Rendering Gaussian Splats with Ray-Tracing

Alex Lin
Department of Electrical Engineering
Stanford University
alexlin0@stanford.edu

Meijin Li
Department of Electrical Engineering
Stanford University
meijin@stanford.edu

Yvette Lin
Department of Computer Science
Stanford University
yvelin@stanford.edu

Abstract

We introduce a novel ray-tracing method optimized for real-time rendering of 3D Gaussian splats. Diverging from the rasterizer of the 3DGS paper [5], we utilize direct ray tracing on 3D Gaussians rather than projecting them into 2D space. Our methodology focuses on computing Gaussian intersections using ray tracing and Bounding Volume Hierarchies (BVH) and refining 3D Gaussian models through gradient-based retraining. Innovations in this framework include the application of ray tracing for 3D Gaussian scene representations and the implementation of hardware-accelerated optimization. By conducting performance experiments on two ray-tracing based rendering methods we developed using CUDA and Nvidia Optix 8.0 respectively, we find that hardware-accelerated ray tracing is able to achieve comparable performance to a rasterizer when rendering 3D Gaussian splats in scenes containing more Gaussians but relatively fewer pixels to be rendered. We additionally demonstrate the preliminary feasibility of optimizing 3D Gaussian splats using our renderer by training scenes from scratch and comparing the visual quality to the rasterizer baseline.

1. Background and Setup

This project is focused on developing a high-performance raytracer designed to manage scenes depicted through 3D Gaussian splats, building upon the foundational work of Barron et al. [5]. While the original paper implemented a technique to project 3D Gaussians into 2D, our approach diverges by employing ray tracing directly on the 3D Gaussian objects. This methodological shift is intended to show that it is feasible to implement ray tracing for 3D Gaussian splats.

The project focuses on two main parts: rendering and scene representation training. During the rendering phase, we employ 3D Gaussian splats scene representations to construct BVH and execute ray tracing. Subsequently, we conduct a backward pass to compute gradients directly within the 3D space using CUDA C++ compute shaders. This process allows for retraining and refinement of the 3D scene representations, enhancing the model's accuracy and performance.

From the outset, our project's objective was to explore the feasibility of using ray-tracing as a technique for rasterizing Gaussian splats. The principal question guiding our research was whether a raytracer, designed for real-time rendering with 3D Gaussian splats, could achieve competitive rendering speed and visual quality compared to conventional rasterization techniques, within the limitations of current GPU technology.

2. Related Work

3D Reconstruction. 3D reconstruction is a crucial domain in computer vision and graphics, characterized by a variety of methods to accurately and efficiently represent scenes. Point clouds, among the earliest forms of 3D representation, offer a direct yet sparse depiction of scene geometry. Techniques such as Structure-from-Motion (SfM) [10] and Multi-View Stereo (MVS) [3] have significantly advanced point cloud processing, enabling the use of photo collections for synthesizing novel views and comprehensive 3D reconstructions. Polygonal meshes [6] and voxel-based representations are essential for 3D reconstruction, offering seamless integration with graphics pipelines and simplifying volumetric data handling, respectively, crucial for rendering detailed surfaces and complex internal structures.

Recently, Neural Radiance Fields (NeRF), introduced by Mildenhall et al. [7], utilize a fully connected neural network to encode a scene's volumetric density and color information, facilitating the synthesis of novel views from sparse datasets.

Gaussian Splatting. Gaussian reconstruction kernels, first proposed by Westover [12], provide an innovative alternative to traditional mesh and point cloud representations for capturing 3D object geometry. This method has been further applied by Rhodin et al. [9] and Wang et al. [11] for rendering isolated objects within 3D reconstruction frameworks. Building on these foundations, the seminal work by 3DGS [5] introduced a breakthrough approach, enabling the comprehensive reconstruction of complex scenes, including background elements, using 3D Gaussians. Following this significant advancement, subsequent optimizations and enhancements have been made in both quality and speed by developments such as Mip-Splatting [13], 3DGS-Avatar [8], and Gaussian Surfels [2], further solidifying the role of Gaussian models in modern 3D reconstruction techniques.

3. Approach

We modified the 3D gaussian splatting codebase [5] to implement a renderer using ray-tracing.

We created two ray-tracing based renderers to render gaussian splats: one software ray tracer using CUDA, and one hardware-accelerated ray tracer using the Nvidia Optix 8.0 API. During development, we had another separate hardware-accelerated approach which did not work well. One key challenge to achieving high performance when rendering Gaussians splats is the need to sort the intersected Gaussians based on depth order due to the need for in-order opacity composing. Below, we first describe the successful implementations of the two methods, which both implement Ray-AABB (axis-aligned bounding boxes) intersections on Gaussians represented as AABBs, but solve sorting differently. Then, we describe opacity composing to compute the final color for each pixel. Finally, we describe the iterations we made towards these solutions and describe the attempt which did not work well.

3.1. AABBs from Gaussians

We fit AABBs for each gaussian which encompass 99% of the gaussian's density. The axes of the ellipsoid representing a 3D gaussian can be obtained from the eigenvectors of the covariance matrix. We scale each eigenvector by three times its eigenvalue to obtain the axis vectors for the ellipsoid and surround all positive and negative axis vectors with an axis-aligned bounding box. Conveniently, the covariance matrix of 3D Gaussians are represented as rotation and scaling matrices, $\Sigma = RSS^TR^T$, so the eigenvectors and eigenvalues can be obtained directly from the rotation and scaling matrices of each gaussian.

3.2. CUDA Ray Tracer

We construct a BVH to query ray-AABB intersections. We construct the BVH using CUDA with a parallel algorithm based off of constructing binary radix trees using Morton codes [4]. For a renderer, the performance of BVH construction is not critical, as the BVH is fixed per-scene and is only constructed once.

During BVH traversal, we construct one ray per pixel and parallelize using one CUDA thread per ray. Each thread individually performs BVH traversal to collect all AABBs which intersects the ray. To minimize execution divergence, we implement BVH traversal using a stack as opposed to a recursive implementation. To improve memory performance, each thread stores its stack structure in thread local shared memory, and we fix the stack size per thread to 1024, which becomes the upper bound of the number of Gaussians a single ray can intersect with.

After all intersected Gaussians are collected, we synchronize threads within a thread block to minimize execution divergence. Each thread block contains a 2x2 pixel patch of rays. This minimizes data divergence since rays that are closer together traverse the BVH in a more similar order. Then, we sort Gaussians by depth order with the Thrust API, which uses radix sort. Finally, we synchronize threads once again after sorting and compose Gaussians to compute color, all within a single fused kernel.

3.3. Hardware Accelerated Ray Tracer

The programming model provided by Nvidia Optix 8.0 involves constructing an acceleration structure and writing shaders with pre-specified functions defined by the API whose scheduling are abstracted away by the API. On Nvidia GPUs which support RTX Ray Tracing, hardware-accelerated acceleration structure construction and ray tracing are utilized during host and shader API calls. Our implementation constructs an AABB acceleration structure and implements the *raygen* shader, *intersection* shader, and *closest hit* shader.

The *raygen* shader constructs rays and calls API functions to trace rays in the scene, using a handle to the acceleration structure. The *intersection* shader computes ray AABB intersection times. The *closest hit* shader is triggered on the AABB with the closest intersection time for a single ray. To implement sorting, for each ray, we iteratively query for the closest AABB intersection, update the ray origin to past the closest intersection, and repeat until no more intersections remain. The resulting intersected Gaussians will be in depth sorted order. Then, we use a separate kernel launch for color computation.

3.4. Color Computation

In this section, we describe the methodology for alpha compositing used to determine the visual characteristics of intersections between multiple 3D Gaussian distributions and a ray traced through the scene. The equation for a ray can be expressed as Equation 1, where t is a scalar denoting the position along the ray. We use t_{bounds} to represent the t where Gaussian's bounds intersect with the ray.

$$R(t) = \text{ray\_pos} + t \cdot \text{ray\_dir} \tag{1}$$

3.4.1 Alpha compositing for 2D Guassians

Initially, we apply the alpha compositing technique to render 3D Gaussian splats as described in [5]. Specifically, we project these 3D Gaussian splats onto a 2D, ensuring their compositional accuracy through depth-ordered alpha compositing. This process arranges the splats from front to back. To optimize Gaussian sorting, we employ GPU radix sort and allocate shared memory, which reduces the reliance on global memory operations.

In the rendering of volumetric data, the accumulated color C along a ray can be expressed as a summation over the contributions from samples intersected by the ray, as given by Equation 2, where N is the number of samples along the ray; T_i and c_i is the transmittance and color of the sample respectively.

$$C = \sum_{i=1}^{N} T_i \alpha_i c_i \tag{2}$$

The α value quantifies the fraction of light absorbed by the sample, with higher values indicating denser or thicker samples, calculated using Equation 3. In this expression, ρ denotes the density and w indicates the opacity of the Gaussian sample. In the density ρ calculation, x is the current point being calculated, μ and Σ represent the mean and covariance of the Gaussian, respectively. The distance metric used in this equation is the Mahalanobis distance.

$$\begin{aligned} \alpha &= w \cdot \rho \\ \rho &= \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right) \end{aligned} \tag{3}$$

The transmittance T_i, which accounts for the cumulative effect of light absorption by all preceding samples along the ray, is computed as Equation 4. This recursive formulation ensures that T_i represents the fraction of light that remains unabsorbed upon reaching the i-th sample.

$$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{4}$$

3.4.2 Alpha compositing for 3D Guassians

Then we implement the alpha compositing in 3D space and consider the overlapping 3d Gaussians. To perform alpha compositing, we first sort the Gaussian distributions based on their nearest intersection point with the ray. As the ray progresses through intersections at the same spatial point of multiple Gaussian blobs, we compute the resultant color by aggregating the influence of each intersecting Gaussian. To expedite calculations at each 3D position along the array, we limit our evaluation to the nearest 10 Gaussians. This approach is feasible because the Gaussians are pre-sorted by their proximity to the intersection points, allowing us to reasonably assume that more distant Gaussians are less likely to intersect with the current point. After identifying potential intersections with the nearest Gaussians, we segment the t_{bounds} by steping a fixed length along the ray, facilitating a precise color computation at each segment.

For the spatial point where multiple Gaussian overlap, each Gaussian's contribution to the final color is determined based on its properties. When calculating the density ρ for a Gaussian that is segmented along the ray, the original density should be multiplied by the interval fraction of each segment σ to determine the α, as shown in Equation 5.

It is essential that the calculation of the final color is not influenced by the order in which Gaussians are processed. Different from sequentially accumulating Gaussians at multiple depths or layers to get the pixel's final color, here, all Gaussians contribute at the same spatial point. Therefore, we use a weighted average method based on the characteristics of each Gaussian, detailed in Equation 6, to calculate cumulative color. The total density at this point acts as a normalization factor, ensuring that the resulting color truly reflects the combined effect of all overlapping Gaussians.

At the point of overlap, the combined alpha value is determined by the highest opacity among the intersecting objects, as indicated in Equation 7, assuming the most opaque object dominates the visibility. Then we utilize this α value to blend with other Gaussians at different layers using the approach in Equation 2.

$$\alpha = w \cdot \rho \cdot \sigma \tag{5}$$

$$C = \frac{\sum_{i=1}^{N} \rho_i \alpha_i c_i}{\sum_{i=1}^{N} \rho_i} \tag{6}$$

$$\alpha_{\text{final}} = \max(\alpha_1, \alpha_2, ..., \alpha_n) \tag{7}$$

3.5. Iteration and Failed Attempt

A few key performance lessons learned from tuning performance of the CUDA ray tracer include: (1) recursion causes high execution divergence, (2) keeping local memory within a thread in shared local memory by tuning thread block dimensions is important, (3) syncing threads when thread have variable workloads can greatly minimize execution divergence, and (4) fusing kernels removes the need for a relatively slow store and load to global memory and keeps computation within fast shared memory.

The first iteration of the hardware-accelerated ray tracer used the same idea as the CUDA ray tracer, in that we use the ray tracer to collect all intersections, but not sort them. This was implemented using the *any hit* shader from the Optix API. As expected, collecting all intersections using hardware-acceleration was extremely performant, but unfortunately the global memory costs required for radix sorting was very expensive. Radix sorting required creating key value arrays for all Gaussians intersected per ray, which resulted in 10^9 elements for the stump scene. One lesson learned when writing to global memory from the *any hit* shader is that the indexing and layout of global memory greatly affects global memory write performance. We leave it to future work to explore whether there is a more efficient global memory layout to more performantly construct radix sort keys and values.

4. Evaluation and Results

Our goal from the start of this project was to explore the feasability of using ray-tracing as a technique for rasterizing Gaussian splats. We define feasability as having equal or faster run-time than the rasterizing method used in the 3DGS paper [5]. We hypothesized that a rasterizer's performance is proportional to the number of primitives in the scene, Gaussians in our case, and a ray tracer's performance is proportional to the number of pixels being rendered, so we hypothesized that a ray tracer could exceed a rasterizer's performance when rendering a small number of pixels. To verify this hypothesis, we conduct experiments which measure runtime performance in scenes with varying number of primitives and varying number of pixels, using the 3DGS rasterizer as the baseline.

Additionally, we hypothesized that our renderer, like the baseline rasterizer from the original 3DGS paper, is capable of optimizing Gaussian splats for scene representation through gradient descent, and achieving comparable results. To determine whether this is true, we conduct experiments comparing the visual quality of scenes optimized with our renderer and optimized with the baseline rasterizer.

4.1. Performance Evaluation

For runtime experiments, we evaluate the CUDA ray tracer, hardware-accelerated ray-tracer, and the rasterizer renderer used by the original 3DGS implementation [5] on pre-trained scenes from the Mip-NeRF 360 dataset [1]. The 3DGS rasterizer serves as the baseline. The metric we use for performance is runtime for rendering a single frame of

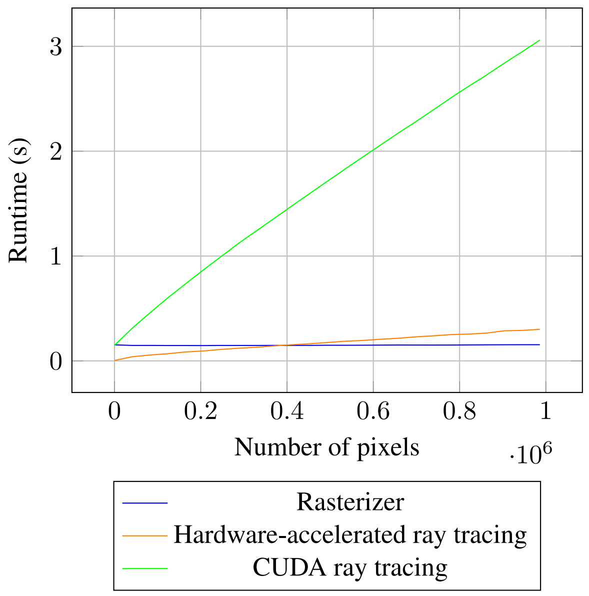

Figure 1. Runtime performance of rendering methods for varying pixel resolutions.

the scene. All runtime experiments were conducted on an Nvidia Tesla T4 GPU. The runtime vs. number of pixels and runtime vs. number of Gaussians experiments use the scene *stump*, which contains 4,961,797 Gaussians and has an aspect ratio of 1245:856.

Runtime vs. number of pixels. We evaluate runtime while varying the number of pixels being rendered while keeping the aspect ratio constant. In Figure 1, we observe a linear relationship between number of pixels and runtime for ray-tracing based renderers, while the rasterizer maintains near-constant runtime. These results make sense since the number of rays is directly proportional to the number of pixels, so naturally there would be a linear performance relationship. The rasterizer, on the other hand, is dominated by the computational costs of rasterizing each gaussian in the scene, which is why we see near-constant performance when varying the number of pixels. The CUDA ray tracer has much higher linear scaling compared to hardware-accelerated ray tracing; we speculate that this may be due to inefficiencies in thread grouping and scheduling, and inefficiencies per ray-BVH traversal leading to much higher constant factors in runtime complexity. To compare the hardware-accelerated ray tracing and rasterizer and estimate the crossover point for the performances of ray tracing versus the rasterizer, we fit a linear model to the runtime data for both methods and compute the number of pixels for which the linear models intersect. We find that the hardware-accelerated ray tracing renderer performs better when there are fewer than 416600 pixels to be ren-

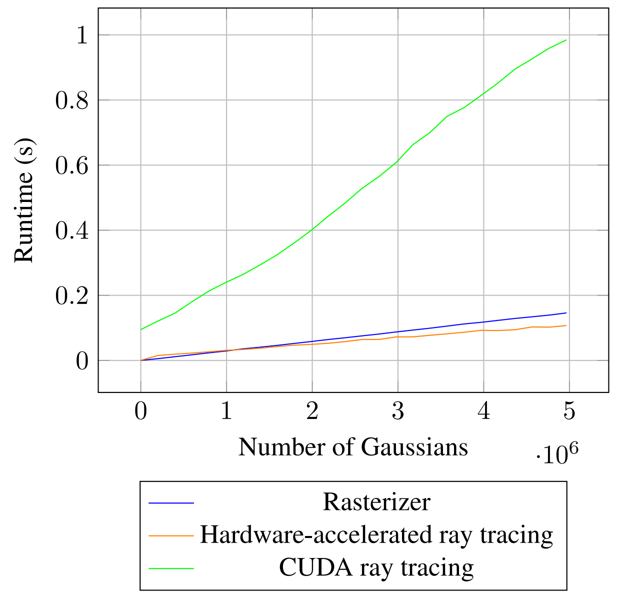

Figure 2. Runtime performance of rendering methods for varying number of Gaussians in the scene.

dered, which is a resolution of approximately 778x535 for this scene.

Runtime vs. number of Gaussians. We evaluate runtime while varying the number of Gaussians in the scene through randomly sampling a subset of Gaussians and rendering 596x410 pixels. We choose a resolution of 596x410 to illustrate how the hardware-accelerated ray tracer and rasterizer scale more clearly by choosing a resolution at which the two methods perform similarly. In Figure 2, we observe linear scaling for all three rendering methods. Like before, the CUDA ray tracing method has much higher linear scaling, likely due to the inefficiencies mentioned before. We also note that the rasterizer has higher linear scaling compared to the hardware-accelerated ray tracer. This makes sense since the work in the rasterizer is dominated by rasterizing each gaussian, whereas in the hardware-accelerated ray tracer, only the BVH acceleration structure and ray intersection queries scales with the number of gaussians, which only scales logarithmically in time complexity.

Baseline performance comparison. To evaluate the performance on scenes from MiP-NeRF 360 in their default configurations, we evaluate performance on different scenes by running the our renderers, plus the rasterization baseline, on the following six pre-trained scenes at default test configurations: *stump*, *bicycle*, *bonsai*, *counter*, *garden*, and *kitchen*. In Figure 3, we visualize the runtime differences for each method per scene. For scenes with larger number of Gaussians such as stump, bicycle, and garden, the perfor-

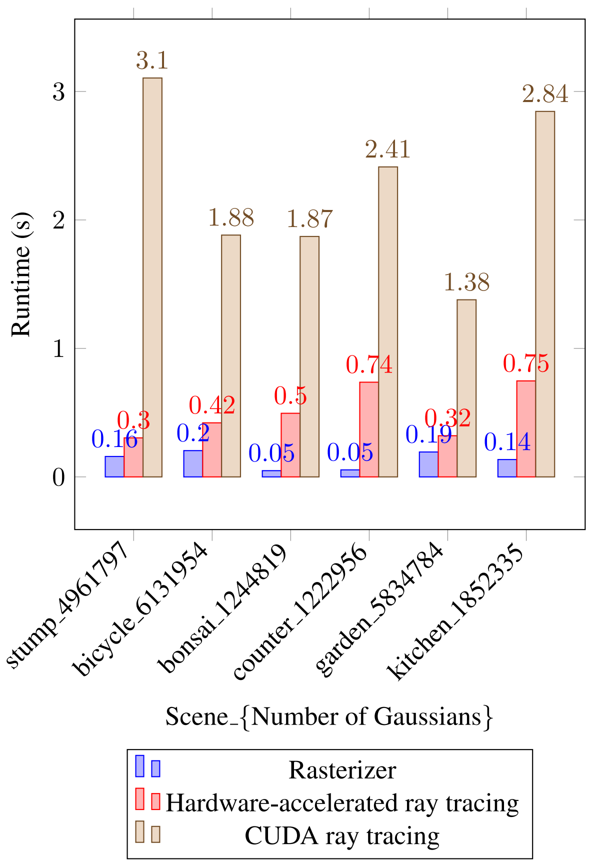

Figure 3. Runtime performance of rendering methods for various scenes

mance gap between rasterizer and hardware-accelerated ray tracing shrinks and the performance gap is approximately $\sim 2\times$. However, for smaller scenes, we see a performance gap of up to $\sim 10\times$. These results are consistent with the findings from Figure 2, since the rasterizer has a higher linear constant scaling factor in the number of Gaussians in a scene compared to the rasterizer, so the performance gap between the two methods close as the scenes contain more gaussians. Additionally, it is expected and consistent with the findings from Figure 1, that the rasterizer performs better in default configurations due to the large number of pixels being rendered of approximately 1 million pixels.

4.2. Optimization Evaluation

We evaluate the capability of our renderer in optimizing Gaussian splat scene representations through gradient descent. Motivation for retraining the Gaussian splat scene representation using our renderer, rather than simply using the pretrained scenes, can be found in Figure 4. In this figure, we show views of a *pretrained* model (provided by [5]) of the *stump* scene, rendered using our renderer, and the baseline 3DGS rasterizer renderer. We provide the ground

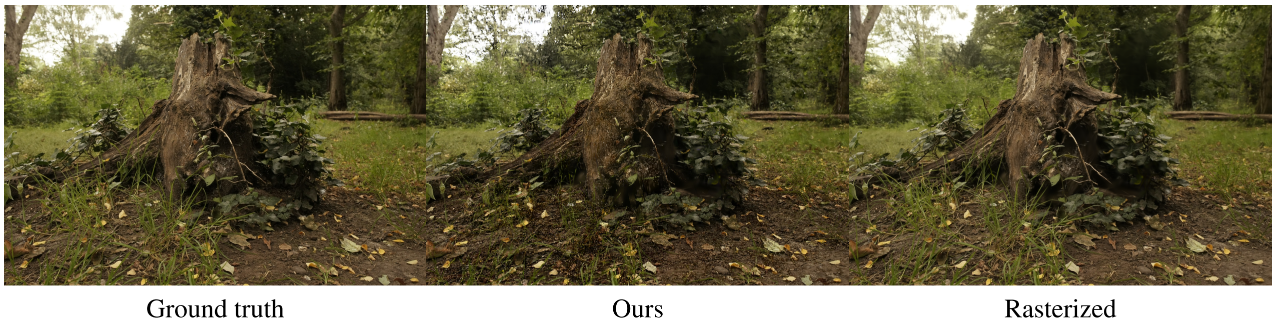

Figure 4. Renders of the *stump* scene using the *pretrained* Gaussian splat scene from 3DGS [5]. We show the ground-truth view, a rendering from the same view using our renderer, and a rendering from the same view using the baseline 3DGS rasterizer.

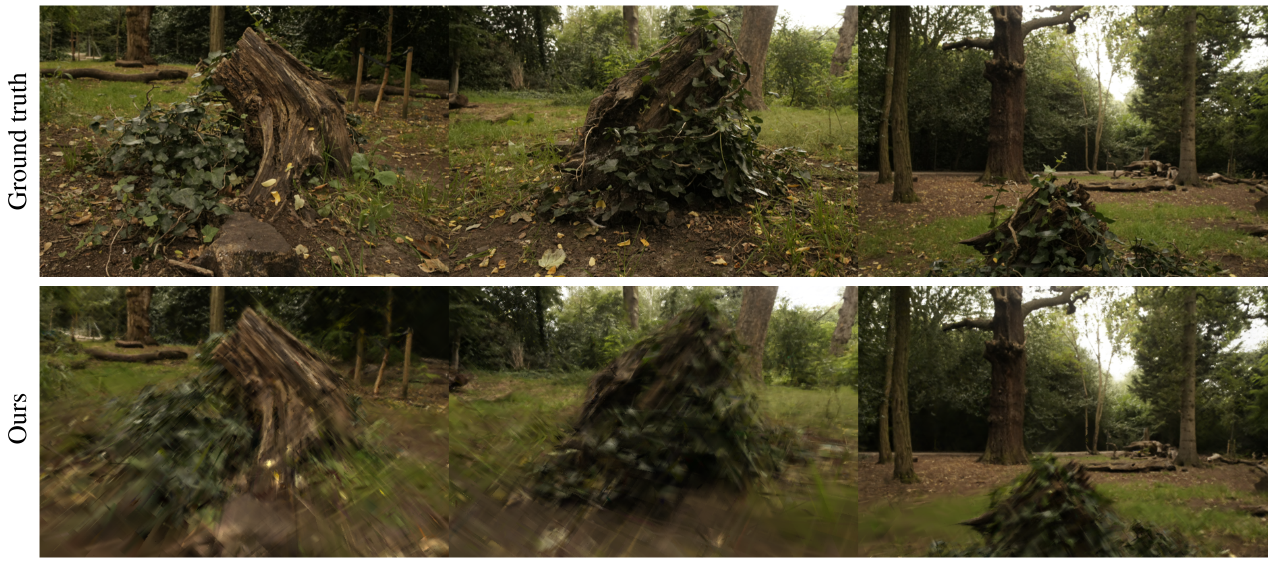

Figure 5. Test views of *stump* scene optimized using our renderer, compared to ground truth.

truth view for comparison. We can see that our result, while mostly reasonable, exhibits more visual artifacts than the rasterized result—in particular, notice that the grass near the foot of stump exhibits artifacts. The fact that the rasterized image is of higher visual quality is unsurprising, since the pretrained model was optimized using the rasterizer. Therefore, our hope is that by training the scene using our renderer instead, we can achieve higher visual quality, comparable to the rasterized result.

To this end, we implement the corresponding backward pass to our renderer, and use our renderer to retrain scenes from Mip-NeRF360 from scratch using gradient descent. For our experiments, we train each scene for 30k epochs on a NVIDIA GeForce RTX 3090 GPU, using the default hyperparameters of [5].

In Figures 5, 6, and 7, we show example qualitative results of training with our renderer on the scenes *stump* and *bonsai* respectively. For each scene, we show several ground truth test views alongside rendered results of our optimized scene from the same views. We note that qualitatively, our results look reasonable and show a preliminary feasibility of this approach, but exhibit some limitations. Most noticeably, our results appear "farsighted," with objects in the background appearing high-fidelity, but objects closer to the camera appearing blurrier. In particular, note that in Figure 6, we are able to reconstruct the spokes of the bike wheel, but that the bonsai tree flowers remain blurry. The limitations in visual quality are also borne out by the PSNR metrics shown in Table 1, where our method unfortunately underperforms compared the rasterizer-trained baseline.

We are not entirely sure what causes this "farsighted" effect, but it may be due to suboptimal choice of hyperparameters or other elements of the optimization strategy. We are optimistic that it is indeed possible to achieve higher visual quality by finetuning and improving the optimization strategy, for two reasons. First, our results, while not perfect, are at least reasonable, and show improvement over the

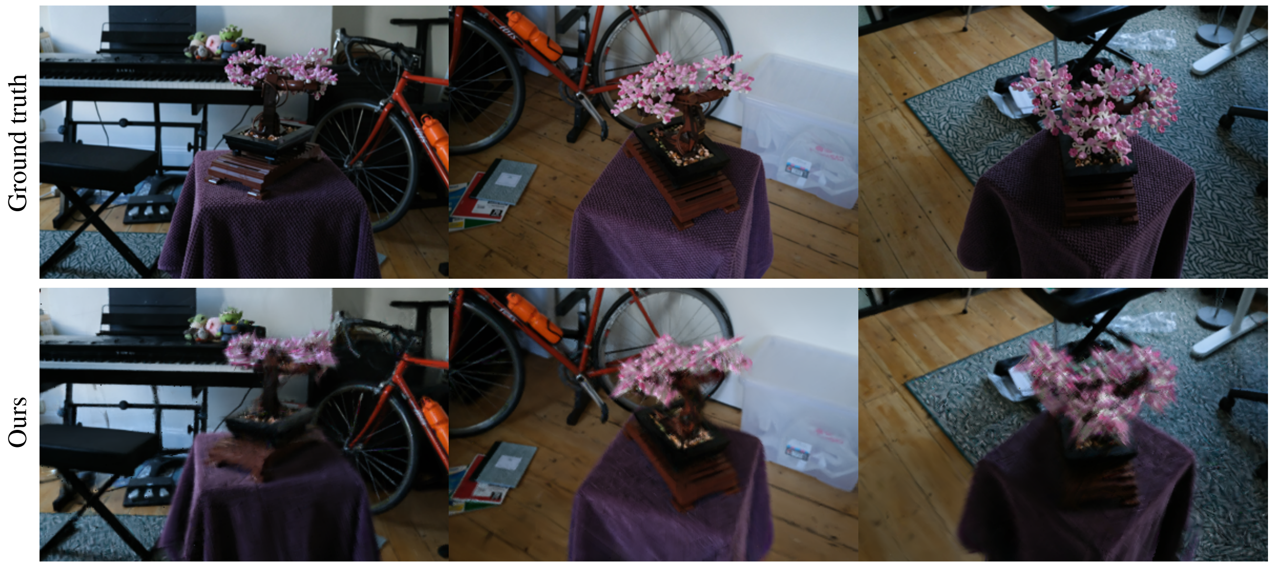

Figure 6. Test views of *bonsai* scene optimized using our renderer, compared to ground truth.

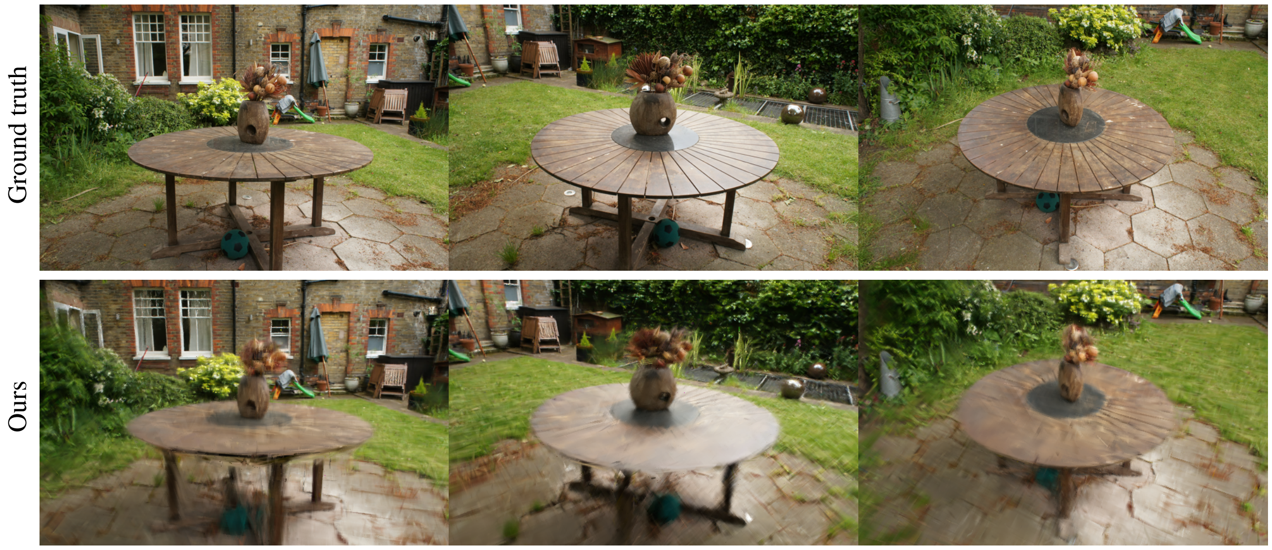

Figure 7. Test views of *garden* scene optimized using our renderer, compared to ground truth.

training epochs, showing that at least some optimization is possible. Second, as shown in Figure 4, our renderings of the pretrained scene, while exhibiting some artifacts, do not exhibit the same noticeable blurriness issue that we observe when retraining from scratch. This points in the direction that a Gaussian splat scene representation without this blurriness issue when seen by our renderer exists, and that it is just up to improving the optimization strategy to be able to find it.

We make an additional note about training time. We observe that our training time is roughly $\sim$5x the training time using the rasterizer, and that most of this is attributable to the building of the BVH, which is done for every iteration.

5. Conclusion

This work demonstrates the feasibility of employing ray tracing for rendering 3D Gaussian splats, displaying advantages over traditional rasterization under specific conditions. Our analysis quantifies performance metrics across varying pixel counts and Gaussian densities in complex scenes. The results indicate that ray tracing exhibits linear performance scaling with pixel count and outperforms rasterization particularly in high-complexity scenes with a large number of Gaussians but fewer pixels. We also

	PSNR (↑)	
Scene	Ours	Rasterized
stump	21.69	25.99
bonsai	25.85	31.98
garden	22.80	27.41
room	22.58	30.63
counter	23.14	28.7
bike	20.45	25.25
Average	22.75	28.33

Table 1. PSNR for our optimized scenes compared the pretrained 3DGS scenes optimized with rasterizer.

demonstrate the preliminary feasibility of using our renderer to train Gaussian splat 3D scene representations from scratch. We expect that as GPU technologies advance, the efficiency and applicability of ray tracing for rendering 3D Gaussian splats will continue to improve, making it a compelling alternative for rendering.

5.1. Future Work

There remain opportunities to investigate to further improve ray-tracing performance of rendering Gaussian splats. Newer hardware aimed towards achieving real-time ray tracing performance in games, which require tracing billions of rays per second, continue to improve ray-tracing support; we expect that these hardware improvements will readily translate to hardware-accelerated ray-tracing performance for Gaussian splats. We also note that the need to sort Gaussians in depth order removes opportunities to parallelize ray intersections for a single ray and the need for iterative closest hit queries increases runtime significantly. To gain more ray query parallelism, one technique worth exploring is to split a ray into disjoint segments and query each segment in parallel. It may also be worth re-visiting using the *any hit* shader with Optix 8.0 to collect all intersections and sort with radix sorting.

For optimization of Gaussian splats using ray tracing, improving the currently shown limitations on visual quality, possibly by experimenting with optimization strategy and hyperparameter tuning, is an important direction for future work. Another interesting avenue for future work is improving the training time. Currently, the BVH is rebuilt at every iteration, and this comprises the majority of the training time. Future work might explore methods for training where the BVH only needs to be rebuilt every so many iterations, or where it is optimized directly.

6. Team Responsibilities

Alex Lin: CUDA ray tracer and hardware-accelerated ray tracer's BVH + ray-gaussian intersection implementation; performance runtime experiments.

Meijin Li: alpha compositing for 2D and 3D Gaussians respectively (effort but unresolved in 3D); use GPU radix sort and shared memory to facilitate rendering.

Yvette Lin: Implemented backward pass for renderer and performed training/optimization experiments.

References

[1] J. Barron, B. Mildenhall, D. Verbin, P. Srinivasan, and P. Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields, 11 2021.

[2] P. Dai, J. Xu, W. Xie, X. Liu, H. Wang, and W. Xu. High-quality surface reconstruction using gaussian surfels. *arXiv preprint arXiv:2404.17774*, 2024.

[3] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz. Multi-view stereo for community photo collections. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE, 2007.

[4] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. pages 33–37, 06 2012.

[5] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[6] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*, pages 347–353. 1998.

[7] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[8] Z. Qian, S. Wang, M. Mihajlovic, A. Geiger, and S. Tang. 3dgs-avatar: Animatable avatars via deformable 3d gaussian splatting. *arXiv preprint arXiv:2312.09228*, 2023.

[9] H. Rhodin, N. Robertini, C. Richardt, H.-P. Seidel, and C. Theobalt. A versatile scene model with differentiable visibility applied to generative pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 765–773, 2015.

[10] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3d. In *ACM siggraph 2006 papers*, pages 835–846. 2006.

[11] A. Wang, P. Wang, J. Sun, A. Kortylewski, and A. Yuille. Voge: a differentiable volume renderer using gaussian ellipsoids for analysis-by-synthesis. *arXiv preprint arXiv:2205.15401*, 2022.

[12] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990.

[13] Z. Yu, A. Chen, B. Huang, T. Sattler, and A. Geiger. Mip-splatting: Alias-free 3d gaussian splatting. *arXiv preprint arXiv:2311.16493*, 2023.