\bar{\theta})\|_2 \leq L\|\theta - \bar{\theta}\|_2$ with $L > 0$, it is guaranteed to decrease the function for small $\alpha \leq \frac{1}{L}$. For multilayer perceptrons this condition applies almost everywhere, while in our case of primitive rendering the discontinuities will be larger.

For non-convex training objectives on large datasets, stochastic gradient descent (SGD) is used, as it offers better exploration of the loss landscape while keeping each training step cheap. It estimates the gradients on a uniformly sampled subset of training samples at each iteration, giving an unbiased estimate of the true gradient. Having a high variance of this estimate causes training to become unstable as it gets far from the Lipchitz gradient, therefore requiring smaller α, and the converged result will be noisy. To address this several techniques may be applied:

- Increasing the sample size n times reduces variance n times under the assumption that the samples are independent. Note that especially for the ray based objective 2.13 samples may be highly correlated.
- Decreasing the learning rate over iterations via some tuned schedule reduces the variance of the final estimate. The drawback is slower convergence near the end. For specific asymptotic schedules, this approach leads to the same convergence guaranties on locally quadratic neighborhoods as GD, reducing noise to zero; however, in practice this is usually irrelevant. Instead commonly used are piecewise-constant ("waterfall"), cosine curve, or exponential schedules.

- Use a running average of gradients to produce a smoother estimate. This introduces a variance-hysteresis trade-off. While lower variance leads to faster and more stable convergence, larger hysteresis can lead to convergence failure or drive it away from local minima. The technique is commonly combined with decreasing the learning rate, which mitigates this effect.

The general formulation of the moving average for the gradient estimate $\tilde{g}_t$ is given by

$$u_{t+1} = \beta_t u_t + (1 - \beta_t)\tilde{g}_t$$

$$\theta_{t+1} = \theta_t - \alpha_t u_t$$

For $\beta_t = \beta \in [0, 1]$ this is known as exponential moving average (EMA), equivalent to SGD with momentum. Starting with $u_0 = 0$ we get $u_t = (1-\beta)\sum_{k=1}^{t} \beta^{t-k}\tilde{g}_t$. If the gradients are the same at each step and $\tilde{g}_t = \tilde{g} = \mathbb{E}_\theta g(\theta)$, this gives $u_t = (1-\beta)\sum_{k=1}^{t} \beta^{t-k}\tilde{g}_t = (1-\beta)\tilde{g}\frac{1-\beta_t}{1-\beta} = \tilde{g}(1-\beta^t)$, which means that the unbiased gradient estimate becomes biased. In practice, this produces a systematic error in earlier iterations, which disappears as t goes to infinity.

As discussed previously, the well-chosen learning rate in GD is constrained by a region in which a function is expected to be smooth, so the gradient is to be "trusted", allowing for higher learning rates. Since the scale of the gradient itself depends on the scale of parameters, the optimal step given the trust region is $\alpha\frac{g_t}{\|g_t\|_2}$. However, this would eliminate adaptation to local curvature, calling for an adaptive approach to normalization.

Adam [27] optimization algorithm solves these problems by using unbiased EMA gradient approximations, tracked with separate β parameters for gradients and for normalizations, resulting in

$$m_{t+1} = \beta_m m_t + (1 - \beta_m)\tilde{g}_t$$

$$v_{t+1} = \beta_v v_t + (1 - \beta_v)\tilde{g}_t^2$$

$$\theta_{t+1} = \theta_t - \alpha\frac{m_t/(1-\beta_m^t)}{\sqrt{v_t/(1-\beta_v^t)} + \epsilon}$$

Note that the effective step size is still not scale invariant, but depends linearly instead of quadratically. Some form of bias is also still present, as it can miss unexpected big steps due to normalization. The letter may cause the convergence to fail on certain simple convex problems and may lead to suboptimal solutions in practice.

2.7.1 Image similarity metrics

In the area of radiance fields, several metrics are used to compare images:

- Mean absolute error (MAE) $\frac{1}{N}\sum_{i=1}^{N}|I_i - \hat{I}_i|$, also known as L1 loss.
- Mean square error (MSE) $\frac{1}{N}\sum_{i=1}^{N}(I_i - \hat{I}_i)^2$, also known as L2 loss.

- Peak signal to noise ratio (PSNR) $10 \log_{10} \frac{1}{\sqrt{\mathbf{MSE}}}$ for images in the range of $[0, 1]$ denotes the average per-pixel distance error in dB scale. This metric is used for benchmarking and is sensitive to pixel noise, color differences, and blur.

- Structured image similarity (SSIM) [54] uses statistics of the image patches to evaluate difference in structural appearance. This metric is more correlated to the human perceptable quality of the image, but may not completely reflect it.

$$D(I, \hat{I}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C2)}$$

- Learned perceptual image patch similarity (LPIPS) [62] this evaluation metric is designed to match human perception. It computes the square difference in output of a pretrained CNN backbone, specifically VGG[50] or AlexNet [29], then computes the mean output of linear layers based on this difference. The comparison is therefore run over a large receptive field, incorporating perceptually important features.

2.8 Camera models

The most basic perspective camera model is known as *pinhole* camera. In this work we will also use distorted and *fisheye* cameras. Since splatting only works with *orthographic* cameras, we will also demonstrate the first order approximation for the camera models.

Projection

The perspective camera model is given by its extrinsic functions defined by the camera transformation in space and by intrinsic functions which specify the image formation inside the camera. The world-to-camera extrinsic function is a linear transformation in homogeneous coordinates $\bar{\mathbf{x}}$ given by the 3x4 world-to-camera matrix $\mathbf{x_c} = \begin{bmatrix} R & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}$.

The pinhole camera model can be characterized by applying the perspective non-linearity

$$\mathbf{x_u} = \mathbf{x_c}/z_c = [u, v, 1] \tag{2.14}$$

corresponding to camera rays intersections with the plane parallel to the image plane and unit distance from the camera center.

Alternatively, the common equidistant fisheye camera model can be thought of as mapping the points $\mathbf{x_c}$ to their camera ray intersections with a unit sphere, so the image is created on a sphere lobe in the coordinate system of the unit sphere surface centered at the intersection with camera z-axis with orthogonal basis vectors along the camera x and y directions. The distance

from the image center then corresponds to the length of the arc between the camera ray and the optical axis, which for the unit circle is

$$\theta = \arctan \frac{\|x_c, y_c\|}{z_c}$$

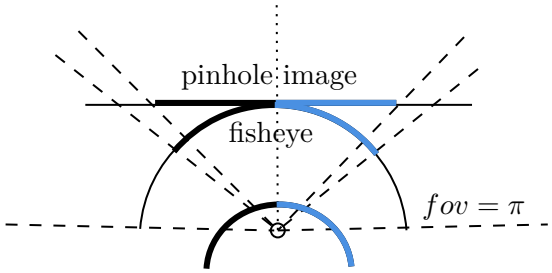
The image point coordinates can then be obtained by rescaling as

$$\mathbf{x_u} = \begin{bmatrix} \frac{x_c}{\|x_c,y_c\|}\theta & \frac{y_c}{\|x_c,y_c\|}\theta & 1 \end{bmatrix}^T \tag{2.15}$$

Note that unlike in pinhole cameras, the angle θ does not have to be restricted to $\pi/2$, as we can use the quadrant aware angle, extending the camera view up to 360°.

FoV

The use of fisheye cameras is motivated by their wider effective FoV, as can be seen in 2.2.



The FoV of a pinhole camera in x axis is given by $2\theta_x$ where $\theta_x = \arctan \frac{W}{2f}$. Given the image size, it is a common intuitive way to define the camera intrinsics in computer graphics, as increasing FoV decreases focus distance. When these intrinsics are used in the fisheye camera, the half image size corresponds to the arc length of the angle θ, or $\frac{W}{2} = f \tan\theta_x = f\theta$. The effective FoV therefore grows faster than pinhole FoV, exceeding the clipping space $z_c > \epsilon$ used in pinhole cameras.

Distortion

Some cameras may show distortion caused by uneven lens curvature, tilt, or index of refraction. Pinhole camera distortion following the Brown [9][8] model is parametrized using $r = \|u, v\|$ as:

- Radial distortion due to lens curvature. The coefficients k_i have to be monotonic and can be positive, zero, or negative. The sign of dominating coefficients will lead to contraction or expansion of the projected image 2.1. The monotonic increase in coefficients will correspond to a higher distortion rate near the edges. Typically, the barrel distortion is done with decreasing coefficients and the pincushion with increasing coefficients.

$$u' = u\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

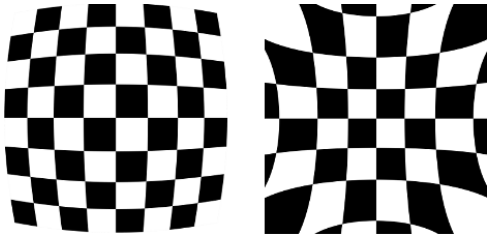


Figure 2.1: Contractive ("barrel") and expansive ("pincushion") distortions respectively (OpenCV [7][40])

- Tangential distortion (due to tilt) is applied after radial distortion:

$$u'' = u' + \left[2p_1u'v' + p_2\left(r^2 + 2u'^2\right)\right]$$
$$v'' = v' + \left[p_1\left(r^2 + 2v'^2\right) + 2p_2u'v'\right]$$

The general OpenCV [7] distortion model [40] deviates from this, featuring thin-prism distortion coefficients s_i and non-polynomial transformations.

$$\begin{bmatrix} u\prime \\ v\prime \end{bmatrix} = \begin{bmatrix} u\frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1uv + p_2\left(r^2+2u^2\right) + s_1r^2 + s_2r^4 \\ v\frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + p_1\left(r^2+2v^2\right) + 2p_2uv + s_3r^2 + s_4r^4 \end{bmatrix}$$

In practice, this model is usually only parameterized by $[k_1,k_2,p_1,p_2]$.

The distortion model used in OpenCV[7] fisheye cameras [39] is Kannala-Brandt polynomial distortion [24], similar to radial distortion, applied at the angle with the optical axis θ:

$$\theta' = \theta(1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8)$$

For mild distortions, the Brown model can be used as an approximation, with tangential coefficients close to zero.

Calibration

The actual image creation process further includes transforming the points to the image space in normalized pixel coordinates. This transformation is defined by the upper-triangular calibration matrix K, defined by the pixel scale $[p_x,p_y]$, focus distance f, the image center pixel position $[c_x,c_y]$, and pixel sheer γ. In our case it can be expressed as 2x3 matrix

$$K = \begin{pmatrix} \frac{f}{p_x} & -\frac{f\cot(\gamma)}{p_x} & c_x \\ & \frac{f}{p_y\sin(\gamma)} & c_y \end{pmatrix}$$

In practice, pixels are usually rectangular $\gamma = \pi/2$, so the camera intrinsics are given by $[f_x = \frac{f}{p_x}, f_y = \frac{f}{p_y}, c_x, c_y]$. Furthermore, in modern cameras, the pixels are usually square, so $p_x = p_y$.

For the perspective camera model, the camera transformation Jacobian 2.5 used in Gaussian splatting is $\mathbf{W} = R$ and ray space mapping corresponds to

$\mathbf{x_u} = \mathbf{m}(\mathbf{x_c})$ after dropping the last coordinate during integration. In general, $\mathbf{m}$ can be defined for distorted and fisheye cameras, and non-unit calibration can be accounted for, additionally transforming by the calibration Jacobian

$$K_{2\times 2} = \begin{pmatrix} f_x & \\ & f_y \end{pmatrix}$$

First order approximation

For pinhole projection 2.14, the linear ray space approximation will then be

$$\mathbf{J_{x_c}} = K_{2\times 2} \begin{pmatrix} \frac{1}{z_c} & & -\frac{x_c}{z_c^2} \\ & \frac{1}{z_c} & -\frac{y_c}{z_c^2} \end{pmatrix} \tag{2.16}$$

The equivalent for fisheye cameras is obtained by differentiating 2.15, which results in

$$\begin{aligned} \mathbf{J_{x_c}} &= K_{2\times 2} \begin{pmatrix} x_c^2 a + y_c^2 b & x_c y_c (a-b) & -\frac{x_c}{\|\mathbf{x_c}\|^2} \\ x_c y_c (a-b) & y_c^2 a + x_c^2 b & -\frac{y_c}{\|\mathbf{x_c}\|^2} \end{pmatrix} \\ a &= \frac{z_c}{\|\mathbf{x_c}\|^2 \|x_c, y_c\|} \qquad b = \frac{\theta}{\|x_c, y_c\|^3} \end{aligned} \tag{2.17}$$

This approximation is used to splat the Gaussians, integrating them using a linear projection model. It will underestimate the scaling factor of the mapping, resulting in larger footprints. The precision of this approximation will decrease when moving further away from the center of the Gaussian.

Undistortion

To compute the world space rays used in the ray tracing setup from image rays we have to apply the forward mapping from the image pixel coordinates $\mathbf{u}$ as $K_{2\times 2}^{-1}(\mathbf{u} - \mathbf{c})$ followed by undistortion, the computation of x_c according to the camera model, and finally the camera rotation R^T. The undistortion procedure requires solving the set of distortion equations per pixel, which can in general only be done numerically, that is, by applying the Newton-Raphson iterative method for equations.

In the context of fisheye cameras, the process of transforming the images to the pinhole camera model is called *undistortion*. Since we choose to distinguish between distortion and fisheye camera model definition, to avoid confusion, we will use the term *unfisheye* to describe this process. It corresponds to computing the camera rays of the pinhole camera, then projecting them using the fisheye camera model, and sampling the fisheye image at these points using bilinear interpolation. Note that this mapping will loose some information at large θ 2.2.

Figure 2.2: Fisheye (left) vs unfisheyed image [44]

An inverse procedure can also be used to produce fisheye images from pinhole images (omitting distortion):

$$\begin{pmatrix} u_f \\ v_f \end{pmatrix} = K_{2\times2}^{-1}(\mathbf{p_f} - \mathbf{c})$$

$$\begin{pmatrix} u_p \\ v_p \end{pmatrix} = \begin{pmatrix} \frac{u_f}{\theta}\tan\theta \\ \frac{v_f}{\theta}\tan\theta \end{pmatrix} \qquad \theta = \|u_f, v_f\|$$

$$\bar{\mathbf{p}}_\mathbf{p} = K_{2\times2}\begin{pmatrix} u_p \\ v_p \end{pmatrix} + \mathbf{c}$$

However, to fully represent the fisheye image, the pinhole camera must have a wider angle.

2.9 Geometric 3D reconstruction

Radiance field methods usually focus on reconstructions based on a known set of camera parameters. Point based methods often times further rely on point clouds for initialization. To solve the general problem of reconstruction from only a set of images and known camera intrinsics, these additional data can be obtained using a set of methods that rely on key point detection and geometric constraints.

Structure-from-motion (SfM) [51] is a method to produce a sparse point cloud along with camera extrinsics, given a set of images of the same scene taken by a set of nearby cameras or a moving camera. As it jointly optimizes the camera trajectory and the 3D points, it is also known in robotics as simultaneous mapping and localization (SLAM)[13][2]. One fundamental drawback of this pipeline is its heavy reliance on feature detectors and local descriptors to generate good tentative correspondences. Consequently, the number of points generated in flat regions with fewer discriminative features will be much lower.

The subsequent work introduced Multi-view stereo (MVS)[18][6], a set of methods to produce a dense point cloud by using more efficient patch matching, depth estimation, generating more correspondences utilizing disparity maps or epipolar constraints, and other techniques ([18],[6]), including learning-based methods.

These methods represent their own adjacent area of research, and even the basics would go well beyond the scope of this work. Luckily, there are advanced open source implementations, the most widely used in this area being COLMAP software ([46], [47]).

2.10 Hardware accelerated ray tracing

The main factor behind the wide adoption of ray tracing based methods is hardware advancement. Direct lighting simulation requires calculating per-ray intersections, and, while each intersection is costly, the situation is further complicated by the large number of primitives. To avoid expensive scene queries, ray tracers use spatial acceleration structures for logarithmic search, cache locality, and packet-based traversal. The typical approaches are spatial subdivisions, bounding volume hierarchies (BVH), and space-local linear codes. Since the task is massively parallel, classic implementations have relied on general purpose GPU infrastructure. Modern day ray tracing on the other hand is largely enabled by NVIDIA RTX framework, which abstracts the acceleration structure and enables fast triangle intersections, implemented in hardware. The framework is accessible through extensions to major APIs. The simplest choice for general applications including ours is CUDA based OptiX [41]. The programmable parts of the traversal logic are specified using kernels called programs. These are launched in per-ray threads scheduled by the framework without any guaranties, thereby disabling use of shared memory and synchronization primitives beyond atomics.

OptiX programs are divided into three groups, each containing one or more programs. Ray generation group programs initialize the ray and define which programs will be called next; miss group programs are invoked in case if no intersection was reported; finally, hit group programs are of three types:

- *Intersection* program is used for custom primitives. When an axis aligned bounding box (AABB) of a primitive is intersected, the program can run finer intersection handling.
- *Closest-hit* program runs at the end of the traversal, receiving the closest hit that was not ignored by the any-hit program.
- *Any-hit* program is invoked for each candidate primitive intersection. It can ignore the intersection, or terminate the traversal, switching to the closest-hit program. Note that the order of intersections is not guaranteed.

There are also less common exception and callable groups, which pose convenience rather than necessity. Since the programs run on the same thread, they can share data through registers, called payload, limited to 128 bytes. The rest of the shared data should reside in global, constant, or texture memory.

OptiX offers two types of acceleration structures: geometry level and instance level. These can be organized in a directed acyclic graph, where

the sink nodes are geometry level structures. This allows for flexible scene instancing, including rendering an arbitrary number of scene copies without memory overhead.

Chapter 3

Related work

This chapter will discuss the modern methods for learning Gaussian scene representations, including rasterization and ray tracing approaches, in relation to optimization methods, camera model, and volume rendering approximations.

3.1 Gaussian Splatting

Addressing the drawbacks of earlier main stream neural radiance field and occupancy grid methods, the work by Kerbl et.al. [10] introduced an efficient rendering and image-supervised optimization algorithm for differentiable point-based Gaussian kernels (3DGS), building on the Zwicker's EWA splatting framework.

3.1.1 Rasterization

The proposed rasterization algorithm for 3D elliptical primitives is divided into the pre-processing, sorting, and rendering phases summarized as follows:

(i) *Pre-processing phase* runs the following operations in a per-point kernel:

(1) Filter the points that belong to the camera frustum and project them. Store the projected points and depths;

(2) Compute the corresponding 2D covariance matrices following the EWA splatting algorithm.

(3) For an image grid of 16x16 tiles, compute the subgrid of tiles intersected by the circular neighborhood around the projected point. The radius of this neighborhood is defined by the largest eigenvalue of the 2D Gaussian scaled by a constant factor $r = 3\sqrt{\lambda_{max}}$ to include the cutoff region. Store the radius r, and the number of overlapped tiles;

(4) In addition, compute and store SH colors.

(ii) *Sorting phase* sorts the particle lists per tile as:

(1) Use parallel prefix sum over the number of overlapped tiles to allocate a buffer of overlapped tile indices per particle;

(2) Run the overlapping again, reusing the radii. Store 64-bit indices to the buffer, with upper 32 bits assigned the tile index, and the lower bits assigned the depth computed in pre-processing;

(3) Use parallel radix sort on the indices, obtaining the sorted lists per tile.

(4) Run a kernel to identify the corresponding offsets of tiles.

(iii) *Rendering phase* runs per-tile thread blocks (per pixel threads):

(1) Collect the tile particles into shared memory;

(2) Traverse these depth sorted particles front-to-back to evaluate the transfer equation. Store the resulting color and transmittance.

This algorithm, originally implemented in CUDA kernels, is extremely fast. Large particles could degrade memory requirement and performance due to per-tile duplication, though they should be avoided anyway due to the EWA approximation. Note that the depth sorting is only per-tile, rather than per-pixel, which may lead to visibility artifacts and introduce noise to optimization.

3.1.2 Rendering formulation

The output color is computed by evaluating the RTE with substitution 2.10, resulting in [1]

$$C(\mathbf{u}) = \sum_{i}^{N} c_i \alpha_i(\mathbf{u}) \prod_{j=1}^{i-1} (1 - \alpha_i(\mathbf{u})) \tag{3.1}$$

The term α_i here will be called *response*, defined in 2D as

$$\alpha_i(\mathbf{u}) = \sigma_i G_{\hat{\Sigma}}(\mathbf{u}) \tag{3.2}$$

introducing the *opacity* term σ_i and the de-normalized Gaussian

$$G_{\hat{\Sigma}(\mathbf{u})} = e^{-\frac{1}{2}(\mathbf{u}-\hat{\mu_\mathbf{i}})^T \hat{\Sigma}_i^{-1}(\mathbf{u}-\hat{\mu_\mathbf{i}})}$$

Note that this differs from the original EWA reconstruction filter 2.6 by the scaling factor. The key observation here is that the ray space mapping Jacobian $\mathbf{J}$ depends on depth, therefore, changing camera distance would change the transparency, which the authors have chosen to avoid. The normalization factors are therefore all approximated by the learned opacity term.

Also note that the 2D means $\hat{\mu}_\mathbf{i}$ are obtained simply by applying the camera projection model

$$\hat{\mu_\mathbf{i}} = K\mathbf{m}(\bar{\mu_\mathbf{i}}) \qquad \bar{\mu_\mathbf{i}} = \mathbf{W}\mu_\mathbf{i} + \mathbf{t} \tag{3.3}$$

where m is the ray space mapping for the camera model. In other words, they are not subject to any approximation. However, their gradients will be influenced by projection approximation, as will be shown later.

[1] The equation here is rewritten in front-to-back order, which is equivalent.

Additionally, we may add a background term for the background color c_{bg} to the expression as

$$C'(\mathbf{u}) = C(\mathbf{u}) + T_N(\mathbf{u})c_{bg}$$

denoting

$$T_i(\mathbf{u}) = \prod_{j=1}^{i-1}(1 - \alpha_i(\mathbf{u}))$$

Typically, the background is considered black, unless the training is performed on images masked with an alpha channel, in which case it is a part of the dataset. Varying background during training is a form of regularization that does not work in general; therefore, the background term is usually omitted.

Since the formulation 3.1 is differentiable, the gradients $\frac{\partial C}{\partial \text{parameter}}$ for each parameter $[c_i,\sigma_i,\mu_\mathbf{i},\Sigma_i]$ can be computed using the chain rule. The color of the particle can be represented via SH coefficients as in equation 2.11 $c_i = c(\mathbf{d})$ to produce the view dependent color. The 2D covariance matrix is calculated using 2.16 and 2.7 as

$$\hat{\Sigma}_i = \mathbf{J}_{\bar{\mu}_i}\mathbf{W}\Sigma_i\mathbf{W}^T\mathbf{J}_{\bar{\mu}_i}^T \tag{3.4}$$

Since the primitives are volumetric, the 3D covariance matrix Σ_i is required to be positive definite. During optimization, this property can be enforced by instead optimizing the component matrices of the corresponding Cholesky decomposition. Since the covariance effectively represents rotation and scale of the Gaussian ellipsoid, it can be decomposed as:

$$\Sigma_i = R_i S_i S_i^T R_i^T \tag{3.5}$$

The scale matrix is only defined by the axis scales $\mathbf{s_i}$.

Rotation

The parameterization chosen for the rotation matrix is a vector $\mathbf{q_i} = (q_r, q_i, q_j, q_k)^T$ corresponding to a quaternion.

$$R(\mathbf{q}) = 2\begin{pmatrix} \frac{1}{2} - \left(q_j^2 + q_k^2\right) & (q_i q_j - q_r q_k) & (q_i q_k + q_r q_j) \\ (q_i q_j + q_r q_k) & \frac{1}{2} - \left(q_i^2 + q_k^2\right) & (q_j q_k - q_r q_i) \\ (q_i q_k - q_r q_j) & (q_j q_k + q_r q_i) & \frac{1}{2} - \left(q_i^2 + q_j^2\right) \end{pmatrix}$$

This approach is commonly used in computer graphics, as it allows one to continuously interpolate rotations at any angles, while avoiding the collapsing degrees of freedom, which happens in Euler parameterization. This global continuity is also desirable in the learning setup. The slight downside of the representation is its redundancy, manifested as $\mathbf{q_i}$ and $-\mathbf{q_i}$ representing the same rotation. To represent the actual rotations the quaternion vectors also have to be normalized, introducing some further complexity to implementations.

An alternative could be Rodrigues parameterization, which is compact, does not require normalization, and is more stable for optimization. However,

its key disadvantage is that the parameterization angle φ is constrained to $[0, \pi)$, which would introduce a discontinuity in training. Despite this, the applicability of this rotation parameterization might be worth testing.

Color

To capture a view-dependent appearance, the method defines the particle colors as functions of the view direction $\mathbf{c_i}(\mathbf{d})$ parameterized by the SH series of order $l_{max} = 3$ for each color component 2.11. This corresponds to 16×3 color coefficients β_l^m for each particle. As the number of parameters increases quadratically with l_{max} and the number of parameter sets per scene is large, the maximum degree must be chosen small. This representation, therefore, despite being general, produces low angular resolution in practice. Fortunately, there is an alternative representation that addresses this problem. The approach is analogous to EWA reconstruction filters, but over the unit sphere. As demonstrated by recent works (MEGS2 [11], Spec-Gaussian [58]), it can reduce memory requirements while achieving results similar to the SH representation, and makes computing sharp view dependent effects feasible. However, it comes with computational cost and may struggle more with low frequencies, so for diffuse scattering and reflections, SH might still be a better option.

3.1.3 Anti-aliasing

In subsequent modifications to the implementation, the 3DGS reconstruction filters mathematically match the EWA formulation, given by

$$G(\hat{\mathbf{x}}) = \sqrt{\frac{|\hat{\Sigma}_i|}{|\hat{\Sigma}_i + s\mathbf{I}|}} e^{-\frac{1}{2}(\hat{\mathbf{x}}-\hat{\mu}_\mathbf{i})^T(\hat{\Sigma}_i^{-1}+s\mathbf{I})(\hat{\mathbf{x}}-\hat{\mu}_\mathbf{i})}$$

for some scale hyperparameter s. The purpose of this filter is to effectively pad the 2D footprint of the Gaussian enough, so the maximum projected frequency is bounded by half the image sampling frequency. To meet this requirement without over-blurring the images, in the downstream MipSplatting[61] paper the authors proposed to set s adaptively, choosing the pixel box filter approximated by the pixel-sized isotropic Gaussian, as physically and empirically justified. Note that this filter is chosen to aid, rather than ensure anti-aliasing. They also observed that the scale of 3D ellipsoids is pushed lower by optimization due to the same screen-space filter applied during training, which they call *shrinkage bias*. This sometimes leads to degenerate cases where the rendered footprint of a Gaussian is fully determined by the low-pass filter, so its proportion in the rendered image will not be preserved in closer or zoomed-in views, despite higher effective sampling frequency. To ensure view-independent faithful reconstruction and avoid the degeneracies, the authors propose applying a second analogous low-pass filter in 3D. Since the spatial frequency of the representation should be limited by the resolution of training images, they choose the minimal 3D filter scale for each Gaussian

defined by the highest frequency of the training image that sees it, ensuring faithful rendering for at least one training image. The obtained 3D filters can then be added directly to the corresponding Gaussian covariances for compact representation. This filtering modification ensures adaptive anti-aliasing while minimizing the erosion artifacts. It produces very good results, outperforming anti-aliased NeRF variants.

Analytic approach

An alternative approach to anti-aliasing has been proposed more recently in AnalyticSplatting[31] addresses the fundamental cause of aliasing, namely that the image sampling frequency is limited by the pixel size as $1/p_x$, since traditional rendering pipelines take one sample per pixel. Physically, the image corresponds to the radiant flux Φ through the camera sensor, obtained by integrating the radiance function $I : \mathbb{R}^2 \to \mathbb{R}$ in the sensor area $\sum_{W\times H} \int_B I(\hat{\mathbf{x}}) d\hat{\mathbf{x}}$ with the pixel $B = [0, p_x] \times [0, p_y]$. Simulating this image formation process by taking one sample per pixel then corresponds to a crude approximation of the integral with only one sample $\int_B I(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \approx I(\hat{\mathbf{x}}_\mathbf{i})$; the correlation of these errors over multiple pixels then leads to aliasing artifacts. The prior MipSplatting approach acknowledged this, noting that this single sample should be multiplied by a pixel area $I(\hat{\mathbf{x}}_\mathbf{i})|B|$ to correspond to a single sample midpoint quadrature. A more precise estimate could be reached by taking more samples, although this would defeat the purpose. Instead, the authors of the analytic approach go further, designing an analytic approximation of the pixel integral for Gaussian splats. The 1D integral of the projected Gaussian can be computed using the CDF $G(x) = \int_{-\infty}^{x} g(x)dx$ as $\hat{\Phi}_{B_x} = G(u + 1/2) - G(u - 1/2)$. To make the 2D Gaussian decomposable along the pixel axes, the authors assume that the axes align with the eigen vectors of the covariance matrix, effectively rotating the pixel. Since the CDF does not have a closed form, they introduce another approximation $G(x) \approx S(x/\sigma) = S_\sigma(x)$ using a sigmoid-like function scaled by the standard deviation $\sigma = \frac{1}{\sqrt{\lambda}}$ along the corresponding axis. This results in the following modification to the RTE:

$$\alpha_i(\mathbf{u}) = \sigma_i \hat{\Phi}_B(\mathbf{u})$$

$$\hat{\Phi}_B(\mathbf{u}) = 2\pi\sigma_x\sigma_y \left[S_{\sigma_x}(u + \frac{1}{2}) - S_{\sigma_x}(u - \frac{1}{2})\right] \left[S_{\sigma_y}(v + \frac{1}{2}) - S_{\sigma_y}(v - \frac{1}{2})\right]$$

where $\alpha_x\alpha_y$ are standard deviations and α_i is the opacity term.

This approach comes with a slight decrease in performance due to algebraic operations, but it reduces the over-smoothing that is sometimes noticeable in MipSplatting, giving a modest improvement in metrics.

3.1.4 Density control

Since the number of particles in the final reconstruction would not make a good hyperparameter, it must be a part of the optimization process. In the

original 3DGS authors advertise its ability to produce good results using only a sparse point cloud for initialization, obtained "for free" from the COLMAP's SfM [46] along with the set of training views. The initial scales of the 3D Gaussians are then chosen as distances to the nearest neighbor, and the initial SH coefficients match the point color for initial training stability. Their ablation study also notes that in simple cases like the NeRF-syntetic dataset, random initialization is sufficient, while for large unbounded scenes it results in floating artifacts.

Since changing the number of parameters is a large discreet optimization problem, it requires some kind of heuristic approach. The second key contribution of 3DGS that addresses this is the adaptive density control algorithm (ADC).

Adaptive density control

The algorithm can be seen as increasing the capacity of the model by adding new Gaussian particles in directions where the objective with respect to the positions (means) of particles is less smooth, where the optimizer struggles the most. The authors [10] identify two types of these regions, namely over-reconstructed and under-reconstructed 3.1. Adding more parameters to these regions increases the expressivity of the function in this region, giving the optimizer more opportunities to converge. However, this can lead to overfitting to some training views, which often manifests itself as stuck floating artifacts closer to the camera. These artifacts can be far from the actual geometry and are characterized by saturated opacity activations, making them hard to dissolve. The proposed regularization technique is to periodically reset the opacity values globally, which allows the optimizer to reach the occluded geometry and find a better minimum. The optimization will often create very transparent particles, trying to remove unnecessary geometry. The strategy therefore includes a pruning step to remove these particles based on some threshold.

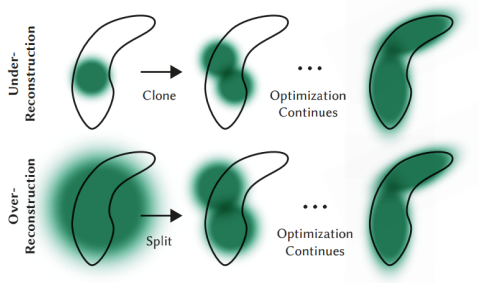


Figure 3.1: Cloning and splitting (3DGS [10])

More specifically, this optimization procedure consists of steps, each run periodically after a certain number of "warm-up" iterations with some fixed interval during the gradient optimization.

- *Densify.* Select particles with the positional gradient based *densification criterion* above a threshold. If the particle scale is larger than a certain threshold, it is *split* into two. The new particle is created with the position sampled from the normal distribution around the original particle with corresponding variance, and both particles are scaled by some constant. The smaller particles are instead *cloned* by copying the original particle and letting it be moved by the optimizer in the next step.
- *Prune* by removing particles with opacity below a certain threshold. 3DGS also suggests pruning particles with a large projected footprint, as they likely correspond to artifacts and may cause unpredictable growth due to splitting. As discussed above, larger particles also increase the error in the rendering approximation.
- *Reset the opacity* for all particles to some small constant close to the pruning threshold.

In particular, this algorithm depends on a fair number of hyper-parameters, namely step frequencies, scaling factor and degree of splitting, gradient and opacity thresholds, and reset threshold. The densification criterion in 3DGS was chosen as the average of screen space position gradient norms accumulated before or between steps. This heuristic and the overall algorithm are not well justified. It leads to unpredictable final scene size and is hard to tune in new setups.

Improvements and alternatives

Subsequent variations, e.g. the work by S.Rota et.al. [43], tune this heuristic to avoid the large jumps due to opacity resets and limit the final scene size. The key point in this and similar works is the direct usage of image metric, rather than its derivative, to identify the densification regions. This is motivated by the observation that moving large Gaussians slightly does not result in them better representing the high frequency geometry; therefore, their position gradients are not good predictors for densification.

An alternative more principled approach to densification is a strategy called MCMC[26] for Markov Chain Monte-Carlo, based on the main idea of the original ADC, which is that the densification is governed by the update rule of the optimizer. The approach treats this update rule as transitioning the state of the scene to one that with higher probability describes the image. The opacities of the Gaussians are treated as confidence in geometry at the corresponding region. Based on the latter, the authors propose to use a modified version of gradient optimization that adds noise to positions depending on their opacity and covariance to encourage exploration of nearby space for particles with less confidence. Particles with higher confidence, on the other hand, will be governed mainly by the default gradient optimization. The approach to changing the number of particles is grounded in the idea that it *should not* by itself impact the probability of the scene being correctly represented, so this probability is only controlled by the modified optimization.

The new particles are therefore created at places with the highest confidence in a way to minimally impact the rendered image. The procedure is then similar to cloning and pruning, except the pruning is replaced by rescaling relocating them to means of some sample of opaque particles to explore more likely regions. Additionally, smaller scene sizes are encouraged by L1 regularization, preventing saturation of opacities and scales.

This approach is easier to control than ADC, it demonstrates high degree of invariance to initialization, and should avoid light floaters and cluttered regions with large Gaussians, where ADC gets stuck.

3.2 Gradient optimization

Since the rendering formulation of 3DGS[25] 3.1 is differentiable, the analytic gradients can be derived with respect to the image loss. The optimization is then performed per image as in 2.12, allowing the use of structured similarity. The common loss choice in these pipelines is a mix of MSE and SSIM loss with coefficient $\lambda = 0.8$:

$$L = (1 - \lambda)L_1 + \lambda L_{SSIM}$$

The gradient of this differentiable loss with respect to the estimated image $\frac{\partial L}{\partial C}$ is typically evaluated using backpropagation, and the same usually applies to the parameter activation functions, which will be discussed in section 3.2. However, the rendering equation depends on the rendering procedure, since it requires first evaluating sequences of Gaussians that contribute to each pixel color. It will require a similar procedure for differentiation as we will derive here.

Gradients of the rendering equation

We first introduce the leading transmittance term

$$T_i = \prod_{j=1}^{i-1}(1 - \alpha_i)$$

So, the derivative with respect to the color of each contributing particle is

$$\frac{\partial C}{\partial c_i} = \alpha_i T_i$$

The response terms α_i on the other hand, also influence the transmittance of all the particles that follow. We therefore separate the equation into leading and trailing parts as follows:

$$C = C^f_{i-1} + c_i \alpha_i T_i + C^b_{i+1}$$

$$C^f_{i-1} = \sum_{j=1}^{i-1} c_j \alpha_j T_j \qquad C^b_{i+1} = \sum_{j=i+1}^{N} c_j \alpha_j T_j$$

The leading term C^f_{i-1} will not be of consequence here, as it does not depend on α_i. To get the parameter out of the trailing term, we first decompose the transmittance analogously:

$$T_j = T_i(1-\alpha_i)\left[\prod_{k=i+1}^{j-1}(1-\alpha_k)\right]$$

Applying this, we get:

$$C^b_{i+1} = K - \alpha_i T_i \sum_{j=i+1}^{N} c_j \alpha_j \left[\prod_{k=i+1}^{j-1}(1-\alpha_k)\right] = K - \alpha_i T_i \frac{C^b_{i+1}}{T_{i+1}}$$

where K is constant with respect to α_i. This finally allows us to express the response derivative as follows:

$$\frac{\partial C}{\partial \alpha_i} = T_i\left(c_i - \frac{C^b_{i+1}}{T_{i+1}}\right) = T_i c_i - \frac{C^b_{i+1}}{1-\alpha_i} \tag{3.6}$$

To evaluate this equation, we have to know the leading transmittance and trailing color at each point. Storing these values per pixel and per particle is infeasible. Instead, they can be recomputed using another pass of the rendering algorithm, which we call *backward*. The backward pass is run after the forward pass and loss computation, receiving some additional input from these operations. In Gaussian splatting methods it reuses the sorted per-tile lists, then runs the analogous per-pixel procedure. The tile list traversal can be performed either:

Front-to-back by reusing the color from forward pass, since $C^b_{i+1} = C - C_i$. The loss gradient in this case would correspond to

$$\frac{\partial L}{\partial \alpha_i} = \frac{\partial L}{\partial C}\frac{\partial C}{\partial \alpha_i} = T_i\langle \frac{\partial L}{\partial C}, \mathbf{c_i}\rangle - \frac{1}{1-\alpha_i}\left(\langle \frac{\partial L}{\partial C}, C\rangle - \langle \frac{\partial L}{\partial C}, C_i\rangle\right) \tag{3.7}$$

Therefore, the additional input of the backward pass is scalar $\langle \frac{\partial L}{\partial C}, C\rangle$.

OR **Back-to-front**, so the C^b_{i+1} is the accumulated color. In this case, the leading transmittance T_i can be obtained by repeatedly dividing the output transmittance T of the forward pass $T_i = T\prod_{j=1}^{i}\frac{1}{(1-\alpha_j)}$.

Gaussian splatting chooses the back-to-front option as more natural, although it leads to a numeric error in T_i, resulting in divergence from the forward pass values.

The optional background term can be added as:

$$\frac{\partial C'}{\partial \alpha_i} = \frac{\partial C}{\partial \alpha_i} + c_{bg}\frac{T_N}{1-\alpha_i}$$

This would increase the norms of screen space and positional gradients in transparent regions, which could lead to more aggressive densification and larger steps in the early training stages. It therefore additionally motivates the ADC approach to start densification only after some number of initial steps.

Numerical stability

The numerical stability is influenced by:

- $\alpha_i \approx 0$. Increasing the number of operations leads to increasing the cumulative error with only a tiny contribution to the output. 3DGS has therefore chosen to reject the particles with response $\alpha_i < \frac{1}{255}$. This choice is arbitrary but inevitable, since the infinite response of the Gaussian has to be clamped at some point.
- $\alpha_i \approx 1$ leads to multiplying and dividing transmittance by small values, which may lead to large relative errors in T. The value of α_i is therefore clamped to 0.99, also as an arbitrary choice.
- Small values of T result in negligible contributions, so, similarly to the first case, these contributions are ignored. The chosen lower bound on T is 10^{-4}.

Covariance and position gradients

In splatting, the response term α_i 3.2 is defined in the screen space. The positional and covariance gradients will therefore be derived in screen space and then propagated to 3D to be used for updates. The screen space positional gradients will also be used in the standard ADC densification procedure. The covariance gradient path using 3.4 and 3.5 will lead to scale and rotation gradients, and will also contribute to the 3D position gradients due to the ray space linearization.

Positional gradients will be computed as follows:

$$\frac{\partial \alpha_i}{\partial \mu_\mathbf{i}} = \frac{\partial \alpha_i}{\partial \hat{\mu}_\mathbf{i}} \cdot \frac{\partial \hat{\mu}_\mathbf{i}}{\partial \mu_\mathbf{i}} + \frac{\partial \alpha_i}{\partial \hat{\Sigma}_i} \cdot \frac{\partial \hat{\Sigma}_i}{\partial \mu_\mathbf{i}}$$

The screen space gradients are:

$$\frac{\partial \alpha_i}{\partial \hat{\mu}_\mathbf{i}} = \alpha_i \hat{\Sigma}_i^{-1}(\mathbf{u} - \hat{\mu}_\mathbf{i})$$

On this path, we compute the derivative of 3.3, resulting in

$$\frac{\partial \hat{\mu}_\mathbf{i}}{\partial \mu_\mathbf{i}} = \mathbf{K}\mathbf{J}_{\bar{\mu}_i}\mathbf{W}$$

The gradient chain rule can be generalized for matrix derivatives by defining the dot product for matrices, called the Frobenius product

$$\langle A, B \rangle \stackrel{!}{=} tr(A^T B) = \mathrm{vec}(A)^T \mathrm{vec}(B)$$

$$\frac{\partial \alpha}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial x} = \langle \left(\frac{\partial \alpha}{\partial \mathbf{x}} \right)^T, \frac{\partial \mathbf{x}}{\partial x} \rangle$$

$$\frac{\partial f}{\partial x} = \sum_i^M \sum_j^N \frac{\partial f}{\partial X_{ij}} \frac{\partial X_{ij}}{\partial x} = \langle \frac{\partial f}{\partial X}, \frac{\partial X}{\partial x} \rangle$$

The following identities can then be used to simplify derivatives

$$\langle A, BC\rangle = \langle B^T A, C\rangle = \langle AC^T, B\rangle \qquad \frac{\partial AB}{\partial x} = \frac{\partial A}{\partial x}B + A\frac{\partial B}{\partial x}$$

Using this, the covariance derivative of the sub-expression $s_i = -\frac{1}{2}\hat{\mathbf{p}}_\mathbf{i}^\mathbf{T}\hat{\Sigma}_i^{-1}\hat{\mathbf{p}}_\mathbf{i}$, further applying $\Sigma = \Sigma^T$, is obtained as:

$$\frac{\partial s_i}{\partial \hat{\Sigma}_i} = \frac{1}{2}\hat{\Sigma}_i^{-1}\hat{\mathbf{p}}_\mathbf{i}\hat{\mathbf{p}}_\mathbf{i}^T\hat{\Sigma}_i^{-1}$$

This gradient can be decomposed, obtaining $\frac{\partial s_i}{\partial \mathbf{J}_{\bar{\mu}_i}}$ and $\frac{\partial s_i}{\partial \Sigma_i}$. The first component then leads to the second part of the positional gradients, proceeding with the set of matrices $\frac{\partial \mathbf{J}_{\bar{\mu}_i}}{\partial \bar{\mu}_\mathbf{i}}$. The expressions for these gradients are quite complicated in the fisheye camera case but are derived in FisheyeGS[33]. The second component is used to derive the scale and rotation gradients by decomposing the covariance as $\Sigma = MM^T$, where $M = RS$. For details on this part, please refer to the mathematical supplement [59] of **gsplat** library. Note that decomposing $\hat{\Sigma}_i^{-1}$ right away would only complicate things in this case because $\mathbf{J}_{\bar{\mu}_i}$ is not invertible.

Note that here we did not account for direction dependent colors $c_i(\mathbf{d})$. Since the view direction is given by $\mathbf{d} = (\mu_i - \mathbf{C})/\|\cdot\|$, they would introduce another component to the position gradient.

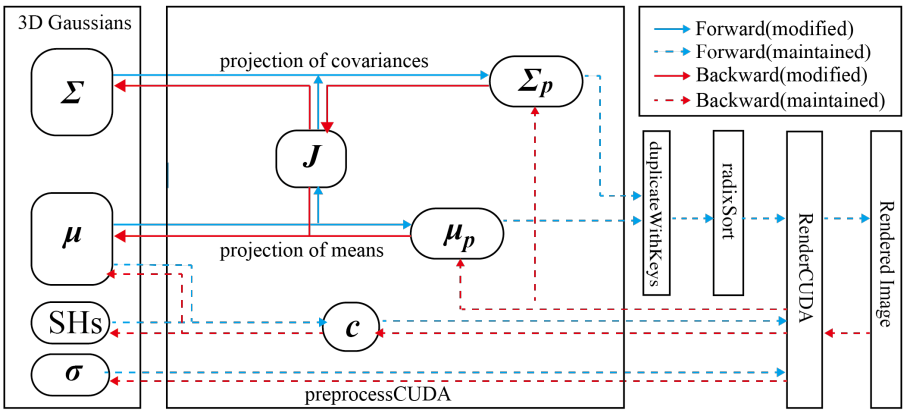


Figure 3.2: 3DGS differentiable rendering pipeline figure from FisheyeGS [33]. The modified parts refer to the ones influenced by changing the camera Jacobian.

Activations

The aforementioned rendering and derivation procedures define the differentiable rendering module, summarized in the diagram 3.2. This module can now be wrapped and used as part of a machine learning pipeline, utilizing a framework like PyTorch [42]. Notice that some of the inputs do not represent the leaf parameters, as they have to be constrained in some ways. This can

be achieved on the framework side, defining the module input in terms of automatically differentiated *activation* functions:

- $\mathbf{\Sigma}$ is constrained to $\mathbf{M}^T\mathbf{M}$ for some regular $\mathbf{M}$ that further decomposes into rotation and scale. Computing it on the framework side, however, would result in storing per-particle 3×3 matrices along with their gradients, which would increase the memory requirement. Implementations therefore usually prefer to accumulate the per-pixel gradients with respect to the compact representations at the cost of additional kernel arithmetic.
- For covariance to remain positive definite, scales $\mathbf{s}_i$ have to be non-zero. From the activation function, we would expect to avoid redundant negative values and a smooth asymptotic behavior around zero. To fulfill these properties, 3DGS chooses the exponent function. This choice is not ideal, as scales would tend to grow fast. An alternative would be the smoother *softplus* function $\log(1 + e^x)$ with derivative corresponding to the sigmoid function. Constraining the scales by the activation function instead of pruning would be possible but would lead to a different problem, namely saturation.
- The optical value σ_i physically represents the fraction of absorbed light, which should fall in the $[0, 1]$ range. The activation function should therefore be either the plain sigmoid (3DGS) or a sigmoid-like alternative.
- The rotation quaternions have to be normalized to represent unit sphere rotations. The activation is therefore $\frac{\mathbf{q_i}}{\|\mathbf{q_i}\|}$.
- SH output has to be positive to represent emission. For parameters $\beta_l^m \in [-1, 1]$ with $l_{max} = 3$ the function will be within the range of $[-0.5, 0.5]$. Shifting this function to 0.5 would then roughly correspond to the color values. Restricting the coefficients is avoided by instead relying on the optimization to push the parameters into the required range. The physical plausibility is then ensured by applying the ReLU activation $\max(0, x)$ to the output. Again, to avoid the redundant storage of color gradients, the SH along with the activations are evaluated in the pre-processing step of the rendering pipeline.

Optimization

Unlike classical machine learning pipelines, gradient optimization in this setup is subject to noise beyond the limited batch size and additional discontinuities beyond zero-measure introduced by:

- The ADC steps, removing and adding new parameters. The standard 3DGS version introduces large, though infrequent, jumps due to pruning after opacity resets, as well as smaller and more frequent updates.
- Particles abruptly entering and leaving the screen space due to position-based culling.

- Visibility errors due to position-based per-tile sorting.

This motivates the use of gradient filtering as a necessary part of optimization. The different parameter types would cause the gradient scales and optimization stability to vary greatly between parameter groups; this motivates separate learning rates for each group. Additionally, the scale invariance of learning rates has to be enforced since:

- Scene scale is determined externally by camera positions.
- All parameter gradients are scaled by opacity, leading to smaller steps for less opaque particles.
- Gradient norms will be smaller for views where the particles are only partially visible.

Most Gaussian scene training approaches therefore rely on Adam optimizer with separate learning rate for each type of the parameter. They additionally use a continuous exponential learning rate schedule for positions to ensure stable convergence for geometry.

Although this training approach is ubiquitous, any part of 3DGS has alternatives. In particular, the authors of 3DGS-LM [22] have reformulated the loss in terms of non-linear least squares to optimize it via the Levenberg-Marquardt algorithm. Unlike the first order methods it is well conditioned and leads to faster convergence. However, since the method is not stochastic, the steps require evaluation on the entire dataset, the authors have found that it still works for large batches.

3.3 Ray based methods

Since 3DGS, splatting methods have been demonstrated to produce fast, interpretable, and high quality reconstructions. However, the rasterization pipeline comes with fundamental limitations regarding precise visibility and simulation of incoherent light effects that emerge in various physical settings. These include both spatial and temporal distortion of light rays reaching the camera sensor, as well as the lighting effects produced by reflections and scattering.

This section will discuss several radiance field methods that target these limitations using hardware accelerated ray tracing for both training and inference, providing an alternative to the splatting rasterization pipeline. These methods, while inheriting the same approach to world space particles, fully abandon the screen-space EWA formulation, as the projected footprints are no longer Gaussian. The per-particle volume rendering formulation is also preserved in some approaches, focusing on performance, while others aim for precise volume rendering, treating the particle kernels as a general density field parameterization.

Note that the tracing approach does not require the 3D kernels to be Gaussian. Different kinds of filter can be used for better surface reconstruction

(flat Gaussians), speed (generalized Gaussians), expressivity (cosine wave modulated Gaussians) [37], or naturally limited support (Epanechnikov kernels [12]). These will not be covered in this work, since we focus on ray tracing 3D Gaussians.

3.3.1 Ray marching

These approaches aim for precise volume integration of the Gaussian density field, eliminating the simplifying assumptions made in Gaussian splatting, namely the visibility approximation, self-occlusion, constant color, and non-overlapping assumption. This leads to precise volumetric rendering of 3D Gaussians, similar to the neural radiance field approach.

Ray marching approach presented in RayGauss[5] computes the RTE 3.1 by taking evenly spaced samples along the ray. At each ray segment, it computes the intersection with Gaussian confidence ellipsoids and accumulates the corresponding transmittance and color contributions. The distinctive quality of this approach is that it combines SH and spherical Gaussians for view dependent colors for faithful diffuse and specular color reconstruction, respectively. Additionally, the colors are treated as per-point, weighted by all Gaussian responses contributing to this point, which leads to a more precise physical representation of emission.

$$c(\mathbf{x}, \mathbf{d}) = \frac{\sum_i c_i(\mathbf{d})\alpha_i(\mathbf{x})}{\sum_i \alpha_i(\mathbf{x})}$$

The significant drawback of this approach, however, is its high cost induced by the per segment ray queries, which are fundamental to ray marching.

The advantage of ray marching, on the other hand, is a noticeable increase in quality enabled by precise volume integration and overlap handling. As demonstrated in the work by A.Celarek et al.[10], it allows for a more precise visibility testing than per-pixel sorting, applied in e.g. 3DGRT [37].

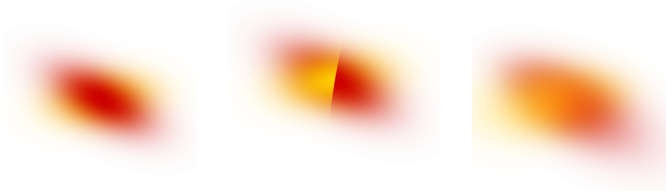


Table 3.1: Position-based depth sorting, pixel-based, and ray marching. [10]

3.3.2 Analytic integration

The more recent work by J.Condor et al.[12] also focused on precise volume rendering of 3D Gaussians, using the existing differentiable ray tracers, rather than custom ray marching. The authors approached the problem via analytic integration of the Gaussian kernels. Given entry and exit parameters t_0 and t_1 of the ray parameterized as $r(t) = \mathbf{x_0} + \mathbf{d}t$, they switch to the symmetric

integration domain defined by $t_s = \frac{t_1+t_2}{2}$, and show that the integral of a normalized 3D Gaussian $g(\mathbf{x})$ along the ray takes the form

$$\int_{-t_s}^{t_s} g(\mathbf{x_t}) = f(\Sigma, \mathbf{d}, \mathbf{x_0})\text{erf}(t_s\sqrt{\frac{1}{2}})$$

where erf() is the standard error function, provided by mathematical libraries. This expression can also be applied for denormalized Gaussians, and is differentiable, so it can potentially be applied in combination with the fast ray tracing.

3.3.3 Fast ray tracing

The main approach discussed here was introduced by Moenne-Loccoz et al.[37](3DGRT), providing a fast approximate volume ray tracing approach and focusing on a direct comparison to 3DGS[10]. The method was shown to perform on par with 3DGS, while featuring a direct support for incoherent rays and mesh-based secondary lighting effects.

Rendering formulation

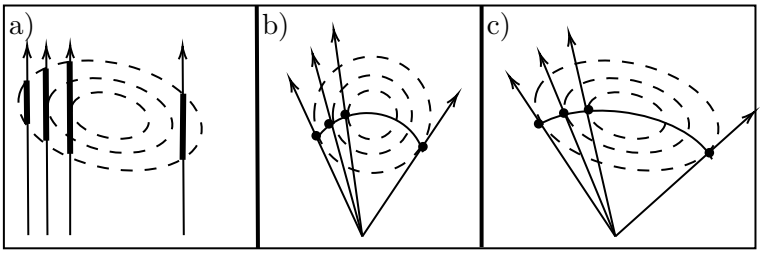


Figure 3.3: a) splatting integration in linearized ray space; b) maximum contributing samples for isotropic Gaussian, corresponding to closest points to the mean, are distributed along a slice of the Gaussian with a quadratic curve; c) anisotropy produced by switching the metric tensor for inverse covariance.

Following 3DGS, the method uses the RTE formulation 3.1 with the response α_i evaluated once per particle. However, instead of evaluating the depth integral of the 3D Gaussian per pixel, the ray tracing approach approximates this integral in 3D space. This is the fundamental difference of the ray tracing and splatting rendering approximation, sketched in 3.3. In case of Gaussian splatting (a), the approximation leads to the splatting integral computed as in 2.8. Ray tracing, on the other hand, computes the integral directly without transforming the ray space. 3DGRT proposes to replace the integral with a single most-contributing sample (c), effectively evaluating the slice of the Gaussian, rather than a projected footprint. Formally, this approach can be seen as the Laplace method for the integral approximation, namely

$$\int_{-\infty}^{\infty} g_v(t)dt \approx K\sqrt{\pi v}g_v(t_{\max}) \qquad v \to 0$$

$$g_v(t) = Ke^{-\frac{(t-t_{\max})^2}{v}}$$

where $g(t)$ is the Gaussian density along the ray with the global maximum at $t_{\max}$. The scaling factor $K\sqrt{\pi v}$ is again approximated globally by the learned opacity. The method therefore is not quadrature and does not treat 3D Gaussians as a density field. Instead, it can be thought of as "2.5D" or billboard tracing.

For a ray parameterized as $\mathbf{r}(t) = \mathbf{c} + t\mathbf{d}$ and an isotropic Gaussian as in (b), the maximum response is achieved at

$$\mathbf{x}_{\max} = \mathbf{o} + \frac{\mathbf{d}}{\|\mathbf{d}\|}\frac{\mathbf{d}^T\mathbf{o}}{\|\mathbf{d}\|}$$

for $\mathbf{o} = (\mu - \mathbf{c})$, therefore $t_{max} = \frac{\mathbf{d}^T\mathbf{o}}{\mathbf{d}^T\mathbf{d}}$. To obtain an analogous expression for an anisotropic Gaussian, we move to the metric space defined by the inner product

$$\langle \mathbf{a}, \mathbf{b} \rangle_{\Sigma^{-1}} = \mathbf{a}^T\boldsymbol{\Sigma}^{-1}\mathbf{b} = \mathbf{a}^T\mathbf{M}^T\mathbf{M}\mathbf{b} = \langle \mathbf{a}_g, \mathbf{b}_g \rangle$$

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

This transformation $\mathbf{a} \to \mathbf{a}_g$ is known as whitening. For a general Gaussian, the maximum response is at

$$\begin{aligned} &\mathbf{x}_g = \mathbf{c}_g + \mathbf{d}_g t \qquad t = \frac{\mathbf{d}_g^T\mathbf{o}_g}{\mathbf{d}_g^T\mathbf{d}_g} \\ &\mathbf{M}^{-1}(\mathbf{M}\mathbf{x}) = \mathbf{M}^{-1}(\mathbf{M}\mathbf{c} + \mathbf{M}\mathbf{d}t) \\ &\mathbf{x} = \mathbf{c} + \mathbf{d}t \end{aligned} \tag{3.8}$$

evaluated as

$$\alpha_i = \sigma_i G_\Sigma(\mathbf{x}_i)$$

for the i-th Gaussian along the ray.

Following 3DGS, the emission is also evaluated via SH as $c(\mathbf{d})$. Since the rendering is ray-based, the direction $\mathbf{d} = (\mu - \mathbf{C})$ can be replaced with the ray direction. This does not significantly impact the result, since the Gaussians are small, but eliminates the contribution of SH to the position gradient, simplifying the computation.

Gradients

Since the RTE formulation is the same as in splatting, the derivatives with respect to color and response are equivalent. For numerical and implementation reasons, the chosen evaluation order is **front-to-back** as in 3.7.

Since the response is now with respect to 3D covariance, we can apply the decomposition for a more efficient gradient evaluation.

$$G_\Sigma = e^{-\frac{1}{2}\mathbf{p_g}^T\mathbf{p_g}} \qquad \mathbf{p_g} = \mathbf{M}(\mathbf{x} - \mu_\mathbf{i}) \tag{3.9}$$

$$\frac{\partial \alpha_i}{\partial \mathbf{p}_g} = -\alpha_i \mathbf{p_g} \qquad \frac{\partial \mathbf{p_g}}{\partial \mathbf{x}} = \mathbf{M_i} \qquad \frac{\partial \mathbf{p_g}}{\partial \mu} = -\mathbf{M_i}$$

We then propagate the gradient with respect to $\mathbf{o_g}$ and $\mathbf{d_g}$ through $\frac{\partial \mathbf{x}}{\partial t} = \mathbf{d}$ as

$$\frac{\partial t}{\partial \mathbf{o_g}} = -\frac{-\mathbf{d}_g}{\mathbf{d}_g^T \mathbf{d}} \qquad \frac{\partial t}{\partial \mathbf{d_g}} = \frac{\mathbf{o}_g^T \mathbf{d}_g^T \mathbf{d}_g - \mathbf{o}_g^T \mathbf{d}_g (2\mathbf{d}_g^T)}{(\mathbf{d}_g^T \mathbf{d}_g)^2} \tag{3.10}$$

Note that for $t^* = \arg\max_t \rho(t)$ derived in 3.8, the contribution of $\frac{\partial t}{\partial \mathbf{o_g}}$ and $\frac{\partial t}{\partial \mathbf{d_g}}$ should be zero, since $\frac{d\rho}{dt} = 0$. We will still keep these gradient paths, as they will be important later. The parameter gradients are then made up of

$$\frac{\partial \alpha_i}{\partial \mu_\mathbf{i}} = \sum_{\mathbf{v_g} \in \{\mathbf{p_g}, \mathbf{o_g}\}} \frac{\partial \alpha_i}{\partial \mathbf{v_g}} \frac{\partial \mathbf{v_g}}{\partial \mu_\mathbf{i}} \qquad \frac{\partial \alpha_i}{\partial \mathbf{M_i}} = \sum_{\mathbf{v_g} \in \{\mathbf{p_g}, \mathbf{o_g}, \mathbf{d_g}\}} \frac{\partial \alpha_i}{\partial \mathbf{v_g}} \frac{\partial \mathbf{v_g}}{\partial \mathbf{M_i}} \tag{3.11}$$

To compute the matrix derivatives only with respect to scalars, we introduce $\partial \mathbf{X}$ as a shorthand for $\frac{\partial \mathbf{X}}{\partial x}$. We also denote $\mathbf{v_g} = \mathbf{M}\mathbf{v}$, applying

$$\partial(\mathbf{M}\mathbf{v}) = (\partial \mathbf{M})\mathbf{v} + \mathbf{M}\partial \mathbf{v}$$

$$\frac{\partial \alpha_i}{\partial \mathbf{v_g}} \partial \mathbf{v_g} = \frac{\partial \alpha_i}{\partial \mathbf{v_g}} (\partial \mathbf{M_i})\mathbf{v} + \frac{\partial \alpha_i}{\partial \mathbf{v_g}} \mathbf{M_i} \partial \mathbf{v}$$

Since $\frac{\partial \mathbf{v}}{\partial \mu_\mathbf{i}} \in \{\mathbf{I}, \mathbf{0}\}$, with respect to the positions we get:

$$\frac{\partial \alpha_i}{\partial \mu_\mathbf{i}} = \sum_{\mathbf{v_g} \in \{\mathbf{p_g}, \mathbf{o_g}\}} \frac{\partial \alpha_i}{\partial \mathbf{v_g}} \mathbf{M_i}$$

Expanding $\partial \mathbf{M}$ analogously, we compute the derivatives with respect to scales $\mathbf{s_i} = (s_k)_{k=1}^3$ and rotations $\mathbf{q_i} = (q_k)_{k=1}^4$ as:

$$\partial \mathbf{M} = \partial(\mathbf{S^{-1}})\mathbf{R}^T + \mathbf{S}^{-1}(\partial \mathbf{R})^T$$

$$\frac{\partial \alpha_i}{\partial s_k} = \sum_{\mathbf{v_g} \in \{\mathbf{p_g}, \mathbf{o_g}, \mathbf{d_g}\}} \left[\frac{\partial \alpha_i}{\partial \mathbf{v_g}} \frac{\partial \mathbf{M_i}}{\partial s_k} \mathbf{v} = \frac{\partial \alpha_i}{\partial \mathbf{v_g}} \frac{\partial \mathbf{S^{-1}}}{\partial s_k} \mathbf{R}^T \mathbf{v} = -\frac{\partial \alpha_i}{\partial v_g^k} \frac{1}{s_k^2} \mathbf{r_k}^T \mathbf{v} \right]$$

$$\frac{\partial \alpha_i}{\partial q_k} = \sum_{\mathbf{v_g} \in \{\mathbf{p_g}, \mathbf{o_g}, \mathbf{d_g}\}} \left[\frac{\partial \alpha_i}{\partial \mathbf{v_g}} \mathbf{S}^{-1} \left(\frac{\partial \mathbf{R}}{\partial q_k} \right)^T \mathbf{v} \right]$$

3.3.4 Numerical stability

The key numerical issue is in the computation of 3.8. For normalized directions, the denominator $\mathbf{d_g}^T \mathbf{d_g}$ depends on the covariance scale as $\frac{1}{s^2}$ and, therefore, will be small, while the numerator depends on $\mathbf{o_g} = \mathbf{M}(\mathbf{c} - \mu)$, growing larger with camera distance. For the numerator with exponent 2^p, and the denominator with exponent 2^q the number $d = (p - q)$ can be large and growing with depth. The absolute error of the denominator will be defined by the ULP corresponding to the exponent 2^q, but the exponent of the resulting ration will be 2^d, amplifying the absolute error of the denominator by the factor of 2^{d-q}. This results in the estimate of t^* being imprecise for particles further away, potentially falling outside of the Gaussian cutoff region, but does not produce a noticeable impact on quality.

However, the situation is more dangerous when computing the position gradient 3.9. The gradient of $\frac{\partial(\mathbf{p_g}^T\mathbf{p_g})}{\partial\mu_\mathbf{i}} = -2\Sigma_i^{-1}\mathbf{x}(t) + ...$ growth with t, which potentially results in overestimation of the norm of position gradient. Such an error may not significantly affect the gradient optimization, but for the position gradient based ADC it would be detrimental, as we will demonstrate in the next chapter.

The problem of large t can be directly addressed using large unnormalized ray directions, but this would simply move the error amplification to the product $\mathbf{d}t$. Applying the classical solution

$$t^* = \frac{\mathbf{o_g}^T\mathbf{d_g}}{\max(\mathbf{d_g}\mathbf{d_g}, \epsilon)}$$

for $\epsilon = 10^{-6}$ is what we will demonstrate to be insufficient for the task. Fortunately, there is a better solution than tuning the epsilon to be larger.

We can avoid computing the ray parameter entirely by instead directly evaluating the product $\mathbf{p_g}^T\mathbf{p_g}$ in whitened space as:

$$\mathbf{p_g}^T\mathbf{p_g} = \|\mathbf{x}_g - \mu_g\|^2 = (\sin\theta\|\mu_\mathbf{g} - \mathbf{o_g}\|)^2 = \|\frac{\mathbf{d}_g}{\|\mathbf{d_g}\|} \times \mathbf{o_g}\|$$

This eliminates the numerical instability, replacing the division by $\|\mathbf{d}_g\|^2$ by just normalization. The gradient computation is analogous, following the paths $\mathbf{v}_g \in \{\mathbf{d}_g, \mathbf{o}_g\}$ in 3.11. The Jacobians of the cross product are the cross product matrices $-[\mathbf{o}_g]_\times$ and $[\mathbf{d_g}]_\times$.

Optimization

The optimization setup of this ray tracing method is largely inherited from 3DGS, including the densification procedure. However, the letter has to be modified for the new setup:

- The density approximation in 3.3 causes the depth integral to be smaller, and therefore the optimal opacity term to be larger. 3DGRT addresses this by increasing the learning rate for the opacity group and the pruning threshold.

- Since the screen space gradients are not computed, the densification criterion is the norm of 3D positional gradients, which will be larger roughly by the factor of $|K\mathbf{J}_{\bar{\mu}}|$. To compensate for this, the authors propose to increase the densification interval from 100 to 300, while keeping the pruning interval the same. This empirical solution therefore tries to mitigate the growth by running more pruning steps.

- The authors additionally suggest to multiply the densification criterion by half camera distance for each particle to increase the amount of detail in far away regions.

This densification procedure is quite brittle and does not adapt well to new camera setups, as will be demonstrated in our experiments.

Ray tracing additionally enables the possibility of reformulating the objective in terms of ray batches as in 2.13. However, this would disable the use of the structural similarity metric. This generally leads to worse training outcomes, degrading both SSIM [54] and PSNR with blurry artifacts.

Proxy mesh

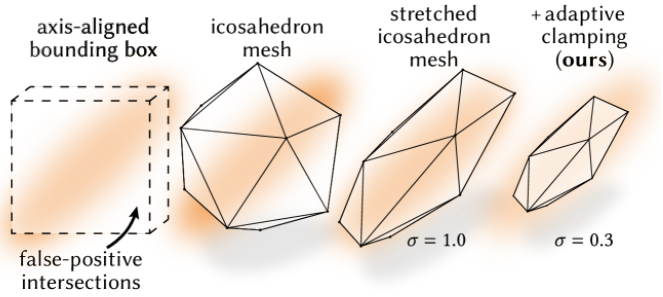


Figure 3.4: Fitting and adaptive scaling of the icosahedron mesh (3DGRT[37])

For efficient ray tracing on GPU the algorithm is implemented in OptiX framework [41], which handles building and traversing the scene BVH. The rendered primitives are ellipsoids defined by a set of points satisfying

$$\alpha_i(\mathbf{x}) > \alpha_{min}$$

These can be naturally represented in OptiX as *custom primitives*, meant for defining general intersections inside of the corresponding axis aligned bounding box (AABB), which provides intersection candidates at low cost. However, since the bounding box is not tight around the ellipsoid, many candidate intersections will end up rejected by the costly custom intersection procedure.

The key idea behind 3DGRT[37] rendering acceleration is, rather than using custom primitives, to design a tight proxy mesh mapping to the primitives. This would reduce the number of rejected intersections while utilizing the fast triangle intersections for low candidate cost. On the other hand, this would come with a memory and potentially performance cost if the number of mesh triangles per particle is large. The authors[37] propose an *icosahedron* mesh of 20 triangles and 12 vertices, uniformly enclosing the unit sphere, stretched to enclose the quadric of the Gaussian response. This adaptive scaling can be derived from the threshold:

$$\sigma e^{-\frac{1}{2}\mathbf{p_g}^T\mathbf{p_g}} > \alpha_{min}$$

$$\|\mathbf{p_g}\| < \sqrt{2\log\frac{\sigma}{\alpha_{min}}}$$

We then stretch the unit sphere to match the threshold, transform the point to unwhitened space, and move them to the mean, which results in this transformation for the original icosahedron vertices $\mathbf{v}$:

$$\mathbf{v}' = \sqrt{2 \log \frac{\sigma}{\alpha_{min}}} \mathbf{M}\mathbf{v} + \mu$$

The case where σ drops below α_{min} corresponds to the particle not being rendered assuming that α_{min} corresponds to the shading threshold (used to accept or reject the sample). However, the official implementation instead uses α_{min} **higher** than the shading threshold, so the mesh is tighter than the Gaussian ellipsoid. The rendered footprint is then defined by the mesh, however, since the contributions at the boundary are small, it does not lead to significant artifacts. Instead, it significantly improves performance by avoiding the rejection of candidate intersections almost entirely.

To handle the numeric error and avoid a degenerate mesh, the authors propose to restrict the minimum scale to $g_{min} = 0.97$ fraction of the kernel density as:

$$\mathbf{v}' = \sqrt{-2 \log \min(\frac{\alpha_{min}}{\sigma}, g_{min})} \mathbf{M}\mathbf{v} + \mu$$

Directly implementing the formula like this would allow to even create meaningful meshes for Gaussians whose opacity is zero. Until these particles are pruned, they continue to generate intersections, since pruning at each iteration would be prohibitively expensive, and creating degenerate triangles could unpredictably affect the OptiX [41] pipeline.

Algorithm

The problem solved by the fast ray tracing algorithm is per-pixel depth sorting and blending of Gaussian primitive intersections. One way to achieve this would be to store all the intersections in large per-pixel buffers with sizes determined by an additional pass, similarly as it is done in 3DGS renderer per tile. However, per-pixel sorting done in this way would require significantly more memory, in addition to the cost of initial pass, allocation, and global memory accesses. Following Knoll et al.[28], the authors propose to instead use only a small sorting buffer of k intersections. Once the buffer is full, it is flushed into the radiance and transmittance accumulators, so the traversal can proceed from the furthest intersected point with an empty buffer. This algorithm can be implemented in OptiX using only *raygen* and *any-hit* programs, since the intersections are handled by the triangular proxy mesh. It inherits the 3DGS blending approach, stopping when the accumulated transmittance falls below $T_{min} = 0.01$, and accepting hits in the buffer if their contribution is greater than $\alpha_{min} = \frac{1}{255}$.

Program 1: Ray-Gen

Input: ray $\mathbf{o}$, $\mathbf{d}$, scene parameters
Output: radiance $\mathbf{C}$, transmittance T
$\mathbf{C} = (0., 0., 0.)$
$T = 1.$
$t_{curr} = 0$
while $t_{curr} < \infty$ *and* $T > T_{\min}$ **do**
  TraceRay$(\mathbf{o}, \mathbf{d}, t_{min} = (t_{curr} + \epsilon), t_{max} = \infty) \rightarrow$ sorted k-buffer

  for *(t_{hit}, i = particle id) in k-buffer* **do**
    $t_{curr} = t_{hit}$ Continue from the furthest intersection
    $t_{hit} = \frac{\mathbf{o}_g^T \mathbf{d_g}}{\mathbf{d_g}^T \mathbf{d_g}}$
    $\alpha_i = \sigma_i G_{\Sigma_i}(\mathbf{o} + t_{hit}\mathbf{d})$

    if $G_{\Sigma_i}(\mathbf{o} + t_{hit}\mathbf{d}) >$ *mesh* α_{min} *Milder condition used when mesh is tighter than* α_{min} *(mesh* $\alpha_{min} > \alpha_{min}$*)*
        and
      $\alpha_{hit} > \alpha_{\min}$ **then**
      Accept the hit
      ComputeRadiance$(\mathbf{o}, \mathbf{d}, i_{\text{particle}}) \rightarrow \mathbf{C}_{hit}$
      $\mathbf{C} = \mathbf{C} + T \cdot \alpha_{\text{hit}} \cdot \mathbf{C}_{hit}$
      $T = T \cdot (1 - \alpha_{\text{hit}})$
    end
  end
end
return $\mathbf{C}, T$

The TraceRay function here switches the pipeline to the scene traversal stage, described by the *anyhit* program. Its purpose here is to sort all the reported candidate intersections by t_{hit} into the k-buffer, and stop the traversal once the buffer is full. At the beginning the buffer is initialized as $[\infty...]$.

Note that the sorting order depends on the mesh, rather than on the actual sample position, as it has been stated by the authors to be sufficient. We will still address this option in our experiments.

Another observation here is that the depth ordering is not necessarily perfect, as it would require OptiX to report all closer intersections before the buffer is full, which is not guaranteed. Some hits may therefore be false-rejected. The authors ablate the larger buffer sizes, claiming that bigger buffers would slightly degrade performance, and will be filled with hits that will not be accepted.

Program 2: Any-Hit

Input: mesh t_{hit} and intersected primitive index i_{triag} from OptiX, and the k-buffer
Output: the updated k-buffer
$i = i_{triag}/20$ Triangles of icosahedra are stored sequentially
$t_{hit} =$ mesh t_{hit} Mesh intersection used for sorting
$h = (id = i, t = t_{hit})$
Insert the new hit in order of t
for j *in* $0 \dots k-1$ **do**
  if $h.t < buffer[j].t$ **then**
    swap(buffer$[j], h$)
  end
end
if $h.t < buffer[k-1].t$ **then**
  IgnoreHit() Unless the buffer is full, we continue the traversal by calling this function
end

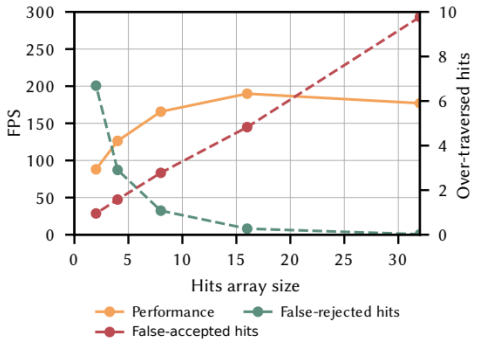


Figure 3.5: The analysis of higher buffer sizes (3DGRT [37])

The authors therefore choose the buffer size of length 16, which additionally exactly fits into the payload registers (of which the maximum is 32), therefore, the buffer does not have to be allocated in memory.

Note that the sorting imperfection is also not guaranteed to be deterministic, so the accumulated transmittance may differ. Since training relies on forward and backward passes being identical, the official implementation stores the maximum intersection distance into a buffer in forward pass to reuse it in the backward pass to stop the traversal instead of the accumulated transmittance.

3.4 Distorted Cameras in Gaussian Splatting

An alternative approach to splatting and pure ray tracing is presented in 3DGUT[56]. The technique demonstrates high quality results, comparable to ray tracing, supports distorted cameras, and is blazing fast.

3.4.1 Unscented transform

Inspired by the approach used in the unscented Kalman filter as an alternative to the extended Kalman filter, the authors propose to represent a Gaussian using a set of predefined sigma points $\mathbf{x_i}$ with corresponding weights w_i^μ and w_i^Σ. To match the first three moments of the Gaussian with dimension D, $2D+1$ points are required. These points can be projected using a generally nonlinear mapping $g(\mathbf{x_i})$ and then used to recreate the Gaussian in screen space as

$$\hat{\mu} = \sum_{i=0}^{6} \hat{\mathbf{x}}_\mathbf{i} w_i^\mu$$

$$\hat{\mathbf{\Sigma}} = \sum_{i=0}^{6} w_i^\Sigma (\hat{\mathbf{x}}_i - \hat{\mu}_\mathbf{i})(\hat{\mathbf{x}}_i - \hat{\mu}_\mathbf{i})^T$$

Note that unlike in splatting, the means are not projected directly, as in general $g(\mathbb{E}X) \neq \mathbb{E}g(X)$.

This approach, called the unscented transform (UT), directly addresses the error of EWA approximation without global ray tracing. The work shows the Kullback-Leiber(KL) divergence of the projected distribution footprint from the precise footprint computed using MonteCarlo sampling.

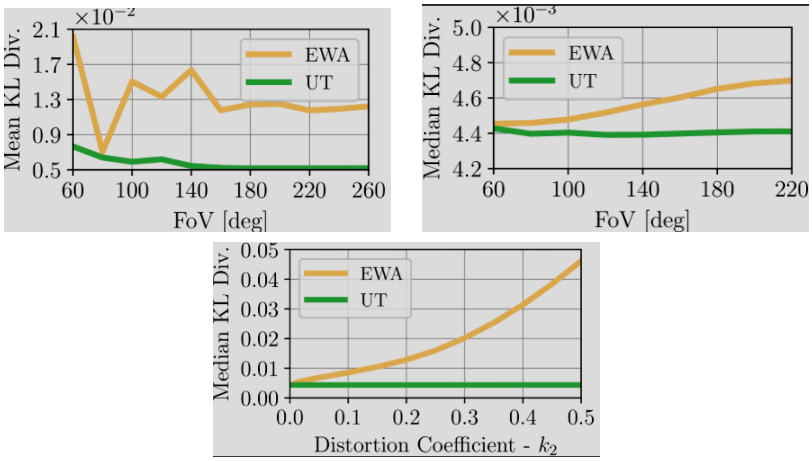


Table 3.2: The difference in footprint error of UT and EWA depending on FOV of fisheye cameras and radial distortion.[56]

The graphs show that for increasing $\theta_{\max}$ in fisheye cameras, the UT approximation is more precise than the one used in splatting. The mean KL divergence is unstable, likely due to the scene structure, as EWA is more precise for smaller Gaussians. The median shows that the approximation does get slightly less precise for larger angles, probably because of the mean projection, rather than covariances. The distortion in the experiments was not linearized, however if it was, the error would also increase proportionally to distortion, while UT remains completely unaffected. The evaluation in table 3.3 shows that the difference between the projection approximations is

not very significant in the *bike* scene with a relatively small fisheye FOV, and is negligible for pinhole cameras without distortion.

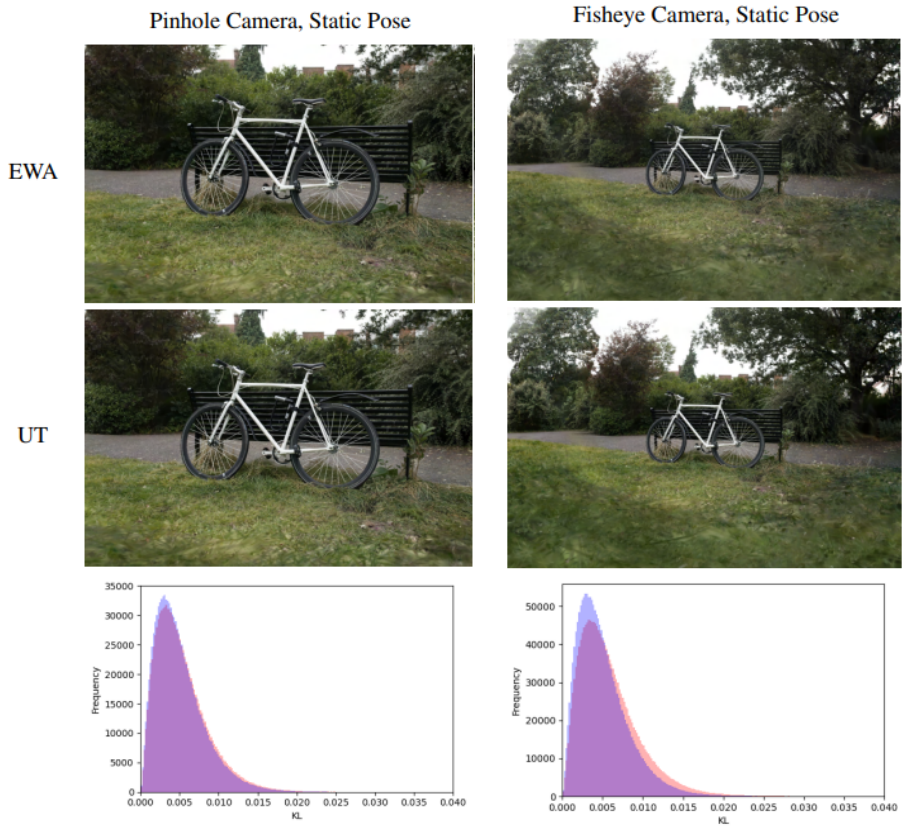


Table 3.3: The frequency histograms show how often the unscented transform is closer to the correct projection than EWA.[56]

3.4.2 Renderer

In terms of rendering algorithm, 3DGUT can be placed between splatting and ray tracing. After projecting Gaussians using the unscented transform, it applies tile based culling and sorting based on means, analogous to the splatting approach. However, it does not use the footprint rasterization, but instead evaluates the sample response in 3D, analogously to 3DGRT. The tile-based algorithm therefore replaces the ray tracing acceleration structure traversal, which is the bottleneck of the ray tracing performance.

Optionally, the Gaussians sorted per tile can be sorted per pixel to produce a better result at low cost, since they are already partially sorted. The approach applies a sorting technique based on the ray t corresponding to the evaluated sample using a typically small rolling buffer. The technique, also called multi-layers alpha blending approximation (MLAB [45]), is analogous to the one we implement and test in this work for ray tracing without knowing about it.

Chapter 4

Implementation

This chapter will introduce our custom Gaussian ray tracer implemented from scratch based on the 3DGRT work, discussing some pitfalls, as well as an alternative rendering algorithm. The implementation is done using the OptiX framework as the main backend and PyTorch [42] for training.

4.1 Rendering

The initial ray tracing approach was implemented in OptiX according to the original paper. Due to the absence of reference in the form of scenes trained with ray tracing, it has been decided to test the implementation on scenes trained with 3DGS to ensure correctness before implementing the optimization. However, the mesh based sorting turned out to be a crude approximation to the 3DGS sorting strategy, resulting in significant visual artifacts. Experimenting with different sorting strategies has led to one possible redesign of the rendering algorithm, resulting from an observation that multiple traversals, as in the case of the naive closest hit strategy, are expensive.

Instead of tracing in slabs, the idea is to use a rolling buffer of sorted samples. Whenever the buffer is full, the smallest sample is popped and accumulated, and the newly arrived sample is added in its place. This approach does not guaranty the perfect sorting, but the situation where the sorting is not perfect can be detected.

Suppose that the buffer is represented by a set of samples $|R^{(1)}|$ =capacity, and no samples have yet been popped. When the new sample with distance t_2 arrives, the buffer changes to $R^{(2)} = \{t_2\} \cup (R^{(1)} \setminus \{R^{(1)}_{min}\})$. If the sample satisfies $R^{(1)}_{min} \leq t_2$, then since $R^{1}_{min} \leq R^{(1)} \setminus \{R^{(1)}_{min}\}$, it holds that $R^{(1)}_{min} \leq R^{(2)}_{min}$, so by induction the sorting is perfect. Conversely, if $R^{(1)}_{min} > t_2$ it means that $R^{(1)}_{min} > R^{(2)}_{min}$, so the next sample popped from the buffer will be out of order.

Applying this, we can allocate a large buffer with a limited maximum capacity, recasting the ray again with increased capacity each time the wrong ordering is detected. Running this process on the scenes from NeRF-synthetic dataset has shown that for some scenes with high depth com-

plexity, namely *mic* and *ship*, the required buffer size reaches and for some views exceeds 1K samples. However, these are extreme cases, and on real scenes the depth complexity is usually much lower. For training, the buffer size of 1024 has shown to be sufficient while still fitting into local memory on a modern GPU. The sample accumulation then happens in *anyhit* program up until the minimum transmittance is reached.

Program 3: Any-Hit

Input: mesh t_{hit} and intersected primitive index i_{triag} from OptiX, the buffer, T, $\mathbf{C}$
Output: the updated k-buffer
$h = (id = i_{triag}/20, t = t_{hit})$
Insert the new hit in order of t
if *len(buffer)*$= k$ **then**
  h_{min} = buffer.pop()
  Process hit...
  $i = h_{min}.i$
  $\alpha_i = \sigma_i G_{\Sigma_i}(\mathbf{o} + t_{hit}\mathbf{d})$
  if *Accept hit* ... **then**
    ComputeRadiance$(\mathbf{o}, \mathbf{d}, i) \rightarrow \mathbf{C}_{hit}$
    $\mathbf{C} = \mathbf{C} + T \cdot \alpha_{\text{hit}} \cdot \mathbf{C}_{hit}$
    $T = T \cdot (1 - \alpha_{\text{hit}})$
  end
end
if $T > T_{min}$ **then**
  IgnoreHit()
end
buffer.push(h)

The *raygen* program then collects the hits that remain in the buffer.

Program 4: Ray-Gen

Input: ray $\mathbf{o}$, $\mathbf{d}$, scene parameters
Output: radiance $\mathbf{C}$, transmittance T
$\mathbf{C} = (0., 0., 0.)$
$T = 1.$

TraceRay$(\mathbf{o}, \mathbf{d}, t_{min} = 0, t_{max} = \infty) \rightarrow$ sorted k-buffer

while *not buffer.empty and* $T > T_{min}$ **do**
  h_{min} = buffer.pop()
  Process hit...
  ...
end
return $\mathbf{C}, T$

Since the buffer is larger, the sorting complexity will impact performance. I therefore implement the buffer as a binary heap. This approach shows similar training speed and equivalent outcome to the 3DGRT algorithm and may be a viable alternative. The original 3DGRT version is in $O(N \cdot k)$, while the

rolling buffer leads to $O(N \cdot \log K)$. However, for low N the performance will be similar.

Sample sorting

Instead of sorting the particles by mesh intersection 3.3.4, in this work, we experiment with sorting based on the actual sample distance t, which was neglected in the original work.

The original algorithm can be modified to support sample based sorting by evaluating the position of the sample in the *anyhit* 7 program. In this case, we can also evaluate the response α_i to reject the samples. The result can then stored into the hit buffer to avoid recomputing it in the *raygen* program. However, since the sample position is subject to numeric error, it may happen to fall outside of the bounding mesh. If we then use this sample to cast the next ray, it may intersect the mesh again, causing double contribution. If this sample is additionally the last one on the path, it may even break the renderer. As a robust solution to this, the sample t can be clamped to the mesh t. Additionally, to reduce the absolute error in t, we evaluated the response based on the "local" ray origin, which corresponds to the last intersection in the previous slab, rather than the global ray origin used for gradients.

Program 5: Any-Hit

Input: mesh t_{hit} and intersected primitive index i_{triag} from OptiX, and the k-buffer
Output: the updated k-buffer
$t_{hit} = \frac{\mathbf{o_g}^T \mathbf{d_g}}{\mathbf{d_g}^T \mathbf{d_g}}$
$\alpha_i = \sigma_i G_{\Sigma_i}(\mathbf{o} + t_{hit}\mathbf{d})$
if *not accepted* **then**
    IgnoreHit()
    return
end
$h = (id = i_{triag}/20, t = \max(t_{hit}, \text{mesh } t_{hit}), \alpha = \alpha_i)$
... add hit to the buffer

Program 6: Ray-Gen

Input: ray $\mathbf{o}$, $\mathbf{d}$, scene parameters
Output: radiance $\mathbf{C}$, transmittance T
$\mathbf{C} = (0., 0., 0.)$ $T = 1.$
$t_{curr} = 0$
while $t_{curr} < \infty$ *and* $T > T_{\min}$ **do**
  TraceRay($\mathbf{o} + t_{curr}\mathbf{d}, \mathbf{d}, t_{min} = \epsilon, t_{max} = \infty$) → sorted k-buffer
  for $(t_{hit}, i = particle\ id)$ *in k-buffer* **do**
    $t_{curr} = t_{hit}$
    Accept the hit
    ...
  end
end
return $\mathbf{C}, T$

Alternatively, sample sorting can be performed using the rolling buffer in a similar way without modification. There is no significant performance difference between the two approaches; however, the clamping of t can slightly skew the sorting. This will manifest when using looser icosahedron meshes with $\alpha_{min} = 1/255$ as a marginal degradation in PSNR (≈ 0.1) in comparison to the rolling buffer approach. We therefore choose to restrict the experiments regarding sample sorting to the rolling buffer algorithm, as it is a cleaner and more robust solution.

Perfect sorting

Program 7: Any-Hit

...
if $h.t < buffer[k-1].t$ **then**
  IgnoreHit()
end
IgnoreHit()

The perfect order of the samples can be ensured by a simple modification to the original algorithm. The ray will then traverse the whole scene, ensuring that all hits will be reported. My experiments confirm that this has no impact on the result, aligning with the paper. What does make a difference in appearance is rather the use of mesh based or sample based sorting, as will be demonstrated in the next chapter.

4.1.1 GPU ray tracing

As common in other works in this domain, this implementation uses the acceleration structure provided by OptiX. It focuses on the 3DGRT approach using the wrapping icosahedron mesh, rather than custom primitives. The large triangular mesh is generated in parallel on GPU, and then the OptiX acceleration structure is built over it. As noted earlier, the *anyhit* program can report multiple hits of the same primitive due to the splitting of primitives across

volumes. This behavior can be prevented by passing the build flag **_REQUIRE_SINGLE_ANYHIT_CALL* when building the structure. To reject the *anyhit* call on the internal part of the mesh instead of passing the normals as a separate buffer, the ray flag **_CULL_BACK_FACING_TRIANGLES* can be passed to the tracing function.

The critical point of implementation is the heavy VRAM usage and high build times for large scenes. In such cases, it is recommended to use **_ALLOW_COMPACTION* flag, which can significantly reduce the memory requirement; however, in our scene structure this does not show an effect. In addition, the structure can be built with **_PREFER_FAST_TRACE* or **_PREFER_FAST_BUILD* flags. In our implementation we prefer the former due to large number of *anyhit* calls and recasts.

During training, the structure is fully rebuilt at each step. This behavior is definitely required after densification, though when the number of particles does not change and updates are small, as in later iterations, the structure can instead be updated without the full rebuild. In our experiments, however, this behavior is not included, which degrades training performance.

The transformed icosahedron meshes are stored in the geometry acceleration structure. Since the meshes are identical up to an affine transformation, they could instead be stored in an instance acceleration structure. In this way, each particle mesh would be stored only once, reducing the memory consumption. This part I also leave for future work.

4.2 Gradients

Parameter gradients are computed in a separate pass of the rendering algorithm, identical to the forward pass, and accumulated to scene-sized buffers using atomic addition. The method is wrapped in a custom differentiable PyTorch function, receiving the activated parameters and returning the rendered image for loss computation.

In the initial stages, the correctness was verified using the PyTorch *gradcheck* function, which takes steps in the parameter space and compares the analytic gradients to those estimated numerically using the finite difference method. The method was run using a small randomly generated scene and an arbitrary image. However, the verification was only loose because the order in which gradients are accumulated impacts the resulting numeric error. Making this order deterministic would require running an additional pass to allocate the gradient buffers per particle, to sum them after the backward pass is finished. I did not implement this behavior, instead setting the non-deterministic threshold high. This *gradcheck* is still included in the implementation, so it can be run on small subset of parameters sampled from the visible set on dataset views with some large padding to avoid abrupt jumps in gradients as particles enter or leave the view.

The optimization was then tested on a small random scene with a single view, without densification; however, this test proved not to be sufficient, since the optimization worked for different reasons than expected. Perhaps,

the better approach would be to verify against a predefined blending order using PyTorch automatic differentiation.

While testing the approach on larger scenes, we started with simple RGB colors instead of spherical harmonics, and passed the $S^{-1}R^T$ as an activated parameter to eliminate errors in scale and rotation gradients. On the other hand, in earlier versions, the gradient with respect to quaternion normalization was computed in shader, which was later replaced by an activation on PyTorch side. This approach can still be returned to in the case of heavily optimizing the memory consumption.

4.3 Adaptive density control

In the early stages of this work, the standard 3DGS densification procedure was used. As was later found, using this procedure on the 3D positional gradients leads to a catastrophic failure in the form of exponential pruning and cloning, since the cloned particles in turn get positional gradients above the threshold. In addition, the same behavior may arise due to errors in gradient computation. Plotting the distributions of the Gaussian parameters and positional gradient norms over time, this behavior was unsurprisingly found to be correlated with growth in the number of small opaque particles. To mitigate this growth in redundant geometry, it was decided to slightly raise the pruning threshold, allowing more unnecessary geometry to be removed after the opacity resets, which is similar to the approach used in the later published official implementation. Tuning the densification algorithm that takes a long time to show in an error-prone codebase has proven to be difficult, so I ended up using the settings proposed by 3DGRT.

The densification criterion used is identical, namely, the average norm of 3D positional gradients multiplied by the half camera distance. To match the 3DGS settings it has been attempted to approximate the screen space gradients as projections of 3D gradients to the space orthogonal to ray $\frac{\partial L}{\partial \hat{\mu}} \approx (I - \mathbf{d}\mathbf{d}^T)\frac{\partial L}{\partial \mu}$ for a normalized ray direction $\mathbf{d}$. It was found to have no effect on the norms of position gradients, suggesting that the 3D gradients are mostly orthogonal to the ray.

The version of the implementation used in the experiments has a high memory requirement. Specifically, a scene of 1M particles takes approximately 8Gb of video memory. To enable evaluation on a GPU with memory limited to 16Gb, I decided to use a naive approach, setting the upper limit on the densification procedure, so whenever cloning or splitting step results in more particles than a predefined threshold, this step will not be taken.

4.4 Evaluation framework

For development and evaluation the NerfBaselines[30] framework was chosen for the following reasons:

- Interactive viewer that enables evaluating the results from novel viewpoints not present in the original dataset, and can be accessed remotely from any device. With a small modification, it also enables real time training visualization, which was useful for debugging.

- Dataset loading, which allowed the implementation to easily support most public datasets.

- Fair comparison with existing benchmarks based on the default evaluation protocol.

For experiments on the Zip-NeRF dataset [19], we used the undistortion procedure provided by Zip-NeRF implementation ([4],[3]), rather than the original NerfBaselines undistortion, as it was found to more closely match the training images.

4.5 FisheyeGS

To compare the implemented ray tracer to Gaussian splatting methods on distorted fisheye views we implemented a NerfBaselines wrapper for FisheyeGS. This version of Gaussian splatting, originally developed for the ScanNet++ dataset ([33],[60]), adds the fisheye camera Jacobian and the corresponding backpropagation to the original rasterizer to support the common fisheye camera model described earlier. The basic NerfBaselines method can therefore be obtained simply by replacing the rasterizer in the gaussian splatting method.

Zip-NeRF dataset [19], unlike the one used in FisheyeGS, features distorted fisheye cameras using Brown distortion. The distortion parameters are $k_1 > k_2 > 0$ for radial distortion, while the tangential distortion parameters are close to zero. Since adding the distortion linearization to the FisheyeGS rasterizer would be too much work, we restrict to comparison with the existing method and use a simpler approach to handle distortion. Namely, we map the pixels of the ground truth image to the undistorted ray space, and then project them back into the rendered image as $\mathbf{p} = K\phi(K^{-1}[u, v, 1]^T)$, where ϕ denotes the aforementioned undistortion procedure. The rendered images are then sampled via the differentiable bilinear interpolation to produce the distorted predictions, which are then used to compute the loss.

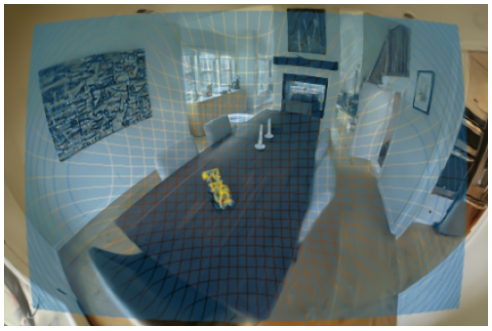
output	ground truth
	
distortion	distorted output
	

Table 4.1: The original fisheye image generated during training does not match the reference. The distortion applied during training and evaluation is shown in blue.

Applying the image distortion introduces some amount of blur due to stretching, so this approach effectively reduces the resolution of the produced image, decreasing the amount of information available for supervision. Fortunately, in this case the distortion is mild, so it may not impact the metrics too much. The authors are reportedly working on a generic support for different kinds of distortion in the differentiable rasterizer.

The public implementation of FisheyeGS [32] clips the projection of means to $\theta = 72.5°$ for performance reasons. Since in our data set, the maximum θ is 88°, this clipping is too restrictive, causing missing geometry at the edges. Removing this restriction is also not sufficient, because the z-clipping in their code is also preserved from the original 3DGS. Additionally, the authors also preserved the depth sorting by z_c, rather than $\|\mathbf{x_c}\|^2$, as it was apparently found to work well on small angles.

Summarizing the above, our version of splatting with fisheye cameras:

- Uses the FisheyeGS code corresponding to derivations of fisheye Jacobians in the paper;
- Removes the angle restriction and z-clipping entirely, allowing for full angle fisheye cameras;
- Uses sorting of means based on distance $\|\mathbf{x_c}\|^2$;
- Applies output image distortion;

- Is wrapped in NerfBaselines.

sorting by z_c

$\theta_{max} = 90°$ (z-clipping)

sorting by distance, full angle

Table 4.2: The sorting artifact appearing at small z_c best visible with wide FoV, still contributes for cameras with FoV close to 180°

Chapter 5

Results and ablations

This chapter demonstrates the experimental verification and comparison for the following:

- Densification behavior in the default 3.1.4 and modified 3.3.4 setup;
- Sample based and mesh based depth sorting 4.1 in training and evaluation, as well as in rendering of pre-trained 3DGS checkpoints;
- Comparison of the previously discussed ray based methods 3.3.33.44.1 to the splatting methods 4.53.1 on selected scenes from commonly used datasets. We also show the comparison on Zip-NeRF dataset[19] with fisheye camera views, to demonstrate the advantage of ray tracing in this setup.

5.1 Densification

Instability

As mentioned in previous sections 3.3.4, using the original 3DGS densification parameters has a catastrophic effect, namely exponential cloning of particles. This behavior does not occur due to small errors in gradient computation, nor can it be mitigated by downscaling the densification criterion in any meaningful way.

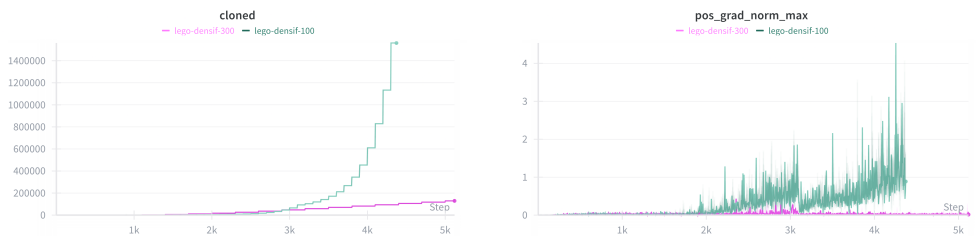


Table 5.1: Exponential cloning on *lego* scene corresponds to growth in norms of positional gradients.

This behavior is correlated with saturation of opacities. Specifically, running densification steps with interval 100 causes the distribution of opacities to become heavily skewed towards 1 around iteration 2000, which in turn causes the positional gradients to become larger. The newly cloned Gaussians then in turn get saturated opacities, leading to exponential cloning. On the other hand, running densification with interval 300, while keeping the prunning interval at 100 allows to prune more particles after the opacity reset, which happens at 3000 to prevent the growth in redundant geometry. This demonstrates the delicate balance needed to keep this thing working.

When gradients are implemented naively using the ray sample directly to compute the response, the densification can exhibit an analogous exponential cloning behavior due to rounding in t, as described earlier. However, this does not happen on most scenes, including large unbounded scenes with complex geometry. The effect was identified when training the *ship* scene from NeRF-synthetic, which has a lot of wide Gaussians representing water.

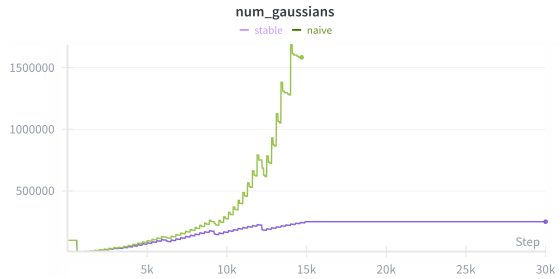


Figure 5.1: Naive gradients on the *ship* scene

Wide meshes

In this approach, heavy pruning occurs after opacity resets, resulting in large jumps in the total number of particles. However, if we make the wrapping mesh wider, so it corresponds to the actual shading threshold, the density will be redistributed more gradually. This will not have a significant impact on the outcome, except a small increase in scene size.

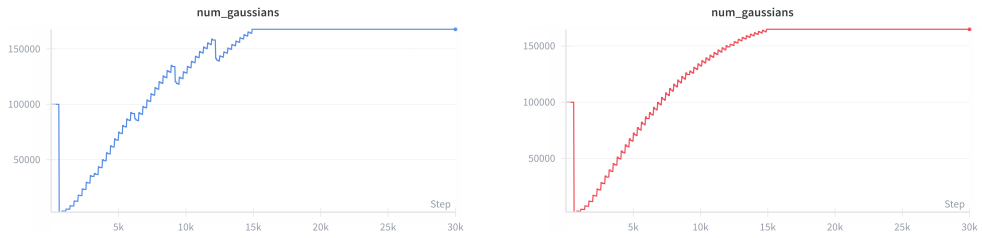
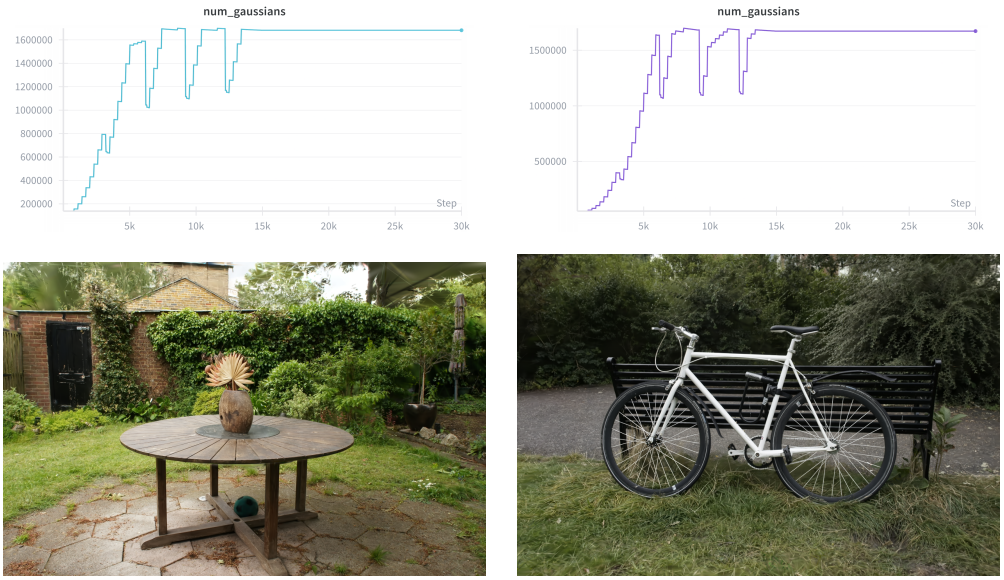


Table 5.2: The default setup (left) compared to wide meshes setup.

Clamping

In this work 3.3.4, we choose to set an upper bound on the number of Gaussians produced by each step of the densification procedure to 1.7M. This may negatively impact the results on the large *garden*, *bicycle*, and *stump* scenes. [1]



PSNR 26.97 PSNR 24.95

Table 5.3: Clamped densification on *garden* and *bicycle* scenes.

An alternative way to reduce the size of the scene would be to train on images with lower resolution, which would reduce the highest frequency of representation. This would allow training all MipNeRF360 scenes with my limited resources. I still include the capped ADC approach in my experiments to compare different versions of the renderer, as well as to demonstrate the decrease in quality.

As will be shown in later sections, our approach shows worse performance in fisheye views than the official 3DGRT implementation. The densification behavior is very different for the fisheye views, since rays in this setup are sparser, producing smaller gradients.

[1] The displayed PSNR values are averages over the training set, not for a specific view.

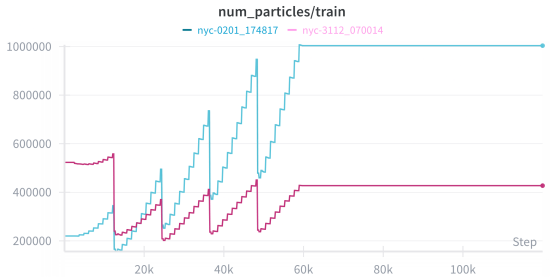


Figure 5.2: Densification comparison for fisheye and undistorted cameras. The former produce much smaller scenes. Note that fisheye and undistorted dataset versions use different initial point clouds.

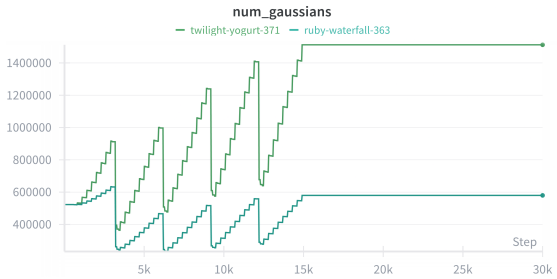


Figure 5.3: Fisheye densification with the modified densification criterion produces a larger scene, behaving similarly to the undistorted case. Both graphs show fisheye.

I tried to address this by "upscaling" the scene, changing the densification criterion to $\tan(\|\frac{\partial \mathbf{L}}{\partial \mu_i}\| \cdot \|\mathbf{c} - \mu_i\|)$ to mimic what the 3DGS projection approximation does to the gradients. This resulted in a stable densification behavior in a sense it did not result in exponential cloning, and the size of the produced scene is much larger. However, in our experiments, this did not show improvement, instead resulting in a significant degradation in quality.

5.2 Renderer

Sorting strategy and mesh clamping

The evaluations presented in this work are run on both versions of the renderer, namely, the versions with sample based and mesh based sorting. The difference between the two approaches is visible when rendering 3DGS scenes. We also address the use of wide meshes to show that they do not produce a significant difference.

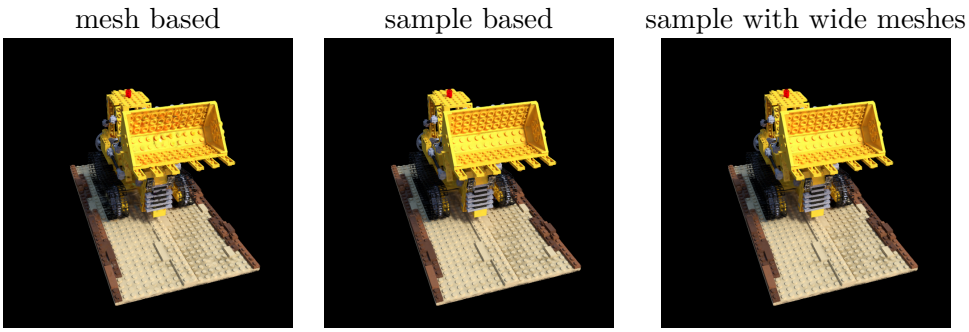


Table 5.4: Rendering the pre-trained 3DGS[37] checkpoint.

The difference from 3DGS results in these renders I would attribute to the projection approximation, which causes Gaussians to appear larger on the screen, and to the shrinkage bias of 3DGS due to wide prefilters.

	PSNR↑	FPS	PSNR↑(wide mesh)	FPS
drums	25.96	116	25.77	75
ficus	36.43	211	36.44	129
mic	35.96	129	35.96	80
ship	31.73	81	31.7	53

Table 5.5: Trained scenes with sample based sorting. The wider meshes show very similar results, while significantly degrading performance.

The following table shows that in most cases, sample based sorting produces a marginal, and sometimes even noticeable improvement. It also removes certain artifacts that may not significantly impact the metrics. As the qualitative results in the appendix A show, this approach significantly improves structure in cluttered and obscured regions. However, it sometimes leads to a subtler blur and color desaturation. This will not affect the structural similarity, but will lead to decrease in PSNR.

	mesh			sample		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
drums	25.81	0.952	0.043	25.96	0.954	0.039
lego	36.47	0.984	0.015	36.64	0.984	0.014
materials	30.30	0.960	0.037	30.30	0.960	0.036
mic	36.04	0.992	0.007	35.96	0.992	0.007
ship	31.70	0.909	0.098	31.73	0.909	0.098
bicycle	24.81	0.743	0.251	24.97	0.746	0.248
bonsai	31.82	0.942	0.203	31.99	0.944	0.200
counter	28.56	0.902	0.204	28.97	0.910	0.195
garden	26.44	0.837	0.149	26.99	0.854	0.130
stump	26.22	0.763	0.239	26.41	0.769	0.231
alameda	22.08	0.798	0.216	22.26	0.806	0.206
london	27.05	0.861	0.195	27.20	0.866	0.187
nyc	27.87	0.897	0.132	28.16	0.901	0.127
fisheye						
alameda	20.84	0.762	0.201	20.87	0.765	0.197
london	23.81	0.770	0.249	23.94	0.777	0.240
nyc	26.12	0.869	0.102	26.22	0.872	0.097

Table 5.6: Performance comparison for the sorting strategy

Splatting benchmark

Compared to 3DGS in the regular setup, we expect the method to show some improvement in metrics due to the per-pixel sorting. Unfortunately, for this benchmark, We relied on metrics from NerfBaselines website, but was too late to realize the difference in evaluation protocol. My scenes were trained on black background, which can improve the PSNR and consequently SSIM metrics. These results are therefore **not directly comparable** in these metrics.

The failure of our method in some cases will also be visible in the extensive qualitative evaluation of the *alameda* scene, showing a noticeable blur. The clamping of densification results in a noticeable degradation in quality, which suggests that the final scene size is somewhat optimal. Contrary to the expectation, on *bonsai* and *drums* our method shows lower PSNR. The most noticeable, however, is the miserable failure on the chair scene. As the previous benchmark shows, this is the problem of my rolling buffer implementation.

Distorted fisheye cameras

I tested the current Gaussian methods and this implementation on fisheye views of *alameda*, *london*, and *nyc* scenes from Zip-NeRF dataset, trained with downscale factor of 8. The evaluation was performed using NerfBaselines default protocol. The evaluation of 3DGRUT, the official implementation of 3DGRT and 3DGUT was performed on the original code with kernel degree 2 and ADC densification, applying the NerfBaselines evaluation on the

	Ours			3DGS		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
bicycle(clamp)	24.95	0.745	0.283	25.21	0.764	0.240
garden(clamp)	26.97	0.854	0.150	27.34	0.866	0.124
stump(clamp)	26.41	0.768	0.268	26.58	0.771	0.250
bonsai	31.93	0.942	0.245	32.20	0.941	0.254
counter	28.95	0.908	0.249	28.96	0.907	0.258
kitchen	31.22	0.928	0.152	31.14	0.926	0.155
drums	25.84	0.953	0.040	26.15	0.953	0.044
lego	36.04	0.984	0.015	35.70	0.982	0.019
materials	29.70	0.961	0.035	30.01	0.959	0.043
mic	35.19	0.991	0.006	35.49	0.991	0.008
ship	30.83	0.908	0.098	30.85	0.904	0.130
ficus	35.96	0.989	0.010	34.79	0.987	0.013
chair	35.92	0.988	0.011	35.84	0.987	0.015

Table 5.7: Comparison of our sample based rolling version to 3DGS on pinhole cameras.

output. The row named GS corresponds to the vanilla 3DGS for undistorted pinhole views and our version of FisheyeGS for fisheye views. The rows named GUT(s.) correspond to 3DGUT using per-pixel sorting with the default buffer of length 16.

	alameda			london			nyc		
	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
GUT	19.73	0.741	0.250	22.59	0.748	0.284	24.38	0.863	0.140
GUT(s.)	19.78	0.748	0.222	23.08	0.761	0.252	25.42	0.878	0.114
GS	20.92	0.764	0.213	23.19	0.743	0.304	25.95	0.862	0.124
GRT	21.06	0.793	0.185	24.00	0.783	0.241	27.86	0.914	0.079
Ours	21.43	0.803	0.175	24.13	0.783	0.238	28.09	0.918	0.074
pinhole undistorted									
GS	20.30	0.730	0.228	22.03	0.723	0.269	24.94	0.843	0.132
GUT	21.40	0.766	0.190	26.38	0.848	0.160	27.05	0.879	0.098
GUT(s.)	21.04	0.766	0.181	26.43	0.854	0.148	27.16	0.881	0.093
GRT	22.01	0.787	0.167	26.87	0.862	0.145	27.65	0.889	0.088
Ours	22.26	0.801	0.153	27.21	0.865	0.139	28.14	0.898	0.078

Table 5.8: Partial evaluation on Zip-NeRF dataset [19]

This evaluation excludes the large *berlin* scene, but it may be sufficient to show the following key results.

- Ray tracing shows a noticeable improvement over splatting when training on fisheye views. However, this improvement may come not only from the absence of EWA approximation but also from per-pixel sorting. Note that scenes trained with splatting on fisheye cameras cannot be viewed

with pinhole camera as a result of particles becoming too wide in screen space.

- The results of ray tracing methods are still better for undistorted views. Despite the larger angles providing more information for supervision, the ray sparsity results in smaller and potentially less accurate gradients, which impacts the descent steps and densification.

- The fisheye splatting method outperforms the unscented transform and shows a stable improvement on fisheye views. In my experiments, it also holds for sorted version of the tracer. Increasing the size of the sorting buffer does not have an effect in 3DGUT, since sorting based on positions is a good ordering approximation.

- My results are worse than the official implementation for the fisheye camera setup, most noticeably on the *nyc* scene. The cause was first attributed to the difference in the undistortion procedure; however, the more likely cause is the yet unexplained blur that happens in certain cases.

- Surprisingly, the sorted version of 3DGUT shows worse PSNR for the undistorted *alameda* scene.

- Our version slightly outperforms the official implementation due to sample based sorting, which corresponds to the above benchmarks.

Our experiments have confirmed that the sample based sorting version of the renderer produces better results than the mesh based version. The version with a smaller buffer size of 512 was also evaluated for fisheye views in experiments that do not appear here. The results were slightly worse for *london* and *nyc* scenes, but slightly better overall for the *alameda* scene.

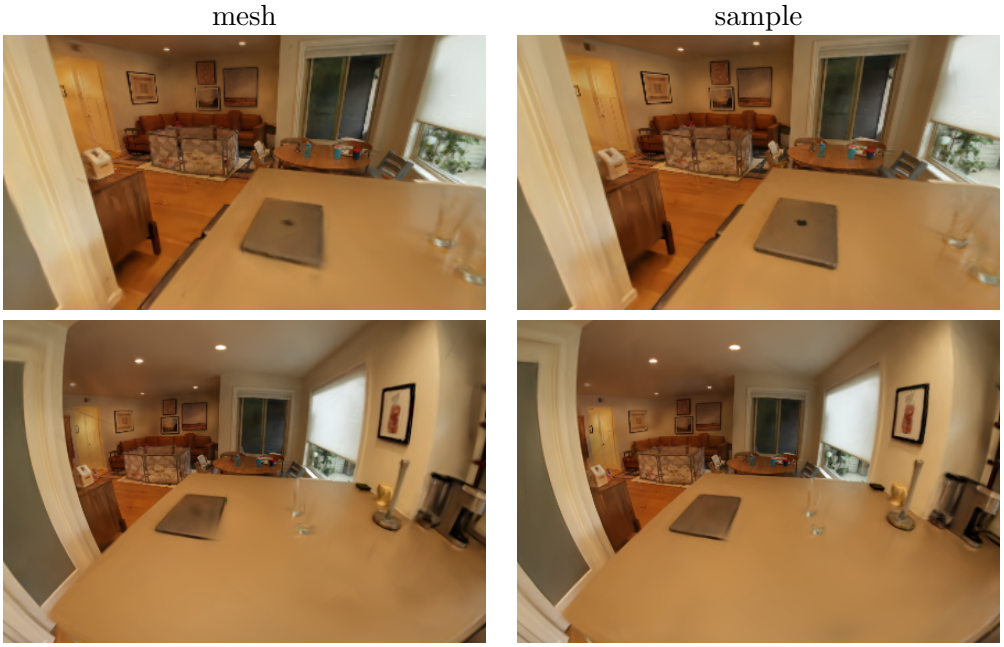


Table 5.9: The effect of renderer version and camera models.

Figure 5.4: Laptop brand is reconstructed in fisheye camera case with the sorting buffer size of 512 instead of 1024.

Inference speed

The size of the rolling buffer has initially been determined for the 3DGS checkpoints. Here, we show the statistics of the maximum image depth complexity accumulated over the test set and the size of the buffer N required to perfectly sort the samples. The size of the buffer is computed in multiples of 16. We also show the average FPS on the test set for the: *1.* rolling buffer with sample based sorting (this); *2.* 3DGRT recasting; and *3.* fast implementation of recasting using payload registers instead of an allocated buffer. The results were computed on NVIDIA GeForce RTX 4060 Ti using a test subset of images.

	size	D_{max}	D_{avg}	N_{max}	FPS (this)	FPS (recast)	FPS (reg.)
lego	314K	636	464	608	66	74	88
drums	352K	685	461	416	73	71	83
mic	307K	1084	773	784	38	38	41
ship	325K	1156	1038	1024	11	20	23
counter	1.2M	634	473	448	3	4	5
stump	4.85M	285	225	240	4	4	4

Table 5.10: The performance on 3DGS scenes.

	size	D_{max}	D_{avg}	N_{max}	MR/s (this)	MR/s (recast)	MR/s (reg.)
lego	314K	636	464	608	42.24	47.36	56.32
drums	352K	685	461	416	46.72	45.44	53.12
mic	307K	1084	773	784	24.32	24.32	26.24
ship	325K	1156	1038	1024	7.04	12.8	14.72
counter	1.2M	634	473	448	1.92	2.56	3.2
stump	4.85M	285	225	240	2.56	2.56	2.56

Table 5.11: The performance on 3DGS scenes.

This shows that the performance of the rolling buffer degrades with depth complexity as we would expect from the sorting complexity; however, for shallow scenes, the performance is similar to the recasting version. Additionally, in large scenes with megapixel resolutions represented here by *counter* and *stump*, the performance is dominated by these parameters, rather than depth, which is smaller in real scenes. Sample based sorting adds the cost of evaluating samples that will not be visible; on the other hand, it allows to reject samples with lower response before they are sorted into the buffer, however, this will only happen rarely with tight meshes. This would therefore be another contributor to the reduced performance of our method.

	size	D_{avg}	FPS (roll)	FPS (reg.)
lego	257K	340	76	107
drums	339K	283	116	147
mic	172K	254	129	131
ship	229K	235	81	93
materials	216K	245	82	118
ficus	168K	210	211	219
bicycle	1.67M	289	3	3
counter	965K	383	4	5
stump	1.68M	274	3	3

Table 5.12: The performance on our trained scenes.

Scenes trained by our ray tracer have a smaller size and lower depth complexity due to the sorting method and the higher T_{min} transmittance threshold, which is highly favorable for performance. The size of the rolling

buffer used here is 1024. However, it can definitely be set lower to reduce memory requirements.

Due to the more costly rendering, but mainly due to frequent data structure rebuilds, our implementation is about 30% slower than 3DGRT. The latter can be straightforwardly fixed in future development.

Chapter 6

Conclusion and future work

In this work we analyzed theoretical foundations, limitations and implementation of current splatting and ray tracing methods for 3D Gaussian-based radiance fields, demonstrating the advantage of the latter due to more faithful rendering. The contributing factors of this advantage in first rays setup were identified to be per-pixel sorting, and the absence of projection approximation in ray tracing.

The work focused on the method of fast ray tracing, namely 3DGRT. We have designed our own minimal implementation of the method, proposing an alternative version of the rendering algorithm, which demonstrates a slightly better performance on undistorted views from Zip-NeRF dataset [19]. The memory cost of our method is higher, however, for sub-megapixel images this cost is often times negligible in comparison to the scene size.

During implementation, we analyzed the adaptive density control algorithm 5.1, demonstrating its main disadvantage in applicability for new rendering setups. In future we consider adding the MCMC densification strategy to our implementation, which can be adapted more easily, and could help with the clutter and light floating artifacts, as discussed in 3.1.4.

We have also adapted the implementation of gaussian splatting for distorted fisheye cameras, which may be useful in future research. Since this method has demonstrated to work well 5.8 in comparison to the unscented transform, we consider adding distortion linerization to this splatting method to be a valuable potential improvement.

In chapter 3 we have discussed the analytic anti-aliasing method used in gaussian splatting, which in our estimation can potentially be applied in the ray tracing setup. We would also note the potential advantage of adopting the analytic Gaussian integration in our fast ray tracing setup.

This work has demonstrated the fundamental advantage of ray tracing in the distorted camera setup 5.8. For future developments, we speculate that setups featuring volumetric media, such as fog or water, can provide a valuable target for the application of ray tracing.

Bibliography

[1] Muhammad Salman Ali et al. *Compression in 3D Gaussian Splatting: A Survey of Methods, Trends, and Future Directions*. arXiv preprint. 2025. DOI: 10.48550/arXiv.2502.19457. arXiv: 2502.19457 [cs.GR]. URL: https://arxiv.org/abs/2502.19457.

[2] Tim Bailey and Hugh Durrant-Whyte. "Simultaneous Localization and Mapping (SLAM): Part II". In: *IEEE Robotics & Automation Magazine* 13.3 (2006), pp. 108–117. DOI: 10.1109/MRA.2006.1678144.

[3] Jonathan T. Barron et al. *CamP Zip-NeRF: A Code Release for CamP and Zip-NeRF*. Software release (JAX). Accessed 2026-01-06. 2024. URL: https://github.com/google-research/google-research/tree/master/camp_zipnerf.

[4] Jonathan T. Barron et al. "Zip-NeRF: Anti-Aliased Grid-Based Neural Radiance Fields". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023, pp. 19697–19705.

[5] Hugo Blanc, Jean-Emmanuel Deschaud, and Alexis Paljic. *RayGauss: Volumetric Gaussian-Based Ray Casting for Photorealistic Novel View Synthesis*. 2024. arXiv: 2408.03356 [cs.CV]. URL: https://arxiv.org/abs/2408.03356.

[6] Michael Bleyer, Christoph Rhemann, and Carsten Rother. "PatchMatch Stereo – Stereo Matching with Slanted Support Windows". In: *Proceedings of the British Machine Vision Conference (BMVC)*. Ed. by Jesse Hoey, Stephen McKenna, and Emanuele Trucco. BMVA Press, 2011, pp. 14.1–14.11. DOI: 10.5244/C.25.14. URL: https://bmva-archive.org.uk/bmvc/2011/proceedings/paper14/.

[7] Gary Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal* (Nov. 2000). Published Nov 1, 2000. URL: https://ptacts.uspto.gov/ptacts/public-informations/petitions/1558289/download-documents?artifactId=8E9mKCVRLTzAIHYKaMMeW0Y7vRNLW2jO3IgIxCIvFTojVG2bGeGwSDI.

[8] D. C. Brown. "Close-Range Camera Calibration". In: *Photogrammetric Engineering* 37.8 (1971), pp. 855–866.

[9] D. C. Brown. "Decentering Distortion of Lenses". In: *Photogrammetric Engineering* 32.3 (1966), pp. 444–462.

[10] Adam Celarek et al. “Does 3D Gaussian Splatting Need Accurate Volumetric Rendering?” In: *Computer Graphics Forum* (2025). DOI: 10.1111/cgf.70032. arXiv: 2502.19318 [cs.CV]. URL: https://arxiv.org/abs/2502.19318.

[11] Jiarui Chen et al. *MEGS^2: Memory-Efficient Gaussian Splatting via Spherical Gaussians and Unified Pruning*. 2025. arXiv: 2509.07021 [cs.CV]. URL: https://arxiv.org/abs/2509.07021.

[12] Jorge Condor et al. *Don't Splat your Gaussians: Volumetric Ray-Traced Primitives for Modeling and Rendering Scattering and Emissive Media*. 2024. arXiv: 2405.15425 [cs.GR]. URL: https://arxiv.org/abs/2405.15425.

[13] Hugh Durrant-Whyte and Tim Bailey. “Simultaneous Localization and Mapping: Part I”. In: *IEEE Robotics & Automation Magazine* 13.2 (2006), pp. 99–110. DOI: 10.1109/MRA.2006.1638022.

[14] Yutao Feng et al. “Gaussian Splashing: Unified Particles for Versatile Motion Synthesis and Rendering”. In: *arXiv preprint arXiv:2401.15318* (2024).

[15] Sara Fridovich-Keil et al. “Plenoxels: Radiance Fields Without Neural Networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022, pp. 5501–5510. URL: https://openaccess.thecvf.com/content/CVPR2022/html/Fridovich-Keil_Plenoxels_Radiance_Fields_Without_Neural_Networks_CVPR_2022_paper.html.

[16] Jean Gallier. *Notes on Spherical Harmonics and Linear Representations of Lie Groups*. Lecture notes, Department of Computer and Information Science, University of Pennsylvania. Dated Nov 13, 2013; accessed 2026-01-06. Nov. 2013. URL: https://www.cis.upenn.edu/~cis6100/sharmonics.pdf.

[17] Jian Gao et al. *Relightable 3D Gaussians: Realistic Point Cloud Relighting with BRDF Decomposition and Ray Tracing*. arXiv preprint, version 2 (Aug 8, 2024). 2024. DOI: 10.48550/arXiv.2311.16043. arXiv: 2311.16043 [cs.CV]. URL: https://arxiv.org/abs/2311.16043.

[18] Michael Goesele et al. “Multi-View Stereo for Community Photo Collections”. In: *2007 IEEE 11th International Conference on Computer Vision (ICCV)*. 2007, pp. 1–8. URL: https://grail.cs.washington.edu/projects/mvscpc/.

[19] Google Research. *Zip-NeRF Dataset (Fisheye and Undistorted Scenes: Alameda, Berlin, London, NYC)*. Dataset, released via the SMERF project page. Four large scenes captured with fisheye cameras; download links provided on the SMERF page (CC-BY 4.0). Accessed 2026-01-06. 2023. URL: https://smerf-3d.github.io/.

[20] Chun Gu et al. “IRGS: Inter-Reflective Gaussian Splatting with 2D Gaussian Ray Tracing”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2025, pp. 10943–10952.

[21] Paul S. Heckbert. “Fundamentals of Texture Mapping and Image Warping”. Master’s thesis. University of California, Berkeley, 1989.

[22] Lukas Höllein et al. *3DGS-LM: Faster Gaussian-Splatting Optimization with Levenberg-Marquardt*. 2024. arXiv: 2409.12892 [cs.CV]. URL: https://arxiv.org/abs/2409.12892.

[23] Binbin Huang et al. *2D Gaussian Splatting for Geometrically Accurate Radiance Fields*. arXiv preprint. 2024. DOI: 10.48550/arXiv.2403.17888. arXiv: 2403.17888 [cs.CV]. URL: https://arxiv.org/abs/2403.17888.

[24] Juho Kannala and Sami S. Brandt. “A Generic Camera Model and Calibration Method for Conventional, Wide-Angle, and Fish-Eye Lenses”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.8 (Aug. 2006), pp. 1335–1340. DOI: 10.1109/TPAMI.2006.153.

[25] Bernhard Kerbl et al. “3D Gaussian Splatting for Real-Time Radiance Field Rendering”. In: *ACM Transactions on Graphics* 42.4 (2023). SIGGRAPH 2023. DOI: 10.1145/3592433. arXiv: 2308.04079 [cs.CV]. URL: https://arxiv.org/abs/2308.04079.

[26] Shakiba Kheradmand et al. *3D Gaussian Splatting as Markov Chain Monte Carlo*. 2024. arXiv: 2404.09591 [cs.CV]. URL: https://arxiv.org/abs/2404.09591.

[27] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations (ICLR)*. 2015. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[28] Aaron Knoll, Ingo Wald, and Paul Navrátil. “Efficient Particle Volume Splatting in a Ray Tracer”. In: *Ray Tracing Gems*. Apress, 2019, pp. 445–461. DOI: 10.1007/978-1-4842-4427-2_29.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012, pp. 1097–1105. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[30] Jonas Kulhanek and Torsten Sattler. *NerfBaselines: Consistent and Reproducible Evaluation of Novel View Synthesis Methods*. 2024. arXiv: 2406.17345 [cs.CV]. URL: https://arxiv.org/abs/2406.17345.

[31] Zhihao Liang et al. “Analytic-Splatting: Anti-Aliased 3D Gaussian Splatting via Analytic Integration”. In: *European Conference on Computer Vision (ECCV)*. 2024. arXiv: 2403.11056 [cs.CV]. URL: https://arxiv.org/abs/2403.11056.

[32] Zimu Liao and contributors. *Fisheye-GS: 3D Gaussian Splatting (3DGS) on fisheye cameras (public implementation)*. GitHub repository. Accessed 2026-01-06. Latest commit on master: 46941fc (Apr 25, 2025). 2025. URL: https://github.com/zmliao/Fisheye-GS.

[33] Zimu Liao et al. *Fisheye-GS: Lightweight and Extensible Gaussian Splatting Module for Fisheye Cameras*. 2024. arXiv: 2409.04751 [cs.CV]. URL: https://arxiv.org/abs/2409.04751.

[34] Huan Ling et al. "Align Your Gaussians: Text-to-4D with Dynamic 3D Gaussians and Composed Diffusion Models". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, pp. 8576–8588.

[35] Nelson Max. "Optical Models for Direct Volume Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108. DOI: 10.1109/2945.468400.

[36] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *European Conference on Computer Vision (ECCV)*. 2020. arXiv: 2003.08934 [cs.CV]. URL: https://arxiv.org/abs/2003.08934.

[37] Nicolas Moenne-Loccoz et al. "3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes". In: *ACM Transactions on Graphics* (2024). SIGGRAPH Asia 2024. DOI: 10.1145/3687934. arXiv: 2407.07090 [cs.GR]. URL: https://arxiv.org/abs/2407.07090.

[38] Thomas Müller et al. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Transactions on Graphics* 41.4 (2022). SIGGRAPH 2022, 102:1–102:15. DOI: 10.1145/3528223.3530127. URL: https://doi.org/10.1145/3528223.3530127.

[39] OpenCV Team. *OpenCV 3.4 Documentation: Fisheye Camera Model (calib3d::fisheye)*. Online documentation. OpenCV 3.4.20-dev documentation page; accessed 2026-01-06. 2026. URL: https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html.

[40] OpenCV Team. *OpenCV Documentation: Camera Calibration and 3D Reconstruction (calib3d module)*. Online documentation. OpenCV 4.14.0-pre documentation page; accessed 2026-01-06. 2026. URL: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html.

[41] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Transactions on Graphics* 29.4 (2010). SIGGRAPH 2010, 66:1–66:13. DOI: 10.1145/1778765.1778803.

[42] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 8024–8035. URL: https://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

[43] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. "Revising Densification in Gaussian Splatting". In: *European Conference on Computer Vision (ECCV)*. 2024. arXiv: 2404.06109 [cs.CV]. URL: https://arxiv.org/abs/2404.06109.

[44] Álvaro Sáez et al. "Real-Time Semantic Segmentation for Fisheye Urban Driving Images Based on ERFNet". In: *Sensors* 19.3 (2019), p. 503. DOI: 10.3390/s19030503. URL: https://www.mdpi.com/1424-8220/19/3/503.

[45] Marco Salvi and Karthik Vaidyanathan. "Multi-Layer Alpha Blending". In: *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14)*. 2014, pp. 151–158. DOI: 10.1145/2556700.2556705.

[46] Johannes L. Schönberger and Jan-Michael Frahm. "Structure-from-Motion Revisited". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 4104–4113. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/Schonberger_Structure-From-Motion_Revisited_CVPR_2016_paper.html.

[47] Johannes Lutz Schönberger et al. "Pixelwise View Selection for Unstructured Multi-View Stereo". In: *Computer Vision – ECCV 2016*. Vol. 9907. Lecture Notes in Computer Science. 2016, pp. 501–518. DOI: 10.1007/978-3-319-46487-9_31.

[48] Volker Schönefeld. *Spherical Harmonics*. Technical Note. RWTH Aachen University, Computer Graphics and Multimedia Group, 2005, p. 18.

[49] Heung-Yeung Shum and Sing Bing Kang. "A Review of Image-based Rendering Techniques". In: *Proceedings of SPIE: Visual Communications and Image Processing 2000*. Ed. by King Ngi Ngan, Thomas Sikora, and Ming-Ting Sun. Vol. 4067. SPIE, 2000, pp. 2–13. DOI: 10.1117/12.386541.

[50] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Published as an ICLR 2015 conference paper. 2014. DOI: 10.48550/arXiv.1409.1556. arXiv: 1409.1556 [cs.CV]. URL: https://arxiv.org/abs/1409.1556.

[51] Noah Snavely, Steven M. Seitz, and Richard Szeliski. "Photo Tourism: Exploring Photo Collections in 3D". In: *ACM Transactions on Graphics* 25.3 (2006). SIGGRAPH 2006, pp. 835–846. DOI: 10.1145/1141911.1141964.

[52] Mikaela Angelina Uy et al. "NeRF Revisited: Fixing Quadrature Instability in Volume Rendering". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023. arXiv: 2310.20685 [cs.CV]. URL: https://arxiv.org/abs/2310.20685.

[53] Junjie Wang et al. “GaussianEditor: Editing 3D Gaussians Delicately with Text Instructions”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, pp. 20902–20911.

[54] Zhou Wang et al. “Image Quality Assessment: From Error Visibility to Structural Similarity”. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[55] Guanjun Wu et al. “4D Gaussian Splatting for Real-Time Dynamic Scene Rendering”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024. arXiv: 2310.08528 [cs.CV]. URL: https://arxiv.org/abs/2310.08528.

[56] Qi Wu et al. *3DGUT: Enabling Distorted Cameras and Secondary Rays in Gaussian Splatting*. 2024. arXiv: 2412.12507 [cs.GR]. URL: https://arxiv.org/abs/2412.12507.

[57] XVERSE Technology Inc. *XScene-UEPlugin: Unreal Engine 5 plugin for real-time 3D Gaussian Splatting rendering*. GitHub repository (tag v1.1.6). Version v1.1.6 (Jul 28, 2025), accessed 2026-01-06. 2025. URL: https://github.com/xverse-engine/XScene-UEPlugin.

[58] Ziyi Yang et al. *Spec-Gaussian: Anisotropic View-Dependent Appearance for 3D Gaussian Splatting*. 2024. arXiv: 2402.15870 [cs.CV]. URL: https://arxiv.org/abs/2402.15870.

[59] Vickie Ye and Angjoo Kanazawa. *Mathematical Supplement for the gsplat Library*. 2023. arXiv: 2312.02121 [cs.MS]. URL: https://arxiv.org/abs/2312.02121.

[60] Chandan Yeshwanth et al. “ScanNet++: A High-Fidelity Dataset of 3D Indoor Scenes”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023, pp. 12–22.

[61] Zehao Yu et al. “Mip-Splatting: Alias-free 3D Gaussian Splatting”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024. arXiv: 2311.16493 [cs.CV]. URL: https://arxiv.org/abs/2311.16493.

[62] Richard Zhang et al. “The Unreasonable Effectiveness of Deep Features as a Perceptual Metric”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 586–595. DOI: 10.1109/CVPR.2018.00068. URL: https://openaccess.thecvf.com/content_cvpr_2018/papers/Zhang_The_Unreasonable_Effectiveness_CVPR_2018_paper.pdf.

[63] Matthias Zwicker et al. “EWA Volume Splatting”. In: *Proceedings of IEEE Visualization 2001 (VIS ’01)*. 2001, pp. 29–36. DOI: 10.1109/VISUAL.2001.964490.

[64] Matthias Zwicker et al. “Surface Splatting”. In: *Proceedings of ACM SIGGRAPH 2001*. 2001, pp. 371–378. DOI: 10.1145/383259.383300.

Appendix A

Qualitative comparison

The following tables show some qualitative effects of mesh based and sample based tracers on the *alameda* scene.

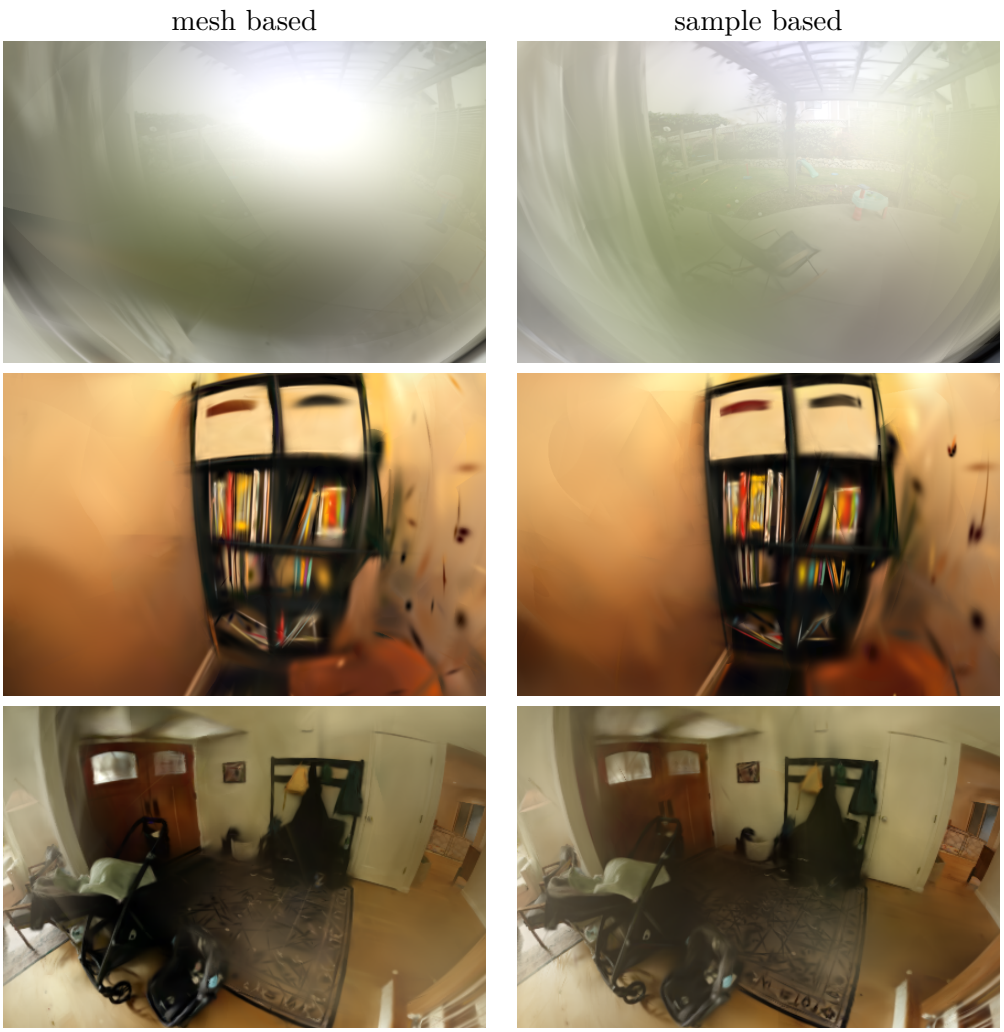


Table A.1: The effect of sample based sorting on fisheye.

Table A.2: The effect of sample based sorting on fisheye.

Table A.3: The effect of sample based sorting on undistorted pinhole.