3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes

NICOLAS MOENNE-LOCCOZ*, NVIDIA, Canada
ASHKAN MIRZAEI*, NVIDIA, Canada and University of Toronto, Canada
OR PEREL, NVIDIA, Israel
RICCARDO DE LUTIO, NVIDIA, USA
JANICK MARTINEZ ESTURO, NVIDIA, Germany
GAVRIEL STATE, NVIDIA, Canada
SANJA FIDLER, NVIDIA, Canada, University of Toronto, Canada, and Vector Institute, Canada
NICHOLAS SHARP†, NVIDIA, USA
ZAN GOJCIC†, NVIDIA, Switzerland

Fig. 1. We propose a method for fast forward and inverse ray tracing of particle-based scene representations such as Gaussians. The main idea is to construct encapsulating primitives around each particle, and insert them into a BVH to be rendered by a ray tracer specially adapted to the high density of overlapping particles. Efficient ray tracing opens the door to many advanced techniques, including secondary ray effects like mirrors, refractions and shadows, as well as highly-distorted cameras with rolling shutter effects and even stochastic sampling of rays. **Project page: GaussianTracer.github.io**

Particle-based representations of radiance fields such as 3D Gaussian Splatting have found great success for reconstructing and re-rendering of complex scenes. Most existing methods render particles via rasterization, projecting them to screen space tiles for processing in a sorted order. This work instead considers ray tracing the particles, building a bounding volume hierarchy and casting a ray for each pixel using high-performance GPU ray tracing hardware. To efficiently handle large numbers of semi-transparent particles, we describe a specialized rendering algorithm which encapsulates particles with bounding meshes to leverage fast ray-triangle intersections, and shades batches of intersections in depth-order. The benefits of ray tracing are well-known in computer graphics: processing incoherent rays for secondary lighting effects such as shadows and reflections, rendering from highly-distorted cameras common in robotics, stochastically sampling rays, and more. With our renderer, this flexibility comes at little cost compared to rasterization. Experiments demonstrate the speed and accuracy of our approach, as well as several applications in computer graphics and vision. We further propose related improvements to the basic Gaussian representation, including a simple use of generalized kernel functions which significantly reduces particle hit counts.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Reconstruction**.

Additional Key Words and Phrases: Radiance Fields, Gaussian Splats, Ray Tracing

*Authors contributed equally.
†Authors contributed equally.

Authors' addresses: Nicolas Moenne-Loccoz, NVIDIA, Montreal, Canada, nicolasm@nvidia.com; Ashkan Mirzaei, NVIDIA, Toronto, Canada and University of Toronto, Toronto, Canada, ashkan@cs.toronto.edu; Or Perel, NVIDIA, Tel Aviv, Israel, operel@nvidia.com; Riccardo de Lutio, NVIDIA, Santa Clara, USA, rdelutio@nvidia.com; Janick Martinez Esturo, NVIDIA, Munich, Germany, janickm@nvidia.com; Gavriel State, NVIDIA, Toronto, Canada, gstate@nvidia.com; Sanja Fidler, NVIDIA, Toronto, Canada and University of Toronto, Toronto, Canada and Vector Institute, Toronto, Canada, sfidler@nvidia.com; Nicholas Sharp, NVIDIA, Seattle, USA, nsharp@nvidia.com; Zan Gojcic, NVIDIA, Zürich, Switzerland, zgojcic@nvidia.com.

1 INTRODUCTION

Multiview 3D reconstruction and novel-view synthesis are a classic challenge in visual computing, key to applications across robotics, telepresence, AR/VR, and beyond. Many approaches have been proposed, but most recently particle-based representations have shown incredible success, ignited by 3D Gaussian Splatting [Kerbl et al. 2023] (3DGS)—the basic idea is to represent a scene as a large unstructured collection of fuzzy particles which can be differentiably rendered by splatting to a camera view with a tile-based rasterizer. The location, shape, and appearance of the particles are optimized using a re-rendering loss.

Meanwhile, more broadly in computer graphics, rendering has long been a duality between rasterization and ray tracing. Traditionally, rasterization supported real-time performance at the expense of approximating image formation, while ray tracing enabled fully general high-fidelity rendering in the expensive offline setting. However, the introduction of specialized GPU ray tracing hardware and real-time renderers has moved ray tracing into the realm of real-time performance.

This work is motivated by the observation that 3DGS is limited by the classic tradeoffs of rasterization. The tile-based rasterizer is ill-suited to rendering from highly-distorted cameras with rolling shutter effects, which are important in robotics and simulation. It also cannot efficiently simulate secondary rays needed to handle phenomena like reflection, refraction, and shadows. Likewise, rasterization cannot sample rays stochastically, a common practice for training in computer vision. Indeed, prior work has already encountered the need for these capabilities, but was limited to rendering with restrictive tricks or workarounds [Niemeyer et al. 2024; Seiskari et al. 2024]. We instead aim to address these limitations by making the ray traced Gaussian particles efficient, with a tailored implementation for specialized GPU ray tracing. To be clear, goal of this work is not to offer an end-to-end solution to unsolved problems like global illumination or inverse lighting on particle scenes, but rather to provide a key algorithmic ingredient to future research on these problems: a fast differentiable ray tracer.

Efficiently ray tracing Gaussian scenes (and more generally semi-transparent surfaces) is not a solved problem ["Tanki" Zhang 2021]. We find that even past algorithms that were specially designed for ray tracing semi-transparent particles [Brüll and Grosch 2020; Knoll et al. 2019; Münstermann et al. 2018] are ineffective on these scene reconstructions, due to the huge numbers of non-uniformly distributed and densely-overlapping particles. Accordingly, we design a customized GPU-accelerated ray tracer for Gaussian particles with a k-buffer [Bavoil et al. 2007] hits-based marching to gather ordered intersections, bounding mesh proxies to leverage fast ray-triangle intersections, and a backward pass to enable optimization. Each of these components is carefully tested for speed and quality on a variety of benchmarks. We found it crucial to tune the details of the algorithm to the task at hand. Our final proposed algorithm is almost 25x faster than our first naive implementation, due to a wide range of algorithmic and numerical factors. We hope that these learnings will be of value to the community leveraging raytracing on particle representations.

The fundamental approach of representing a scene with particles is not limited to the Gaussian kernel; and recent work has already shown several natural generalizations [Huang et al. 2024]. Our ray tracing scheme, as well as the benefits and applications above, likewise generalizes more broadly to particle-based scene representations, as we show in section Section 4.5.

We evaluate this approach on a wide variety of benchmarks and applications. On standard multiview benchmarks, ray tracing nearly matches or exceeds the quality of the 3DGS rasterizer of Kerbl et al. [2023], while still achieving real-time rendering framerates. More importantly, we demonstrate a variety of new techniques made easy and efficient by ray tracing, including secondary ray effects like shadows and reflections, rendering from cameras with high distortion and rolling shutter, training with stochastically sampled rays and more.

In summary, the contributions of this work are:

- A GPU-accelerated ray tracing algorithm for semi-transparent particles.
- An improved optimization pipeline for ray-traced, particle-based radiance fields.
- Generalized Gaussian particle formulations that reduce the number of intersections and lead to improved rendering efficiency.
- Applications demonstrating the utility of ray tracing, including: depth of field, shadows, mirrors, highly-distorted cameras, rolling shutter, incoherent rays, and instancing.

2 RELATED WORK

2.1 Novel-View Synthesis and Neural Radiance Fields

Classical approaches to novel-view synthesis can be roughly categorized based on the sparsity of the input views. In the case of sparse views, most methods first construct a proxy geometry using multi-view stereo [Schönberger and Frahm 2016; Schönberger et al. 2016] and point cloud reconstruction methods [Kazhdan et al. 2006; Kazhdan and Hoppe 2013] and then unproject the images onto this geometry either directly in terms of RGB colors [Buehler et al. 2001; Debevec et al. 1996; Wood et al. 2000] or extracted latent features [Riegler and Koltun 2020, 2021]. The novel views are rendered by projecting the color or features from the geometry to the camera plane. In the case of densely sampled input views, the novel-view synthesis can instead be formulated directly as light field interpolation problem [Davis et al. 2012; Gortler et al. 1996; Levoy and Hanrahan 1996].

Neural Radiance Fields (NeRFs) [Mildenhall et al. 2020] have revolutionized the field of novel-view synthesis by representing the scenes in terms of a volumetric radiance field encoded in a coordinate-based neural network. This network can be queried at any location to return the volumetric density and view-dependent color. The photo-realistic quality of NeRFs has made them the standard representation for novel-view synthesis. Follow-up works have focused on improving the speed [Müller et al. 2022; Reiser et al. 2021], quality [Barron et al. 2021, 2022, 2023], and surface representation [Li et al. 2023; Wang et al. 2021, 2023a; Yariv et al. 2021]. NeRF has also been extended to large-scale scenes [Li et al. 2024a; Turki

Fig. 2. **Runtime Ray Tracing Effects:** Our ray-based pipeline is easily compatible with conventional ray-based visual effects at test time, including reflections (top left), depth of field (top middle), refractions (bottom left), hard shadows cast by meshes (bottom middle), and myriad combinations of them (right).

et al. 2022], sparse inputs views [Niemeyer et al. 2022], in-the-wild image collections [Martin-Brualla et al. 2021], and reflections [Guo et al. 2022]. Finally, several works investigated ways to speed up the inference by baking the neural fields to more performant representations [Duckworth et al. 2023; Reiser et al. 2024, 2023; Wang et al. 2023b]. While achieving high quality and fast rendering speeds, these methods often employ compute-expensive multi-stage training procedures.

2.2 Point-Based and Particle Rasterization

Grossman and Dally [1998] defined point-based rendering as a simple rasterization of object surface points along with their color and normals. However, due to the infinitesimal size of the points, such simple rasterization inevitably led to holes and aliasing. To address these limitations, later work converted points to particles with a spatial extent, such as surfels, circular discs, or ellipsoids [Pfister et al. 2000; Ren et al. 2002; Zwicker et al. 2001]. More recently, points or particles have also been augmented with neural features and rendered using rasterization in combination with CNN networks [Aliev et al. 2020; Kopanas et al. 2021; Rückert et al. 2022] or NeRF-like volumetric rendering [Ost et al. 2022; Xu et al. 2022].

Differentiable rendering through alpha blending was also extended to volumetric particles. Pulsar [Lassner and Zollhöfer 2021] proposed an efficient sphere-based differentiable rasterization approach, which allows for real-time optimization of scenes with millions of particles. The seminal 3DGS work of Kerbl et al. [2023] instead represented the scenes using fuzzy, anisotropic 3D Gaussian particles. By optimizing the shape, position, and appearance of these Gaussian particles through an efficient tile-based rasterizer, 3DGS achieves SoTA results in terms of perceptual quality and efficiency. 3DGS inspired many follow-up works that aim to reduce the render time or memory footprint [Fan et al. 2023; Niedermayr et al. 2023; Papantonakis et al. 2024], improve surface representation [Guédon and Lepetit 2023; Huang et al. 2024], and support large-scale scenes [Kerbl et al. 2024; Ren et al. 2024], and more.

Jointly, these works have made significant progress, but they still inherit limitations of rasterization. Indeed, they are not able to represent highly distorted cameras, model secondary lighting effects, or simulate sensor properties such as rolling shutter or motion blur. Several works have tried to work around these limitations. Niemeyer et al. [2024] first train a Zip-NeRF [Barron et al. 2023] that can model distorted and rolling shutter cameras and then render perfect pinhole cameras from the neural field and distill them into a 3DGS representation. This allows for fast inference, but is still limited to perfect pinhole cameras. To address secondary lighting effects, recent works bake occlusion information into spherical harmonics at each Gaussian [Gao et al. 2023; Liang et al. 2023] or leverage shading models and environment maps [Jiang et al. 2024]. The latter two of these render only with rasterization; in contrast Gao et al. [2023] traces rays for initial visibility determination, but uses only a visibilty forward pass, and restricts ray tracing to the training phase, relying on rasterization during inference and inheriting its limitations. In contrast, our method uses optimized ray-tracing as the sole renderer throughout both training and inference, which allows for inserting objects, refraction, lens distortion, and other complex effects. Additionally, Gao et al. [2023] use axis-aligned bounding boxes (AABBs) to enclose particles, which results in approximately 3× lower FPS during inference compared to the stretched icosahedrons employed in our optimized tracer (Section 5.2). Finally, for complex lens effects, Seiskari et al. [2024] model the motion blur and rolling shutter of the camera by approximating them in screen space through rasterization with pixel velocities. Unlike these works, we formulate a principled method for efficient ray tracing of volumetric particles, which natively alleviates all the limitations mentioned above and further allows us to simulate effects such as depth of field and perfect mirrors.

2.3 Differentiable Ray Tracing of Volumetric Particles

Ray tracing became the gold standard for offline rendering of high-quality photo-realistic images [Whitted 1979]. The advent of dedicated hardware to efficiently compute the intersection of camera

rays with the scene geometry has also enabled its use for real-time rendering applications such as gaming and the simulation industry. Modern GPUs are exposing ray tracing rendering pipelines, from the computation of dedicated acceleration structures to a specific programmable interface [Joshi et al. 2007].

However, these works are highly optimized for rendering opaque surfaces and efficiently handling order independent semi-transparent surfaces or particles remains challenging ["Tanki" Zhang 2021].

A first class of works [Aizenshtein et al. 2022; Münstermann et al. 2018] proposes to first estimate the transmittance function along the ray and subsequently to integrate the particles' radiance based on this estimate. It assumes the traversal of the full scene to be fast enough; an assumption that does not hold in Gaussian particles for scene reconstruction.

A second class of works consists in re-ordering the particles along the ray. Knoll et al. [2019] propose a slab-tracing method to trace semi-transparent volumetric RBF (radial basis function) particles, which enables real-time ray tracing of scenes consisting of several millions of such particles. However, its efficiency is largely based on the assumption of the isotropic shape of the particles and a high level of uniformity in their spatial distribution. In [Brüll and Grosch 2020], the multi-layer alpha blending approach from [Salvi and Vaidyanathan 2014] is extended to ray tracing. Their multi-layer alpha tracing supports efficient rendering of any semi-transparent surface but its approximation of the particle's ordering may produce rendering artifacts.

Our formulation takes root in these precursory works. However as opposed to [Knoll et al. 2019], it is guaranteed to process every particle intersecting the ray, and contrary to [Brüll and Grosch 2020] the hit processing order is consistent, which ensures the differentiability of our tracing algorithm.

Compared to rasterization, differentiable ray tracing of semi-transparent particles has seen much less progress in recent years. Perhaps the most similar rendering formulation to ours was proposed in Fuzzy Metaballs [Keselman and Hebert 2022, 2023], but it is limited to scenes with a small set of 3D Gaussian particles (several tens) and images with very low resolution. Different to Fuzzy Metaballs, our method can easily handle scenes with several millions of particles from which it can render full HD images in real-time. In another direction, Belcour et al. [2013] incorporate defocus and motion blur in to ray tracers by leveraging Gaussian approximations as a sampling technique, rather than a scene representation as used here.

3 BACKGROUND

We begin with a short review of 3D Gaussian scene representation, volumetric particle rendering, and hardware-accelerated ray tracing.

3.1 3D Gaussian Parameterization

Extending Kerbl et al. [2023], our scenes can be represented as a set of differentiable semi-transparent particles defined by their kernel function. For example, the kernel function of a 3D Gaussian particle $\rho : \mathbb{R}^3 \to \mathbb{R}$ at a given point $\boldsymbol{x} \in \mathbb{R}^3$ can be expressed as

$$\rho(\boldsymbol{x}) = e^{-(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}, \tag{1}$$

where $\boldsymbol{\mu} \in \mathbb{R}^3$ represents the particle's position and $\Sigma \in \mathbb{R}^{3\times3}$ the covariance matrix. To ensure the positive semi-definiteness of the covariance matrix Σ when optimizing it using gradient descent, we represent it as

$$\Sigma = \boldsymbol{R}\boldsymbol{S}\boldsymbol{S}^T\boldsymbol{R}^T \tag{2}$$

with $\boldsymbol{R} \in \mathrm{SO}(3)$ a rotation matrix and $\boldsymbol{S} \in \mathbb{R}^{3\times3}$ a scaling matrix. These are both stored as their equivalent vector representations, a quaternion $\boldsymbol{q} \in \mathbb{R}^4$ for the rotation and a vector $\boldsymbol{s} \in \mathbb{R}^3$ for the scale. For other particle variants explored in this work, please refer to Section 4.5.

Each particle is further associated with an opacity coefficient $\sigma \in \mathbb{R}$, and a parametric radiance function $\phi_{\boldsymbol{\beta}}(\boldsymbol{d}) : \mathbb{R}^3 \to \mathbb{R}^3$, dependent on the view direction $\boldsymbol{d} \in \mathbb{R}^3$. In practice, following Kerbl et al. [2023], we use a spherical harmonics functions $Y_\ell^m(\boldsymbol{d})$ of order $m = 3$ defined by the coefficients $\boldsymbol{\beta} \in \mathbb{R}^{48}$. Note that we are using the ray direction while Kerbl et al. [2023] uses $\frac{\boldsymbol{\mu}-\boldsymbol{o}}{\|\boldsymbol{\mu}-\boldsymbol{o}\|}$ for performance reason.

Therefore the parametric radiance function can be written as

$$\phi_{\boldsymbol{\beta}}(\boldsymbol{d}) = f\left(\sum_{\ell=0}^{\ell_{\max}} \sum_{m=-\ell}^{\ell} \beta_\ell^m Y_\ell^m(\boldsymbol{d})\right) \tag{3}$$

where f is the sigmoid function to normalize the colors.

3.2 Differentiable Rendering of Particle Representations

Given this parametrization, the scene can be rendered along a ray $\boldsymbol{r}(\tau) = \boldsymbol{o} + \tau\boldsymbol{d}$ with origin $\boldsymbol{o} \in \mathbb{R}^3$ and direction $\boldsymbol{d} \in \mathbb{R}^3$ via classical volume rendering

$$\boldsymbol{L}(\boldsymbol{o},\boldsymbol{d}) = \int_{\tau_n}^{\tau_f} T(\boldsymbol{o},\boldsymbol{d}) \left(\textstyle\sum_i \left(1 - e^{-\sigma_i \rho_i(\boldsymbol{o}+\tau\boldsymbol{d})}\right)\boldsymbol{c}_i(\boldsymbol{d})\right) d\tau, \tag{4}$$

where $\boldsymbol{c}_i(\boldsymbol{d}) = \phi_{\boldsymbol{\beta}_i}(\boldsymbol{d})$ is the color of the i^{th} Gaussian obtained by evaluating its view-dependant radiance function. The transmittance function $T(\boldsymbol{o},\boldsymbol{d})$ is defined as

$$T(\boldsymbol{o},\boldsymbol{d}) = e^{-\int_{\tau_n}^{\tau} \sum_i \sigma_i \rho_i(\boldsymbol{o}+t\boldsymbol{d}) dt}. \tag{5}$$

Considering $\alpha_i = \sigma_i \rho_i(\mathbf{x}_i)$, Equation 4 can be approximated using numerical integration as

$$\boldsymbol{L}(\boldsymbol{o},\boldsymbol{d}) = \sum_{i=1}^{N} \boldsymbol{c}_i(\boldsymbol{d}) \alpha_i \prod_{j=1}^{i-1} 1 - \alpha_j, \tag{6}$$

where in this linear approximation, $\mathbf{x}_i$ is defined as the point along the ray $\boldsymbol{r}$ with the highest response ρ_i of the i^{th} Gaussian (see 4.3 for more details). For derivations of the higher order approximations of ρ_i please refer to [Keselman and Hebert 2022].

3.3 Hardware-Accelerated Ray Tracing

In this work we use NVIDIA RTX hardware through the NVIDIA OptiX programming interface [Parker et al. 2010]. Through this interface, geometric primitives such as triangle meshes are processed to construct a *Bounding Volume Hierarchy* (BVH[1]). This acceleration

[1]For clarity, throughout this work we reference BVH as the de-facto hardware acceleration structure. However, since in practice NVIDIA OptiX's specification interfaces the implementation of bottom level acceleration structures, we emphasize our pipeline does not depend on a particular implementation.

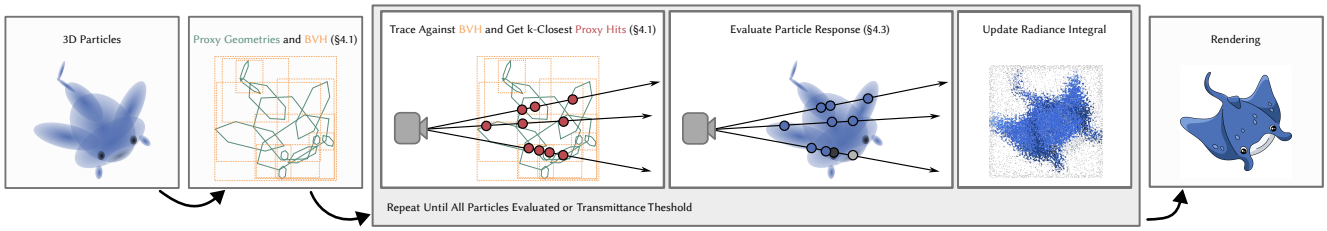


Fig. 3. **Overview of the Accelerated Tracing Algorithm:** Given a set of 3D particles, we first build the corresponding bounding primitives and insert them into a BVH. To compute the incoming radiance along each ray, we trace rays against the BVH to get the next k particles. We then compute the intersected particles' response and accumulate the radiance according to Equation 6. The process repeats until all particles have been evaluated or the transmittance meets a predefined threshold and the final rendering is returned.

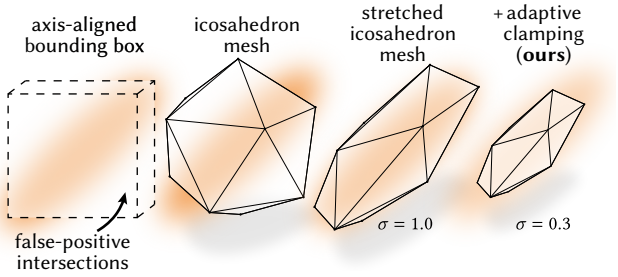


Fig. 4. **Proxy Geometries:** Examples of BVH primitives considered.

structure is optimized for the computation of ray-primitive intersections by dedicated hardware, the RT cores. The programmable pipeline sends traversal queries to this hardware, freeing the GPU streaming-multiprocessors (SMs) for the main computational load, e.g material shading. The interactions of the SMs with the RT cores are done through the following programmable entry points:

- *ray-gen* program (ray generation) is where the SMs may initiate a scene traversal for a given ray.
- *intersection* program is called during the traversal to compute the precise intersection with potentially hit primitives that are not directly supported by the hardware.
- *any-hit* program is called during the traversal for every hit and may further process or reject the hit.
- *closest-hit* program is called at the end of the traversal, for further processing of the closest accepted hit.
- *miss* program is called at the end of the traversal for further processing when no hit has been accepted.

Such a pipeline is highly optimized to render opaque primitives, i.e. the number of expected hits during a traversal is low, with a minimal amount of interactions between the SMs and the RT cores. Rendering volumes, where the primitives are semi-transparent, requires traversing and processing many hits per ray. To efficiently trace a volume, specific approaches must be designed, tailored to the type of primitives (or particles), their expected size, and distribution across the volume (see for example [Knoll et al. 2019]). In this work we propose an efficient and differentiable algorithm to ray trace a volume made of optimized semi-transparent particles for high-fidelity novel view rendering.

4 METHOD

The proposed volumetric particle tracer requires two core components: a strategy to represent particles in an acceleration structure (BVH) to efficiently test for intersections against them, using adaptive bounding mesh primitives (Section 4.1), and a rendering algorithm which casts rays and gathers batches of intersections, efficiently scheduled onto the NVIDIA OptiX tracing model (Section 4.2).

4.1 Bounding Primitives

Any ray-tracer must somehow insert the primitive particles into a BVH and query the primitives intersected by a ray. The first challenge is then to decide how to insert particles into a BVH and conservatively test intersection against them.

The NVIDIA OptiX programming model supports three primitive types which can be inserted into the BVH: triangles, spheres, and custom primitives given by their axis-aligned bounding boxes (AABBs). These options admit many possible strategies to build a BVH over particles, such as constructing naive axis-aligned bounds as AABBs or spheres, or building bounding triangle meshes. These strategies have a tradeoff between the cost to test intersection *vs.* the tightness of the bound. For instance, simply intersecting a ray with the AABB around each particle is fast, but a diagonally-stretched Gaussian particles will cause the traversal to have to evaluate many false-positive intersections which actually contribute almost nothing to the rendering. None of these strategies necessarily affect the appearance of the rendered image, but rather the computation speed and number of low-contribution particles needlessly processed. Billboard mesh proxies are used elsewhere [Niedermayr et al. 2023], but do not apply in our general setting where rays may come from any direction.

Stretched Polyhedron Proxy Geometry. After experimenting with many variants (Section 5.2.1), we find it most effective to encapsulate particles in a stretched regular icosahedron mesh (Figure 4), which tightly bounds the particle and benefits from hardware-optimized ray-triangle intersections. A hit against any front-facing triangle of the bounding mesh triggers processing of the corresponding particle, as described later in Section 4.3. We fit the bounding proxy by specifying a minimum response α_{min} which must be captured (typically $\alpha_{\text{min}} = 0.01$), and analytically compute an anisotropic rescaling of the icosahedron to cover the whole space with at least

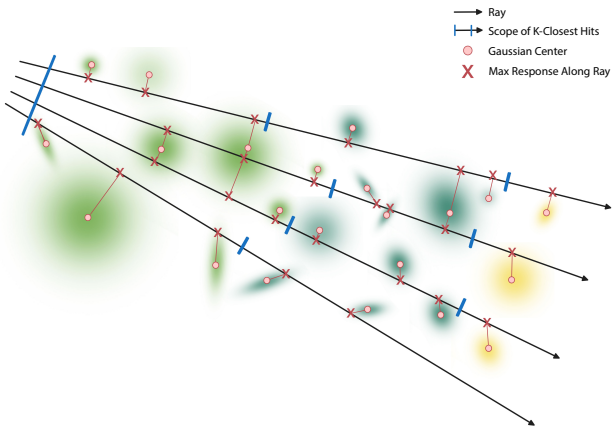


Fig. 5. **Rendering:** on each round of tracing, the next k closest hit particles are collected and sorted into depth order along the ray, the radiance is computed in-order, and the ray is cast again to collect the next hits.

$\alpha_{\min}$ response. Precisely, for each particle we construct an icosahedron with a unit inner-sphere, and transform each canonical vertex $\boldsymbol{v}$ according to:

$$\boldsymbol{v} \leftarrow \sqrt{2\log(\sigma/\alpha_{\min})}\boldsymbol{S}\boldsymbol{R}^T\boldsymbol{v} + \boldsymbol{\mu}. \tag{7}$$

Importantly, this scaling incorporates the opacity of the particles, so that large nearly-transparent particles may have smaller bounding primitives, resulting in adaptive clamping of the particles.

4.2 Ray Tracing Renderer

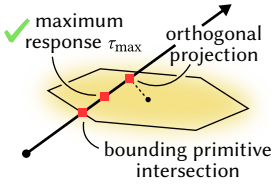
Motivation. Given the ability to cast rays against particles, volumetric rendering as in Equation 6 requires accumulating the contribution of particles along the ray in a sorted order. One naive approach within the NVIDIA OptiX programming model (Section 3.3) is to repeatedly cast the ray, process the nearest particle with a *closest-hit* program, then re-cast the ray to find the next particle. Another is to traverse the scene only twice, once to estimate the transmittance function, and once to the compute the integral as in [Münstermann et al. 2018]. Both of these strategies are prohibitively expensive, due to the cost of traversing the scene.

Our renderer builds on past approaches for tracing semi-transparent surfaces or particles: Knoll et al. [2019] repeated gather slabs of particles and sort within each slab, while Brüll and Grosch [2020] process all semi-transparent surfaces into a k-buffer, merging adjacent particles when the list overflows. As discussed in Section 2.3, because of their approximations, these algorithms do not produce a consistent rendering, which prevents differentiation and generates artifacts.

Algorithm. Figure 5, Figure 3, Procedure 1, and Procedure 2 summarize our approach. To compute incoming radiance along each ray, a *ray-gen* program traces a ray against the BVH to gather the next k particles, using an *any-hit* program to maintain a sorted buffer of their indices. For efficiency, at this stage the particle response is not yet evaluated; all primitive hits are treated as intersected particles. The *ray-gen* program then iterates through the sorted array of primitive hits, retrieves the corresponding particle for each, and renders them according to Equation 6. The process then repeats, tracing a new ray from the last rendered particle to gather the next k particles. The process terminates once all particles intersecting the ray are processed, or early-terminates as soon as enough particle density is intersected to reach a predefined minimum transmittance $T_{\min}$. Compared to past approaches this renderer allows for processing the intersection in a consistent order, without missing any particle nor approximating the transmittance.

Nonetheless, this proposed algorithm is just one of many possible variants, chosen for performance after extensive benchmarking. See Section 5.2 for timings and ablations against a selection of alternatives considered; we find that subtle changes to the algorithm have a dramatic effect of speed and quality on densely-clustered multi-view scenes.

4.3 Evaluating Particle Response



After identifying ray-particle intersections, we must choose how to compute the contribution of each particle to the ray. As with prior work, we take a single sample per particle, but we still must choose at what distance τ along the ray to evaluate that sample. Knoll et al. [2019] orthogonally project the center of the particle on to the ray; this strategy is reasonable for isotropic particles, but can lead to significant error for stretched anisotropic particles. Instead, we analytically compute a sample location $\tau_{\max} = \operatorname{argmax}_\tau \rho(\boldsymbol{o} + \tau\boldsymbol{d})$, the point of maximum response from the particle along the ray. For Gaussian particles, this becomes

$$\tau_{\max} = \frac{(\boldsymbol{\mu} - \boldsymbol{o})^T\Sigma^{-1}\boldsymbol{d}}{\boldsymbol{d}^T\Sigma^{-1}\boldsymbol{d}} = \frac{-\boldsymbol{o}_g^T\boldsymbol{d}_g}{\boldsymbol{d}_g^T\boldsymbol{d}_g} \tag{8}$$

where $\boldsymbol{o}_g = \boldsymbol{S}^{-1}\boldsymbol{R}^T(\boldsymbol{o} - \boldsymbol{\mu})$ and $\boldsymbol{d}_g = \boldsymbol{S}^{-1}\boldsymbol{R}^T\boldsymbol{d}$.

Note that this strategy incurs a slight approximation in the ordering: the particle hits are integrated in the order of the bounding primitive intersections instead of the order of the sample locations. However, we empirically confirmed that this approximation does not lead to any substantial loss in the quality of the end result.

4.4 Differentiable Ray Tracing and Optimization

Differentiable Rendering. Beyond forward-rendering of particle scenes, our ray tracing renderer is also *differentiable*, to support optimizing particle scenes from observed data. To backpropagate (*i.e.*, reverse-mode differentiate) through the renderer with respect to particle parameters, we first perform an ordinary forward-pass render and compute the desired objective functions. Then, in the backward pass we re-cast the same rays to sample the same set of particles in order, computing gradients with respect to each shading expression and accumulating gradients in shared buffers with atomic scatter-add operations. We implemented all derivative expressions by hand in an NVIDIA OptiX *ray-gen* program which is structurally similar to Procedure 1.

Procedure 1 RAY-GEN($\boldsymbol{o}, \boldsymbol{d}, T_{\text{min}}, \alpha_{\text{min}}, k, \tau_{\text{SceneMin}}, \tau_{\text{SceneMax}}$)

Input: ray origin $\boldsymbol{o}$, ray direction $\boldsymbol{d}$, min transmittance T_{min}, min particle opacity α_{min}, hit buffer size k, ray scene bounds τ_{SceneMin} and τ_{SceneMax}
Output: ray incoming radiance $\boldsymbol{L}$, ray transmittance T

$\boldsymbol{L} \leftarrow (0., 0., 0.)$ ▷*radiance*
$T \leftarrow 1.$ ▷*transmittance*
$\tau_{\text{curr}} \leftarrow \tau_{\text{SceneMin}}$ ▷*Minimum distance along the ray*
while $\tau_{\text{curr}} < \tau_{\text{SceneMax}}$ and $T > T_{\text{min}}$ **do**
  ▷*Cast a ray to the BVH for the next k hits, sorted*
  $\mathcal{H} \leftarrow \text{TraceRay}(\boldsymbol{o} + \tau_{\text{curr}}\boldsymbol{d}, \boldsymbol{d}, k)$
  for $(\tau_{\text{hit}}, i_{\text{prim}})$ in $\mathcal{H}$ **do** ▷*Render this batch of hits*
    $i_{\text{particle}} \leftarrow \text{GetParticleIndex}(i_{\text{prim}})$
    $\alpha_{\text{hit}} \leftarrow \text{ComputeResponse}(\boldsymbol{o}, \boldsymbol{d}, i_{\text{particle}})$ ▷$\sigma\rho(\boldsymbol{o} + \tau\boldsymbol{d})$
    if $\alpha_{\text{hit}} > \alpha_{\text{min}}$ **then**
      $\boldsymbol{L}_{\text{hit}} \leftarrow \text{ComputeRadiance}(\boldsymbol{o}, \boldsymbol{d}, i_{\text{particle}})$ ▷*Refer to Equation 3 for SH evaluation*
      $\boldsymbol{L} \leftarrow \boldsymbol{L} + T * \alpha_{\text{hit}} * \boldsymbol{L}_{\text{hit}}$
      $T \leftarrow T * (1 - \alpha_{\text{hit}})$
    $\tau_{\text{curr}} \leftarrow \tau_{\text{hit}}$ ▷*Resume tracing from last hit*
end while
return $\boldsymbol{L}, T$

Procedure 2 ANY-HIT($\tau_{\text{hit}}, i_{\text{prim}}, \mathcal{H}, k$)

Input: hit location τ_{hit}, primitive index i_{prim}, hit array $\mathcal{H}$, hit buffer size k
Output: the hit array $\mathcal{H}$ may be updated in-place with a new entry

h $\leftarrow (\tau_{\text{hit}}, i_{\text{prim}})$
for i in $0 \ldots k$-1 **do** ▷*insertion sort into hit array*
  if $h.\tau_{\text{hit}} < \mathcal{H}[i].\tau_{\text{hit}}$ **then**
    swap($\mathcal{H}[i], h$)
▷*ignore k-closest hits to prevent the traversal from stopping*
if $\tau_{\text{hit}} < \mathcal{H}[k-1].\tau_{\text{hit}}$ **then**
  IgnoreHit()

Optimization. To fit particle scenes using our ray tracer, we adopt the optimization scheme of Kerbl et al. [2023], including pruning, cloning and splitting operations. One significant change is needed: Kerbl et al. [2023] track screen-space gradients of particle positions as a criteria for cloning and splitting, but in our more-general setting, screen space gradients are neither available nor meaningful—instead, we use gradients in 3D world-space for the same purpose. Recent work has proposed many promising extensions to the optimization scheme of Kerbl et al. [2023]. While our ray tracer is generally compatible with any of these extensions, we stay faithful to the original approach for the sake of consistent comparisons. It should also be noted that as particles are updated during optimization, the ray tracing BVH must be regularly rebuilt (see Figure 9, bottom left for BVH build time).

(a) Particle Kernel Functions

(b) Kernelized Surface Particles Reconstruction

Fig. 6. **Particle Kernel Functions:** (a) In addition to 3D Gaussians, in this work we investigated three other particle types: the generalized Gaussian (GG2), kernelized surface (SGG2) and cosine wave modulation (CSGG2) particles. (b) Shows radiance and normal reconstructions obtained with the kernelized surface particles for two scenes.

Training with Incoherent Rays. Optimization in computer vision often benefits from stochastic descent, fitting to randomly-sampled subsets of a problem on each iteration. However, differentiable rasterization can only efficiently render whole images or tiles, and thus efficient stochastic optimization over the set of pixels in a scene is not possible. In our ray tracer, we are free to train with *stochastically-sampled* rays, drawn at random or according to some importance sampling during training, see Section 5.1. Note that when stochastic sampling is used, window-based image objectives such as SSIM cannot be used.

4.5 Particle Kernel Functions

Our formulation does not require the particles to have a Gaussian kernel, enabling the exploration of other particle variants. We consider a general particle defined by its kernel function $\hat{\rho}(\boldsymbol{x})$. In addition to the standard Gaussian, we investigate three other variants, visualized in Figure 6:

- The standard 3D Gaussian kernel given in Equation 1 as

$$\hat{\rho}(\boldsymbol{x}) = \sigma e^{-(\boldsymbol{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})},$$

- Generalized Gaussians of degree n (we use $n = 2$):

$$\hat{\rho}_n(\boldsymbol{x}) = \sigma e^{-((\boldsymbol{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}))^n}. \tag{9}$$

- Kernelized surfaces: 3D Gaussians with a null $\boldsymbol{z}$ scale as in [Huang et al. 2024].
- Cosine wave modulations along axis i:

$$\hat{\rho}_c(\boldsymbol{x}) = \hat{\rho}(\boldsymbol{x})(0.5 + 0.5cos(\psi(\boldsymbol{S}^{-1}\boldsymbol{R}^T(\boldsymbol{x} - \boldsymbol{\mu}))_i)) \tag{10}$$

  with ψ an optimizable parameter.

Comparative evaluations with these particles are presented in Table 4. The generalized Gaussian kernel function defines denser particles, reducing the number of intersections and increasing the performance by a factor of 2 compared to standard Gaussians, see Section 5.2.3 for more discussion. The kernelized surface variant defines flat particles with well-defined normals, which can be encapsulated by a two triangle primitive (Section 4.1) well-adapted to our tracing model. Finally, the modulation by a cosine wave aims to model a particle with spatially varying radiance.

Table 1. Results for our approach and baselines on a variety of novel view synthesis benchmarks.

	MipNeRF360				Tanks & Temples				Deep Blending			
Method\Metric	PSNR↑	SSIM↑	LPIPS↓	Mem. ↓	PSNR↑	SSIM↑	LPIPS↓	Mem. ↑	PSNR↑	SSIM↑	LPIPS↓	Mem. ↓
Plenoxels	23.63	0.670	-	2.1GB	21.08	0.719	-	2.3GB	23.06	0.795	-	2.7GB
INGP-Base	26.43	0.725	-	13MB	21.72	0.723	-	13MB	23.62	0.797	-	13MB
INGP-Big	26.75	0.751	-	48MB	21.92	0.745	-	48MB	24.96	0.817	-	48MB
MipNeRF360	29.23	0.844	-	8.6MB	22.22	0.759	-	8.6MB	29.40	0.901	-	8.6MB
Zip-NeRF	30.38	0.883	0.197	-	-	-	-	-	-	-	-	-
3DGS (paper)	28.69	0.871	-	734MB	23.14	0.853	-	411MB	29.41	0.903	-	676MB
3DGS (checkpoint)	28.83	0.867	0.224	763MB	23.35	0.837	-	422MB	29.43	0.898	-	704MB
Ours (reference)	28.69	0.871	0.220	387MB	23.03	0.853	0.193	519MB	29.89	0.908	0.303	329MB
Ours	28.71	0.854	0.248	383MB	23.20	0.830	0.222	489MB	29.23	0.900	0.315	287MB

5 EXPERIMENTS AND ABLATIONS

In this section we evaluate the proposed approach on several standard benchmark datasets for quality and speed, and perform ablation studies on key design choices in Section 5.2. Additional details on experiments and implementation can be found in the appendix.

Method Variants. In the experiments that follow, we will refer to two variants of our method. The *Ours (reference)* variant corresponds to [Kerbl et al. 2023] as closely as possible. It employs regular 3D Gaussian particles, and leaves the optimization hyperparameters unchanged. We treat this as a high-quality configuration. The *Ours* variant is adapted based on the experiments that follow, improving runtime speed at a slight expense of quality. It uses degree-2 generalized Gaussian particles, a density learning rate to 0.09 during optimizing, as well as optimizing with incoherent rays in a batch size of 2^{19} starting after 15,000 training iterations. Empirically, we find that the larger density learning rate of this model produces denser particles. When combined with the faster fall-off of degree-2 generalized Gaussian particles compared to regular Gaussians, this leads to fewer hits along the rays and faster rendering speeds with minimal quality loss.

5.1 Novel View Synthesis Benchmarks

Baselines. There is a significant amount of recent and ongoing research on scene representation. We include comparisons to several representative well-known methods, including 3DGS [Kerbl et al. 2023], INGP [Müller et al. 2022], and MipNeRF360 [Barron et al. 2022], as a standard for comparison. The latter two are widely-used ray-marching methods that, like this work, do not have the limitations of rasterization. We additionally compare with the non-neural method of Plenoxels [Sara Fridovich-Keil and Alex Yu et al. 2022].

Evaluation Metrics. We evaluate the perceptual quality of the novel-views in terms of peak signal-to-noise ratio (*PSNR*), learned perceptual image patch similarity (*LPIPS*), and structural similarity (*SSIM*) metrics. To evaluate the efficiency, we measure GPU time required for rendering a single image without the overhead of storing the data to a hard drive or visualizing them in a GUI. Specifically, we report the performance numbers in terms of frames-per-second measured on a single NVIDIA RTX 6000 Ada. For all evaluations, we use the dataset-recommended resolution for evaluation.

5.1.1 MipNeRF360. `MipNeRF360` [Barron et al. 2022] is a challenging dataset consisting of large scale outdoor and indoor scenes. In our evaluation, we use the four indoor (`room`, `counter`, `kitchen`, `bonsai`) and three outdoor (`bicycle`, `garden`, `stump`) scenes without licensing issues. In line with prior work, we used images downsampled by a factor two for the indoor and a factor four for the outdoor scenes in all our evaluations.

Table 1 shows quantitative results, while novel-views are qualitatively compared in Figure 7. In terms of quality, our method performs on par or slightly better than 3DGS [Kerbl et al. 2023] and other state-of-the-art methods. For this dataset, we also compare our method against the recent top-tier method of Zip-NeRF [Barron et al. 2023] which achieves 30.38 PSNR. In terms of runtime (Table 2), at 78 FPS our efficient ray tracing implementation is approximately three times slower than rasterization (238 FPS), while maintaining interactive speeds compared to high-quality ray-marching based works MipNeRF360 and Zip-NeRF (<1 FPS). Zip-NeRF employs multisampling to approximate conical frustums in the ray-casting process, combining MipNeRF360's anti-aliasing techniques with the speedup mechanism of INGP. While achieving unprecedented rendering quality, Zip-NeRF does not support real-time rendering (< 1 FPS).

5.1.2 Tanks & Temples. `Tanks & Temples` dataset contains two large outdoor scenes (`Truck` and `Train`) with a prominent object in the center, around which the camera is rotating. These scenes are particularly challenging due to the illumination differences between individual frames as well as the presence of dynamic objects.

Similar to the results on `MipNeRF360` dataset, our method again performs on par with the state-of-the-art methods in terms of quality, while the speed is approximately 1.7 times slower than 3DGS at 190 FPS. Qualitative results are depicted in Figure 7. On this dataset, *Ours* achieves better PSNR than our *Ours (reference)*, but is still worse in terms of LPIPS and SSIM. We hypothesize that this is due to the lack of SSIM loss supervision when training the model with incoherent rays.

5.1.3 Deep Blending. Following Kerbl et al. [2023], we use two indoor scenes `Playroom` and `DrJohnson` from the `Deep Blending` dataset. Table 1 shows that our reference implementation *Ours (reference)* outperforms all baselines across all qualitative metrics. Different to other datasets, we observe a larger quality drop of *Ours*. This is a result of a quality drop on `Playroom` where we observed instability of the training with incoherent rays. We leave more detailed investigation and parameter tuning of incoherent rays training for future work.

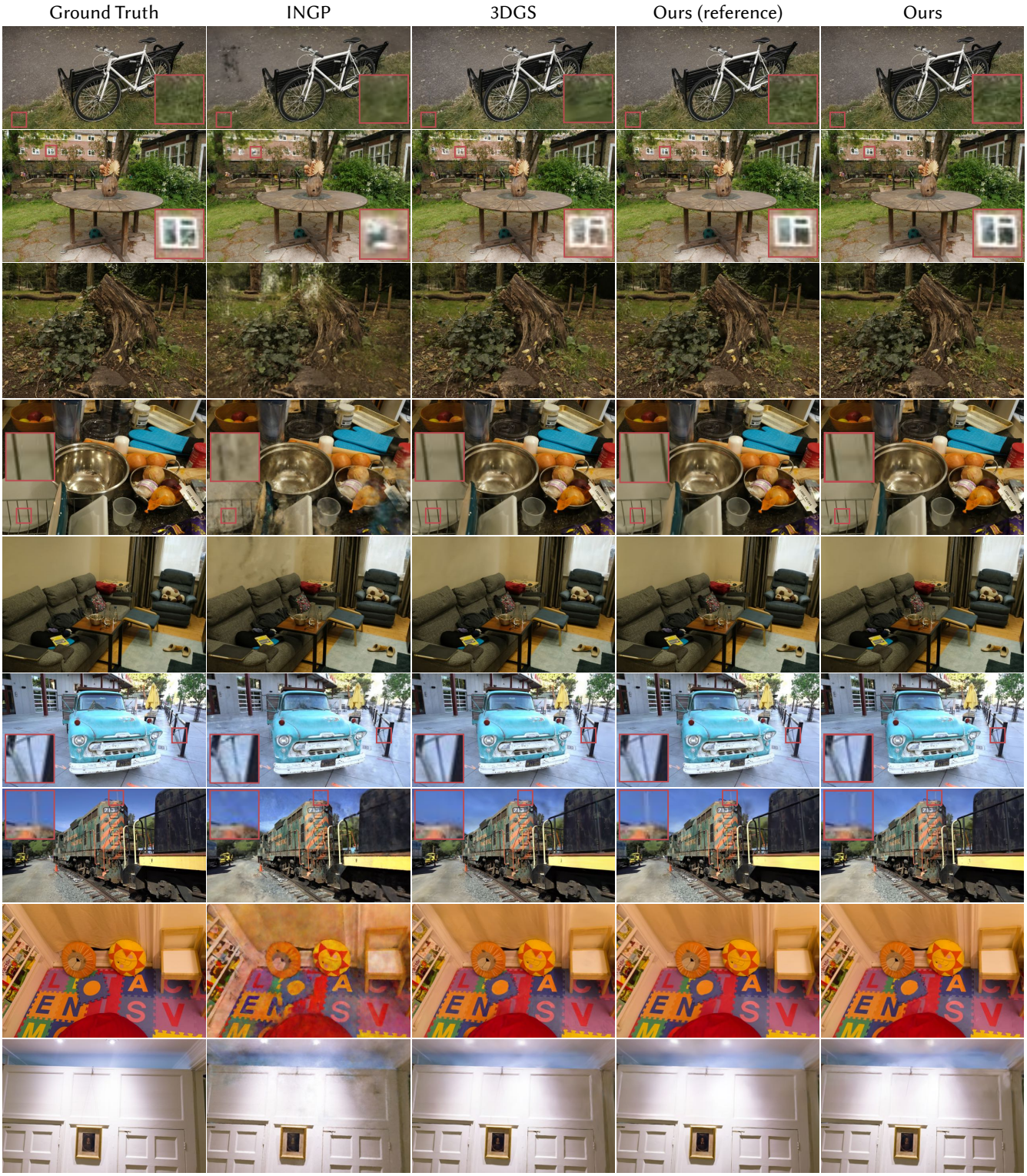


Fig. 7. **Novel-View Synthesis:** Qualitative comparison of our novel-view synthesis results relative to baselines (insets (•) show per-result closeups). For fairness, this comparison uses the same test views picked by [Kerbl et al. 2023]. Additional comparisons with [Barron et al. 2022] are included in the appendix.

Table 2. Rendering performance: rasterization v.s. ray tracing.

Method	FPS↑ MipNeRF360	Tanks & Temples	Deep Blending
3DGS (checkpoint)	238	319	267
Ours (reference)	55	143	77
Ours	78	190	119

5.1.4 NeRF Synthetic. NeRF Synthetic is a commonly used synthetic object-centring dataset introduced by Mildenhall et al. [2020]. The eight scenes with synthetic objects were rendered in Blender and display strong view-dependent effects and intricate geometry structures. See Table 5 in the appendix for a per-scene breakdown.

Both versions of our method outperform all the baselines in terms of PSNR. In fact, *Ours* outperforms *Ours (reference)* on seven out of eight scenes. We conjecture this is due to the simplicity of scenes which are well represented with less hits, and the positive contribution of training with incoherent rays. On these simpler scenes with lower resolution images, our method is capable of rendering novel views at 450FPS and is only 50% slower than 3DGS.

5.1.5 Zip-NeRF and Distortion. The rasterization approach in 3DGS [Kerbl et al. 2023] implicitly treats supervision images as coming from perfect-pinhole cameras, whereas our ray-tracing approach can easily be applied directly to highly-distorted cameras such as fisheye captures. Images can be undistorted through postprocessing to enable fitting with 3DGS, but this comes at a cost, including significant cropping or wasted space in the image plane.

We demonstrate this effect on the Zip-NeRF dataset [Barron et al. 2023], which includes four large-scale scenes featuring both indoor and outdoor areas. These scenes are originally captured from highly distorted fisheye cameras, with undistorted versions also provided through postprocessing. Table 3 and Figure 8 compare the quality of training 3DGS on undistorted views, *vs.* our ray-traced method on the undistorted views or the original distorted images. Our model achieves the highest quality when trained on the original distorted images, partly due to the loss of input supervision signals caused by the cropping of marginal pixels during undistortion. Note that rendering 3DGS from the original fisheye views is impossible, as its tile-based rasterizer is designed for perfect-pinhole rendering.

5.2 Ray Tracing Analysis and Ablations

We evaluate the performance of the proposed ray tracer and compare to alternatives. Experiments are evaluated on the union of all validation datasets from Section 5.1. We measure forward rendering time, which we observed to correlate closely with the per-iteration time for the backward pass.

5.2.1 Particle Primitives. We first consider different bounding primitive strategies as discussed in Section 4.1. The primitives evaluated are:

- **Custom primitive AABBs**: bounding box primitive, see Figure 4 left.
- **Octahedron**: an eight-faced regular polyhedron mesh.
- **Icosahedron**: a twenty-faced regular polyhedron mesh.
- **Icosahedron + unclamped**: icosahedron without adaptive clamping.

Table 3. Comparison of PSNR achieved by our method versus 3DGS [Kerbl et al. 2023] when trained and tested on distorted or undistorted views.

Method	Test views undistorted	original (fisheye)
3DGS w/ undistorted inputs	24.18	N/A
Ours (reference) w/ undistorted inputs	24.59	23.69
Ours (reference) w/ original inputs	24.71	24.40

Scales are determined as in Equation 7, except the *unclamped* variant which omits the opacity term in that expression.

Figure 9 (bottom-left) shows the time to build a BVH relative to the number of particles for the different primitives. For simple AABBs, the build time is almost constant whereas for the more complex icosahedron based primitives, the build time is close to linear with more than 30ms per millions of particles. The same figure also gives the framerate *vs.* the number of particles for different primitives. First, the *number* of particles does not strictly determine the performance. Second, more complex primitives with tighter bounding envelopes yield higher framerates, and adaptive clamping based on opacity has a large positive effect.

5.2.2 Tracing Algorithms. We consider several alternatives of the proposed ray tracer from Section 4.2, both comparisons to prior work and variants of our method. The evaluated approaches are:

- **Naive closest-hit tracing**: repeated *closest-hit* ray-tracing to traverse every particle hitting the ray in depth order.
- **Slab tracing [Knoll et al. 2019] (SLAB)**: tracing slabs along the ray, order independent collection of the k-first hits in the *any-hit* program, sorting and integrating the hits in the *ray-gen* program.
- **Multi-layer alpha tracing [Brüll and Grosch 2020] (MLAT)**: tracing a single ray with in depth order collection of the hits and merging the closest hits when the k buffer is full in the *any-hit* program.
- **Our proposed method**: tracing a dynamic number of rays with in-order collection of the hits, stopping to evaluate contributions when the k buffer is full in the *any-hit* program.
- **Ours + tiled tracing**: tracing one ray per $N \times N$ tile, but still evaluating appearance per pixel, akin to tile-based rasterization.
- **Ours + stochastic depth sampling**: tracing a single ray with in depth order collection of the k first accepted samples. Samples are accepted based on the importance sampling $q(\boldsymbol{x}) = \hat{\rho}(\boldsymbol{x})$.

For each algorithm with a choice of parameters (size of the array, number of slabs, or number of samples), we perform a parameter sweep and present the best-performing setting.

The performance relative to the accuracy of these implementations are shown in the top-left of Figure 9. Naive closest-hit tracing is almost twice as slow as our method, due to over-traversal of the BVH. Slab tracing and multi-layer alpha tracing are designed to minimize over-traversal and hence achieve much better runtime performance, but this speed comes from approximate image formation (ordering approximation for MLAT, under-sampling particles for SLAB), and the accuracy of these methods is substantially lower.

Fig. 8. Qualitative comparison of our results against 3DGS [Kerbl et al. 2023] when trained on distorted or undistorted views and then rendered from undistorted views.

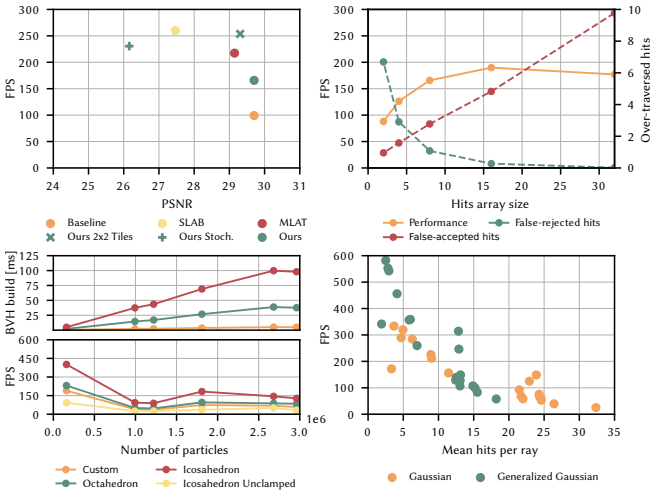


Fig. 9. **Quantitative Ablation.** Top left: comparison of the different tracing algorithms on the combination of our datasets. Top right: Impact of the hit payload buffer size on our proposed tracing algorithm. Bottom left: Impact of the different primitives on both the BVH building time and the FPS. Bottom right: Mean number of hits vs. mean FPS for every sequence of our dataset.

Table 4. Quality and speed tradeoffs for various particle kernel functions.

Particle\Metric	Tanks & Temples PSNR↑	Tanks & Temples FPS↑	Deep Blending PSNR↑	Deep Blending FPS↑
Gaussian (reference)	23.03	143	29.89	77
Generalized Gaussian ($n = 2$)	22.68	277	29.74	141
2D Gaussians	22.70	241	29.74	122
Cosine wave modulation	22.77	268	29.79	126

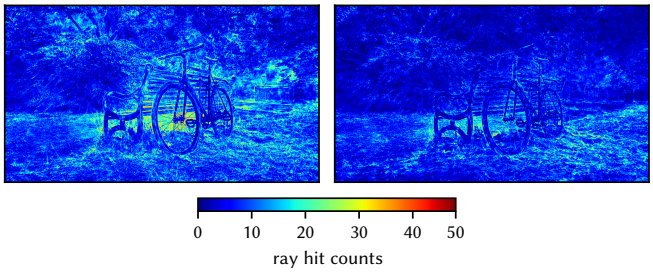


Fig. 10. **Ray Hits for Kernel Functions:** Visualization of the number of per-ray particles hits for the Gaussian (left) and for the generalized Gaussian kernel function of degree 2 (right) (• represents no hits).

In the differentiable setting, we find that these these approximations make those methods unusable for optimizing scenes. Adding tile-based rendering to our approach yields a promising speedup, at the cost of a small approximation. We do not see immediate benefit from stochastic depth sampling, because most of the computation has to be done in the *any-hit* program, preventing a good balance of the tracing workload.

5.2.3 Particle Kernel Function. In Section 4.5 we consider particle kernel functions beyond Gaussians. Table 4 gives results; notably generalized Gaussians with $n = 2$ significantly increase ray tracing speed at only a small cost of quality.

Figure 9 (bottom-right) shows the mean-hits number versus the performance for the Gaussian kernel and the generalized Gaussian kernel of degree 2. It reaffirms that the performance depends on the number of hits rather than the number of particles, as noted previously. This explains the source of the speedup for the generalized Gaussian kernel, as the sharper extent reduces the number of hits. See Figure 10 for a visual plot.

5.2.4 Hit Buffer Size. Figure 9 (top-right) measures the effect of the particle buffer size k, which determines how many particle hits are gathered during each raycast before stopping to evaluate their response. *False rejected hits* are hits which are traversed, but not collected into the buffer because it is full with closer hits; these hits often must be re-traversed later. *False accepted hits* are hits which are gathered into the buffer, but ultimately do not contribute to radiance because the transmittance threshold is already met. Both of these false hits harm performance, and choosing the particle batch size is a tradeoff between them. We find $k = 16$ to offer a good compromise and use it in all other experiments.

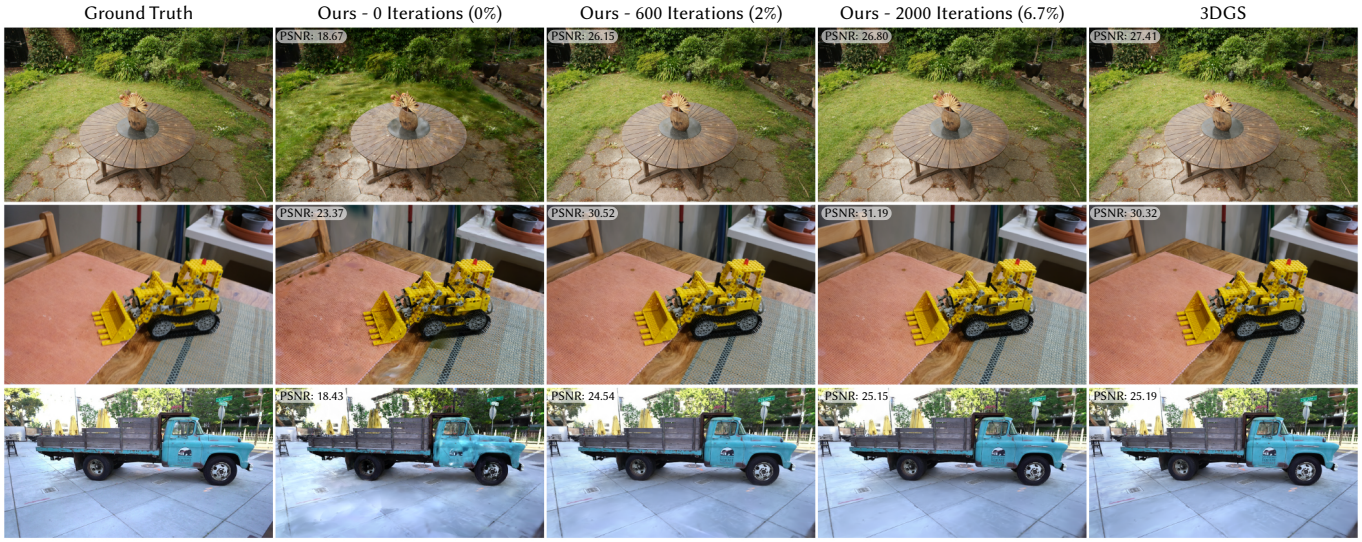


Fig. 11. **3DGS Finetuning:** Qualitative results of finetuned models from pretrained 3DGS [Kerbl et al. 2023] checkpoints after different numbers of iterations.

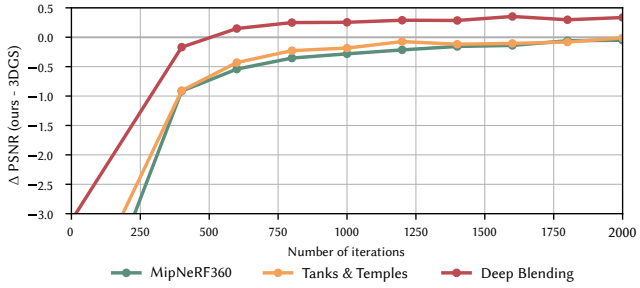


Fig. 12. **Finetuning Pretrained 3DGS Models:** After only 500 iterations of finetuning, we can recover most of the perceptual quality of 3DGS [Kerbl et al. 2023]. After 2k iterations we match or outperform 3DGS across all datasets.

6 APPLICATIONS

Most importantly, efficient differentiable ray tracing enables new applications and techniques to be applied to particle scene representations.

6.1 Ray-Based Effects

First, we the radiance field rendering pipeline with a variety of visual effects which are naturally compatible with ray tracing (Figure 2 and 13). Here we demonstrate only manually-specified forward rendering, although inverse rendering in concert with these effects is indeed supported by our approach, and is a promising area for ongoing work.

Reflections, Refractions and Inserted Meshes. Optical ray effects are supported by interleaved tracing of triangular faces and Gaussian particles. Precisely, we maintain an extra acceleration structure consisting only of mesh faces for additional inserted geometry. When casting each ray in Procedure 1, we first cast rays against inserted meshes; if a mesh is hit, we render all particles only up to the hit, and then compute a response based on the material. For refractions and reflections, this means continuing tracing along a new redirected ray according to the laws of optics. For non-transparent diffused meshes, we compute the color and blend it with the current ray radiance, then terminate the ray.

Depth of Field. Following [Müller et al. 2022], we simulate depth of field by progressively tracing multiple independent ray samples per pixel (spp), weighted together with a moving average to denoise the output image. The examples in Figures 2 and 13 use 64-256 spp, although convergence is often reached with fewer samples by selecting subsamples with low discrepancy sequences [Burley 2020].

Artificial Shadows. Even in radiance field scenes with baked-in lighting, simple shadow effects can be emulated by casting a ray towards a point or mesh emitter, and artificially darkening the contribution from that point if the light is not visible. We adopt this approach, casting shadow rays after computing the directional contribution from each particle.

6.2 Instancing

In rendering, *instancing* is a technique to render multiple transformed copies of an object with greatly reduced cost. Although rendering libraries may support some form of instancing in the context of rasterization, the technique is particularly effective for ray tracing. This is because repeated copies of an object can be stored as linked references in subtrees of the BVH, without actually duplicating the geometry. This allows for scaling to 100s or 1000s of instanced objects at little additional cost—the same is not possible with rasterization. Our efficient ray tracer enables particle scene data to be instanced, as shown in Figure 14 where we crop an object from a fitted scene and render 1024 copies of it at 25 FPS.

Fig. 13. **Rendering Effects:** The ray traced nature of our reconstructions allows seamless integration with traditional ray traced operations for reflecting and refracting rays, as well as casting shadows on nearby particles, as well as applying camera effects.

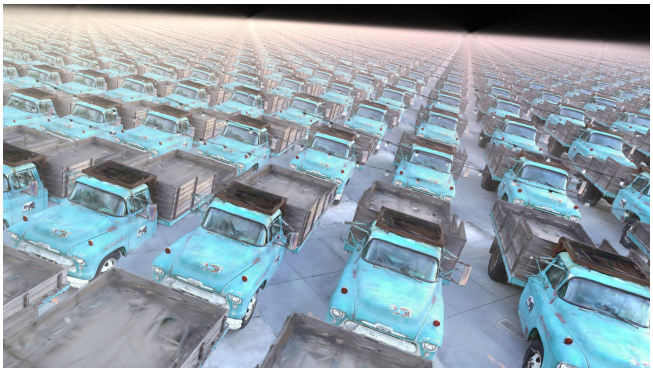


Fig. 14. **Instancing:** 1024 instances of the Tank & Temples Truck, rendered at more than 25 FPS.

Fig. 15. **Stochastic Sampling:** Left, scene rendered with our proposed algorithm. Center, rendered with stochastic sampling (4 samples). Right, denoised with the NVIDIA OptiX denoiser.

(a) Nonlinear Camera Models (b) Rolling Shutter Motion Compensation

Fig. 16. **Complex Cameras:** (a) Compared to rasterization-based approaches, our ray tracing-based formulation naturally supports complex camera models as inputs like distorted fisheye lenses (left), which can be re-rendered using different camera models like regular perspective cameras (right), achieving high reconstruction quality relative to *unseen* references (see insets (●) - this *synthetic* `cozyroom` scene is by [Ma et al. 2021]). (b) Ray tracing also naturally enables compensating for time-dependent effects like rolling shutter imagine sensors, which induce distortions due to sensor motion. This effect is illustrated on the left by multiple different frame tiles f_i of a *single* solid box rendered by a left- and right-panning rolling shutter camera with a top-to-bottom shutter direction. By incorporating time-dependent per-pixel poses in the reconstruction, our method faithfully recuperates the true undistorted geometry (right).

6.3 Denoising and Stochastic Sampling

In ray tracing more broadly, research on stochastic integration techniques is key to highly sample-efficient yet perceptually compelling renders. As a small demonstration, we show how our approach can be combined with stochastic sampling and denoising.

As discussed in Section 5.2.2, our approach may be extended to stochastic-sampling by rejecting the hits received in the *any-hit* program based on the importance sampling distribution $q = \hat{\rho}(\boldsymbol{x})$. Since traversal stops as soon as the k-closest accepted samples are collected, this modification noticeably improves performance (see Figure 9 top-left). This performance comes at a quality cost, but as shown in Figure 15, the resulting noise has statistics that an off-the-shelf denoiser can easily remove.

6.4 Complex Cameras and Autonomous Vehicle Scenes

Ray tracing makes it easy, efficient, and accurate to render from exotic cameras which are far from ideal pinholes, such as highly-distorted fisheye cameras and those with rolling shutter effects (Figure 16). While optical distortions for low-FOV lenses can be tackled to some extent by image rectification, and rolling shutter distortions can be approached by associating rasterized tiles to row/columns with consistent timestamps, both workarounds can't be applied simultaneously, as image rectification distorts the sets of concurrently measured pixels in a non-linear way. In ray tracing, handling complex cameras simply means generating each ray with source and direction which actually correspond to the underlying camera, even if those rays may be incoherent and lack a shared origin.

Autonomous Vehicles. The imaging systems used on autonomous vehicle (AV) platforms and other robot systems often incorporate such cameras, and it is very important to reconstruct and render them accurately in those applications. Figure 17, gives an example of an autonomous driving scene reconstructed from a side-mounted camera, which exhibits both apparent intrinsic camera and rolling shutter distortion effects.

To further highlight the importance of accurately handling these effects, we perform a quantitative evaluation of our method against its rasterization equivalent 3DGS [Kerbl et al. 2023] on autonomous driving scenes. We select 9 scenes from the `Waymo Open Perception` dataset [Sun et al. 2020] with no dynamic objects to ensure accurate reconstructions. Both methods are trained with the images captured by the camera mounted on the front of the vehicle to reconstruct the scene. We make several changes to the training protocol to adapt it to this data, including incorporating lidar depth, see the appendix for details. For the case of 3DGS, we rectify the images and ignore the rolling shutter effects, while with our tracing algorithm we can make use of the full camera model and compensate for the rolling shutter effect (see Figure 17). Ray tracing achieves a rectified PSNR of 29.99 on this benchmark, compared to 29.83 for ordinary 3DGS—the improvement is modest, but it corresponds to correctly reconstructing important geometries, such as the signpost in Figure 17.

Fig. 17. **AV Scene Reconstruction:** Real-world AV and robotics applications often have to respect both distorted intrinsic camera models and are, at the same time, affected by time-dependent effects like rolling shutter distortions as frames are exposed at high sensor speeds. Our ray tracing-based reconstruction is well-suited to handle *both* challenges simultaneously, which we illustrated by an example of a side-facing top-to-bottom rolling shutter camera on an AV vehicle: the top inset (●) depicts faithful removal of the intrinsic camera model distortion by rendering with an undistorted camera, while the bottom inset (●) shows our ability to remove the apparent rolling-shutter distortions of the inputs by rendering from a *static* camera pose (linear indicators (●) exemplify the complex distortion of the inputs).

7 DISCUSSION

7.1 Differences Between Ray Tracing and Rasterization

Here, we recap key differences between our ray tracer and the Gaussian splatting rasterizer proposed by Kerbl et al. [2023].

Generality. Splat rasterization accelerates rendering by processing a screen grid of pixel rays emanating from a single viewpoint in 16x16 tiles, whereas ray tracing uses a BVH and can render along arbitrary distributions of rays from any direction.

Primary vs. *General Rays.* As in graphics more broadly, ray tracing has significant benefits over rasterization to model general lighting and image formation. In this work, we demonstrate a variety of effects such as reflection, refraction, depth of field, and artificial shadows enabled by ray tracing (Section 6.1). In addition, we note that differentiable ray tracing opens the door to further research on global illumination, inverse lighting, and physical BSDFs.

Complex Cameras. Using per-pixel rays, ray tracing can easily model more general image formation processes that exhibit non-linear optical models such as highly-distorted and high-FOV fisheye lenses, as well as time-dependent effects like rolling shutter distortions, which originate from rows/columns exposed at different timestamps (*cf.* [Li et al. 2024b]). These are important for robotics (Section 6.4), yet difficult or impossible to model with tile-based rasterization.

Speed. For forward rendering, our approach achieves real-time performance, and is only about 2× slower than 3DGS's tiled rasterization in the basic case of rendering primary rays from pinhole cameras (see Table 2). For differentiable rendering to fit scenes, our ray tracing approach is 2×-5× slower than rasterization, mainly due to the need to rebuild the BVH (Section 4.4), and additional arithmetic needed to evaluate the backward pass as terms are no longer shared between pixels. As an example, for the *Tanks & Temples*'s *Truck* scene, ray tracing is 3.3× slower per iteration of optimization, with a mean iteration time of 100ms (30ms spent on the BVH construction, 15ms on the forward pass and 30ms on the backward pass), while the rasterization requires 30ms per-iteration (7ms on the forward pass and 10ms on the backward pass). Furthermore, our approach enables training with incoherent rays as discussed in Section 4.4, but in that case ray incoherency further increases the cost of ray tracing, leading to 5× slower optimization.

Sub-Pixel Behavior. Splat rasterization implicitly applies a pixel-scale convolution to Gaussians [Zwicker et al. 2001], whereas our ray tracer truly point-samples the rendering function and has no such automatic antialiasing. This may lead to differences of rendered appearance for subpixel-skinny Gaussian particles. However, point-sampled rendering is well-suited to modern denoisers (see Section 6.3).

Interoperability. It is possible to directly render scenes trained with the rasterizer under the ray tracing renderer, however due to subtle differences noted above, there will be a noticeable drop in quality when directly switching between renderers. This can be quickly remedied with fine-tuning (see Figure 12).

Approximations. Rasterization makes an approximation by evaluating directional appearance through the spherical harmonics $\boldsymbol{\beta}$ from a single ray direction, meaning each particle has constant appearance direction in all pixels of an image. To support arbitrary distributions of rays in our ray tracer, each particle is evaluated exactly with the appropriate incoming ray direction. Additionally, rasterization approximates depth ordering in 16x16 tiles.

7.2 Limitations and Future Work

Our ray tracer is carefully designed to make use of hardware acceleration and offers significant speedup over baseline implementations (Figure 9), however ray tracing is still slower than rasterization when rendering from a pinhole camera. Additionally, the need to regularly rebuild the BVH during training incurs additional cost and adds overhead to dynamic scenes. Nonetheless, our implementation is still fast enough for training and interactive rendering, and more importantly it opens the door to many new capabilities such as distorted cameras and ray-based visual effects (Section 6). See Section 7.1 for an in-depth discussion of the trade-offs of rasterization *vs.* ray tracing in this setting.

Looking forward, this work creates great potential for further research on inverse rendering, relighting, and material decomposition on particle representations. Indeed, recent work in this direction [Gao et al. 2023; Liang et al. 2023], has relied on approximations due to the lack of an efficient ray tracer. More broadly, there is much promising research to be done unifying advances in scene reconstruction from computer vision with the formulations for photorealistic rendering from computer graphics.

ACKNOWLEDGMENTS

We are grateful to Hassan Abu Alhaija, Ronnie Sharif, Beau Perschall and Lars Fabiunke for assistance with assets, to Greg Muthler, Magnus Andersson, Maksim Eisenstein, Tanki Zhang, Dietger van Antwerpen and John Burgess performance feedback, to Thomas Müller, Merlin Nimier-David, and Carsten Kolve for inspiration, to Ziyu Chen, Clement Fuji-Tsang, Masha Shugrina, and George Kopanas for experiment assistance, and to Ramana Kiran and Shailesh Mishra for typo fixes. The manta ray image is courtesy of abby-design.

REFERENCES

Maksim Aizenshtein, Niklas Smal, and Morgan McGuire. 2022. Wavelet Transparency. *CoRR* abs/2201.00094 (2022). arXiv:2201.00094 https://arxiv.org/abs/2201.00094

Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. 2020. Neural point-based graphics. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII 16.* Springer, 696–712.

Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. 2021. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. *ICCV* (2021).

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

Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. 2023. Zip-NeRF: Anti-Aliased Grid-Based Neural Radiance Fields. *arXiv* (2023).

Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. 2007. Multi-fragment effects on the GPU using the k-buffer *(I3D '07).* Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/1230100.1230117

Laurent Belcour, Cyril Soler, Kartic Subr, Nicolas Holzschuch, and Fredo Durand. 2013. 5D covariance tracing for efficient defocus and motion blur. *ACM Transactions on Graphics (TOG)* 32, 3 (2013), 1–18.

Felix Brüll and Thorsten Grosch. 2020. Multi-Layer Alpha Tracing. In *Vision, Modeling, and Visualization*, Jens Krüger, Matthias Niessner, and Jörg Stückler (Eds.). The Eurographics Association. https://doi.org/10.2312/vmv.20201183

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured Lumigraph Rendering. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 425–432.

Brent Burley. 2020. Practical Hash-based Owen Scrambling. *Journal of Computer Graphics Techniques (JCGT)* 10, 4 (29 December 2020), 1–20. http://jcgt.org/published/0009/04/01/

Abe Davis, Marc Levoy, and Fredo Durand. 2012. Unstructured Light Fields. *Comput. Graph. Forum* 31, 2pt1 (2012), 305–314.

Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. 1996. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. Association for Computing Machinery, 11–20.

Daniel Duckworth, Peter Hedman, Christian Reiser, Peter Zhizhin, Jean-François Thibert, Mario Lučić, Richard Szeliski, and Jonathan T. Barron. 2023. SMERF: Streamable Memory Efficient Radiance Fields for Real-Time Large-Scene Exploration. arXiv:2312.07541 [cs.CV]

Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. 2023. LightGaussian: Unbounded 3D Gaussian Compression with 15x Reduction and 200+ FPS. arXiv:2311.17245 [cs.CV]

Jian Gao, Chun Gu, Youtian Lin, Hao Zhu, Xun Cao, Li Zhang, and Yao Yao. 2023. Relightable 3D Gaussian: Real-time Point Cloud Relighting with BRDF Decomposition and Ray Tracing. *arXiv:2311.16043* (2023).

Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. 1996. The Lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. Association for Computing Machinery, 43–54.

Jeffrey P Grossman and William J Dally. 1998. Point sample rendering. In *Rendering Techniques' 98: Proceedings of the Eurographics Workshop in Vienna, Austria, June 29—July 1, 1998 9*. Springer, 181–192.

Antoine Guédon and Vincent Lepetit. 2023. SuGaR: Surface-Aligned Gaussian Splatting for Efficient 3D Mesh Reconstruction and High-Quality Mesh Rendering. *arXiv preprint arXiv:2311.12775* (2023).

Yuan-Chen Guo, Di Kang, Linchao Bao, Yu He, and Song-Hai Zhang. 2022. Nerfren: Neural radiance fields with reflections. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18409–18418.

Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2024. 2D Gaussian Splatting for Geometrically Accurate Radiance Fields. *SIGGRAPH* (2024).

Yingwenqi Jiang, Jiadong Tu, Yuan Liu, Xifeng Gao, Xiaoxiao Long, Wenping Wang, and Yuexin Ma. 2024. Gaussianshader: 3d gaussian splatting with shading functions for reflective surfaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5322–5332.

Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. 2007. Harmonic Coordinates for Character Articulation. *ACM Trans. Graph.* 26, 3 (jul 2007), 71–es. https://doi.org/10.1145/1276377.1276466

Michael M. Kazhdan, Matthew Bolitho, and Hugues Hoppe. 2006. Poisson Surface Reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing (SGP '06, Vol. 256)*. Eurographics Association, 61–70.

Michael M. Kazhdan and Hugues Hoppe. 2013. Screened poisson surface reconstruction. *ACM Trans. Graph.* 32, 3 (2013), 29:1–29:13.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* 42, 4 (July 2023). https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. 2024. A Hierarchical 3D Gaussian Representation for Real-Time Rendering of Very Large Datasets. *ACM Transactions on Graphics* 43, 4 (July 2024). https://repo-sam.inria.fr/fungraph/hierarchical-3d-gaussians/

Leonid Keselman and Martial Hebert. 2022. Approximate Differentiable Rendering with Algebraic Surfaces. In *European Conference on Computer Vision (ECCV)*.

Leonid Keselman and Martial Hebert. 2023. Flexible techniques for differentiable rendering with 3d gaussians. *arXiv preprint arXiv:2308.14737* (2023).

Aaron Knoll, R Keith Morley, Ingo Wald, Nick Leaf, and Peter Messmer. 2019. Efficient particle volume splatting in a ray tracer. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (2019), 533–541.

Georgios Kopanas, Julien Philip, Thomas Leimkühler, and George Drettakis. 2021. Point-Based Neural Rendering with Per-View Optimization. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 40, 4 (June 2021). http://www-sop.inria.fr/reves/Basilic/2021/KPLD21

Christoph Lassner and Michael Zollhöfer. 2021. Pulsar: Efficient Sphere-based Neural Rendering. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), 1440–1449.

Marc Levoy and Pat Hanrahan. 1996. Light Field Rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. Association for Computing Machinery, 31–42.

Moyang Li, Peng Wang, Lingzhe Zhao, Bangyan Liao, and Peidong Liu. 2024b. USB-NeRF: Unrolling Shutter Bundle Adjusted Neural Radiance Fields. arXiv:2310.02687 [cs.CV]

Max Zhaoshuo Li, Thomas Müller, Alex Evans, Russell H. Taylor, Mathias Unberath, Ming-Yu Liu, and Chen-Hsuan Lin. 2023. Neuralangelo: High-Fidelity Neural Surface Reconstruction. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ruilong Li, Sanja Fidler, Angjoo Kanazawa, and Francis Williams. 2024a. NeRF-XL: Scaling NeRFs with Multiple GPUs. arXiv:2404.16221 [cs.CV]

Zhihao Liang, Qi Zhang, Ying Feng, Ying Shan, and Kui Jia. 2023. Gs-ir: 3d gaussian splatting for inverse rendering. *arXiv preprint arXiv:2311.16473* (2023).

Li Ma, Xiaoyu Li, Jing Liao, Qi Zhang, Xuan Wang, Jue Wang, and Pedro V. Sander. 2021. Deblur-NeRF: Neural Radiance Fields from Blurry Images. *arXiv preprint arXiv:2111.14292* (2021).

Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. 2021. NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR*.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *ECCV*.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Trans. Graph.* 41, 4, Article 102 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530127

Cedrick Münstermann, Stefan Krumpen, Reinhard Klein, and Christoph Peters. 2018. Moment-Based Order-Independent Transparency. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 7 (jul 2018), 20 pages. https://doi.org/10.1145/3203206

Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. 2023. Compressed 3d gaussian splatting for accelerated novel view synthesis. *arXiv preprint arXiv:2401.02436* (2023).

Michael Niemeyer, Jonathan T. Barron, Ben Mildenhall, Mehdi S. M. Sajjadi, Andreas Geiger, and Noha Radwan. 2022. RegNeRF: Regularizing Neural Radiance Fields for View Synthesis from Sparse Inputs. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. 2024. RadSplat: Radiance Field-Informed Gaussian Splatting for Robust Real-Time Rendering with 900+ FPS. *arXiv preprint arXiv:2403.13806* (2024).

Julian Ost, Issam Laradji, Alejandro Newell, Yuval Bahat, and Felix Heide. 2022. Neural point light fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18419–18429.

Panagiotis Papantonakis, Georgios Kopanas, Bernhard Kerbl, Alexandre Lanvin, and George Drettakis. 2024. Reducing the Memory Footprint of 3D Gaussian Splatting. In *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, Vol. 7.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.* 29, 4, Article 66 (jul 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 335–342.

Christian Reiser, Stephan Garbin, Pratul P. Srinivasan, Dor Verbin, Richard Szeliski, Ben Mildenhall, Jonathan T. Barron, Peter Hedman, and Andreas Geiger. 2024. Binary Opacity Grids: Capturing Fine Geometric Detail for Mesh-Based View Synthesis. *SIGGRAPH* (2024).

Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. 2021. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. In *International Conference on Computer Vision (ICCV)*.

Christian Reiser, Richard Szeliski, Dor Verbin, Pratul P Srinivasan, Ben Mildenhall, Andreas Geiger, Jonathan T Barron, and Peter Hedman. 2023. Merf: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes. *arXiv preprint arXiv:2302.12249* (2023).

Konstantinos Rematas, Andrew Liu, Pratul P. Srinivasan, Jonathan T. Barron, Andrea Tagliasacchi, Thomas Funkhouser, and Vittorio Ferrari. 2022. Urban Radiance Fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 12932–12942.

Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. 2024. Octree-GS: Towards Consistent Real-time Rendering with LOD-Structured 3D Gaussians. *arXiv preprint arXiv:2403.17898* (2024).

Liu Ren, Hanspeter Pfister, and Matthias Zwicker. 2002. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum*, Vol. 21. Wiley Online Library, 461–470.

Gernot Riegler and Vladlen Koltun. 2020. Free View Synthesis. In *European Conference on Computer Vision*.

Gernot Riegler and Vladlen Koltun. 2021. Stable View Synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Darius Rückert, Linus Franke, and Marc Stamminger. 2022. Adop: Approximate differentiable one-pixel point rendering. *ACM Transactions on Graphics (ToG)* 41, 4 (2022), 1–14.

Marco Salvi and Karthikeyan Vaidyanathan. 2014. Multi-layer alpha blending. *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2014). https://api.semanticscholar.org/CorpusID:18595625

Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance Fields without Neural Networks. In *CVPR*.

Johannes Lutz Schönberger and Jan-Michael Frahm. 2016. Structure-from-Motion Revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. 2016. Pixelwise View Selection for Unstructured Multi-View Stereo. In *European Conference on Computer Vision (ECCV)*.

Otto Seiskari, Jerry Ylilammi, Valtteri Kaatrasalo, Pekka Rantalankila, Matias Turkulainen, Juho Kannala, Esa Rahtu, and Arno Solin. 2024. Gaussian Splatting on the Move: Blur and Rolling Shutter Compensation for Natural Camera Motion. *arXiv preprint arXiv:2403.13327* (2024).

Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Haithem Turki, Deva Ramanan, and Mahadev Satyanarayanan. 2022. Mega-NERF: Scalable Construction of Large-Scale NeRFs for Virtual Fly-Throughs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 12922–12931.

Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. 2021. NeuS: Learning Neural Implicit Surfaces by Volume Rendering for Multi-view Reconstruction. *NeurIPS* (2021).

Zian Wang, Tianchang Shen, Jun Gao, Shengyu Huang, Jacob Munkberg, Jon Hasselgren, Zan Gojcic, Wenzheng Chen, and Sanja Fidler. 2023a. Neural Fields meet Explicit Geometric Representations for Inverse Rendering of Urban Scenes. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zian Wang, Tianchang Shen, Merlin Nimier-David, Nicholas Sharp, Jun Gao, Alexander Keller, Sanja Fidler, Thomas Müller, and Zan Gojcic. 2023b. Adaptive Shells for Efficient Neural Radiance Field Rendering. *ACM Trans. Graph.* 42, 6, Article 259 (2023), 15 pages. https://doi.org/10.1145/3618390

Turner Whitted. 1979. An improved illumination model for shaded display. *Seminal graphics: pioneering efforts that shaped the field* (1979). https://api.semanticscholar.org/CorpusID:9524504

Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. 2000. Surface Light Fields for 3D Photography. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., 287–296.

Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. 2022. Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5438–5448.

Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. 2021. Volume rendering of neural implicit surfaces. In *Thirty-Fifth Conference on Neural Information Processing Systems*.

Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. 2001. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 371–378.

Tianyi "Tanki" Zhang. 2021. Handling Translucency with Real-Time Ray Tracing. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX* (2021), 127–138.

A IMPLEMENTATION AND TRAINING DETAILS

We wrap the NVIDIA OptiX tracer as a Pytorch extension and train our representation using Adam optimizer for 30,000 iterations. We set the learning rates for rotations, scales, and zeroth-order spherical harmonics to 0.001, 0.005, and 0.0025, respectively. The learning rate for the remaining spherical harmonics coefficients is 20 times smaller than for the zeroth-order coefficient. Finally, we set the density learning rate to either 0.05 for high-quality settings or 0.09 for fast-inference settings.

After initial 500 iterations, we start the densification and pruning process, which we perform until 15,000 iterations are reached. To densify the particles, we accumulate 3D positional gradients, scaled by half the distance of each particle to the camera, to prevent under-densification in distant regions. In line with 3DGS [Kerbl et al. 2023], we split the particles if their maximum scale is above 1% of the scene extent and clone them otherwise. Pruning directly removes particles whose opacity is below 0.01. Additionally, we employ a simple heuristic to cap the maximum number of particles to 3,000,000. We denote this pruning strategy as *visibility pruning*. Specifically, if the densification step results in more particles, we reduce their number to 2,700,000 by pruning particles with minimum accumulated weight contribution on the training views. Moreover, while densification and pruning are in effect and similar to 3DGS, we reset the particle densities to 0.01 every 3000 iterations. During training, we perform early stopping to terminate the tracing of rays whose accumulated transmittance falls below 0.001. During inference, we increase this threshold to 0.03 for improved efficiency. We begin by solely training the constant spherical harmonic and progressively increase the spherical harmonics' degree every 1000 iterations, up to a maximum of 3. We update the BVH every iteration and reconstruct it after each pruning and densification.

For experiments with random-rays, during the last 15,000 iterations, we sample random rays across all training views with a batch size of $2^{19} = 524{,}288$, and only use the $L1$ loss to supervise the particles.

A.1 Autonomous Vehicles

To fit autonomous vehicle scenes, we modify our training protocol, including incorporating lidar and depth supervision. To initialize, we randomly sample 1 million lidar points visible in at least one training image. These points are assigned an initial color via lookup projected into a training image, and assigned an initial scale based on the distance to the closest recorded camera pose. During training, we use lidar to supervise depth; in our ray tracer depth is computed by integrating the distance along the ray to each sample as if it were radiance. Note that in 3DGS, lidar depth must be approximated by projecting lidar rays onto camera images, yet in ray tracing lidar rays can be directly cast into the scene. Additionally, we reconstruct the sky following [Rematas et al. 2022] and employ a directional MLP which predicts the color of the sky based on the ray direction. A sky segmentation is included as input, and used to supervise ray opacities computed from the particle scene.

Table 5. Quantitative evaluation on the `NeRF Synthetic` dataset [Mildenhall et al. 2020]

	NeRF Synthetic								
Method	Chair	Drum	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65	31.10
MipNeRF	35.14	25.48	33.29	37.48	35.70	30.71	36.51	30.41	33.09
INGP	35.00	26.02	33.51	37.40	36.39	29.78	36.22	31.10	33.18
AdaptiveShells	34.94	25.19	33.63	36.21	33.49	27.82	33.91	29.54	31.84
3DGS (paper)	35.83	26.15	34.87	37.72	35.78	30.00	35.36	30.80	33.32
Ours (reference)	35.90	25.77	35.94	37.51	36.01	29.95	35.66	30.71	33.48
Ours	36.02	25.89	36.08	37.63	36.20	30.17	34.27	30.77	33.38

Table 6. Quantitative PSNR ablation on the maximum number of allowed particles using *ours*.

	Maximum Allowed Particles					
Dataset	1×10^6	2×10^6	3×10^6	4×10^6	5×10^6	6×10^6
`Tanks & Temples`	23.21	23.19	23.20	23.14	23.15	23.20
`Deep Blending`	29.24	29.17	29.23	29.14	29.24	29.15

B ADDITIONAL EXPERIMENTS AND ABLATIONS

Figure 18 shows qualitative comparisons of our method against MIPNeRF360 [Barron et al. 2022]. The zoomed-in insets demonstrate that both of our settings achieve comparable or better renderings with sharp features. The first three rows contain scenes from the `MipNeRF360` dataset, while the last two rows feature scenes from `Tanks & Temples`.

As mentioned in Section A, we propose a simple visibility pruning strategy to prevent the number of particles from exceeding a certain threshold. Table 6 presents an ablation study on the maximum number of allowed particles for scenes in two datasets: `Tanks & Temples` and `Deep Blending`. When densification causes the number of particles in the scene to exceed the threshold, we prune the least visible particles based on their accumulated contribution to the training views, reducing the number of particles to 90% of the threshold. The results show that our visibility pruning strategy, which filters out particles that contribute the least to the rendered views, maintains quality even with as few as one million particles.

Fig. 18. **Additional Qualitative Comparisons:** novel-view synthesis results relative to the MIPNeRF360 baseline (insets (●) show per-result closeups).