

Supplementary

RayGauss: Volumetric Gaussian-Based Ray Casting for Photorealistic Novel View Synthesis

1. Overview

In this supplementary material, we first provide technical details about our method: an explicit description of the different basis functions tested and their influence on various aspects of the algorithm, details about our OptiX implementation, and a description of optimization details. Next, we address secondary features specific to our approach: the ability to simultaneously cast rays from different viewpoints compared to splatting and the possibilities enabled by the OptiX API [11]. Finally, we discuss additional tests conducted on the hyperparameters of our algorithm: the sampling step Δt and the density threshold σ_ϵ . We then compare our method to state-of-the-art approaches attempting to produce an antialiased scene representation. For this purpose, we test our approach by adding brute force supersampling to study its potential for antialiasing.

2. Description of the different basis functions and their intersection

We provide more details here on the different tested basis functions and the resulting treatments. Our implementation is flexible and allows easy modification of the basis function used. Such modification impacts the following aspects of the code: construction of the Bounding Volume Hierarchy, intersection program, and weights evaluation for each sample, as explained below.

2.1. Explicit expressions of the studied basis functions

As explained in the main article, we limit ourselves to the study of decreasing radial and elliptical basis functions, which can be expressed depending on $r(\mathbf{x}) = \frac{d_2(\mathbf{x}, \mu)}{R}$ in the radial case or $r(\mathbf{x}) = d_M(\mathbf{x}, \mu)$ in the elliptical case. Here, $R \in \mathbb{R}$ can be interpreted as a shape parameter, d_2 denotes the Euclidean distance in \mathbb{R}^3 , and $d_M(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)}$ represents the Mahalanobis distance associated with the covariance matrix Σ and mean position μ .

Also, we have studied two types of basis functions: those

with local, or more precisely compact, support that vanish for $r > 1$, and those with global support. We describe below these different functions.

Functions with local/compact support: The studied functions with compact support are as follows:

- The Bump function is defined as:

$$\phi_{Bump}(r) = \begin{cases} e^{1 - \frac{1}{1-r^2}} & \text{if } r < 1 \\ 0 & \text{if } r \geq 1 \end{cases} \quad (1)$$

- Wendland functions denote a class of functions, and here we use one of the most commonly employed:

$$\phi_{Wendland}(r) = \begin{cases} (1-r)_+^4 (4r+1) & \text{if } r \leq 1 \\ 0 & \text{if } r > 1 \end{cases} \quad (2)$$

It can be noted that the Bump function has been slightly modified compared to its usual expression, as it has been multiplied by e^1 , so that $\phi_{Bump}(0) = 1$, which corresponds to the behavior of the other basis functions studied and facilitates experiments.

Functions with global support: The studied functions with global support are as follows:

- The inverse multiquadric function:

$$\phi_{Inv_Multi}(r) = \frac{1}{\sqrt{1+r^2}} \quad (3)$$

- The inverse quadratic function:

$$\phi_{Inv_Quad}(r) = \frac{1}{1+r^2} \quad (4)$$

- The C^0 -Matérn function:

$$\phi_{C0_Matérn}(r) = \exp(-r) \quad (5)$$

- The Gaussian function:

$$\phi_{Gaussian}(r) = \exp\left(-\frac{r^2}{2}\right) \quad (6)$$

Also, the expression used to evaluate weights for each sample corresponds to one of the expressions given above, depending on the tested case. In addition, as specified in the main article, global support functions are truncated to consider only their value within a domain restricted by the following condition:

$$\sigma_l(\mathbf{x})_{\text{approx}} = \begin{cases} \sigma_l(\mathbf{x}) & \text{if } \sigma_l(\mathbf{x}) \geq \sigma_\epsilon \\ 0 & \text{if } \sigma_l(\mathbf{x}) < \sigma_\epsilon \end{cases} \quad (7)$$

where:

$$\sigma_l(\mathbf{x}) = \tilde{\sigma}_l \cdot \phi_l(\mathbf{x}) \quad (8)$$

with ϕ_l the l -th basis function and $\tilde{\sigma}_l$ the associated density parameter. This allows considering only functions with a non-negligible contribution at a given point by setting a sufficiently low threshold. Moreover, no domain truncation is necessary for compactly supported functions, as they naturally vanish outside a compact domain by their definition.

After modification, the support of these functions corresponds to a solid sphere if the functions are radial or to a solid ellipsoid if they are elliptical. Since elliptical functions yield better results in our case, we will subsequently explain only the case of ellipsoids (which is also a generalization of spheres). Additionally, the OptiX API we use does not natively support intersection with ellipsoids but allows for the definition of a custom intersection program. This works by defining two aspects: the definition of axis-aligned bounding boxes (AABBs) that encompass each primitive and the definition of the custom primitive intersection (ellipsoid in our case). Thus, for a given ray and a given slab assimilated to a segment, we test the intersection of the segment with primitives in the following way: OptiX first finds an intersection with the AABBs, and for the intersected AABBs, it executes the custom program defined by us, calculating the intersection between a segment and an ellipsoid. Furthermore, the ellipsoid depends on the basis function used. So, as previously mentioned, the basis function influences the construction of the BVH through the provided AABBs and the custom intersection program. We will discuss these two points below.

2.2. Ellipsoid definition for the basis function

To calculate the intersection of a ray with the support of a basis function, we must first explicitly define its expression. In the case of truncated global functions, by using equations 7 and 8, we derive that the support of the l -th truncated global function is the set of points satisfying the condition:

$$\phi_l(r(\mathbf{x})) \geq \frac{\sigma_\epsilon}{\tilde{\sigma}_l} \quad (9)$$

Also, since all tested global support basis functions are

invertible and decreasing, ϕ_l^{-1} is decreasing, and:

$$r(\mathbf{x}) \leq \phi_l^{-1}\left(\frac{\sigma_\epsilon}{\tilde{\sigma}_l}\right) \quad (10)$$

Here, since we are considering the elliptical case, we have:

$$\sqrt{(\mathbf{x} - \mu_l)^T \Sigma_l^{-1} (\mathbf{x} - \mu_l)} \leq \phi_l^{-1}\left(\frac{\sigma_\epsilon}{\tilde{\sigma}_l}\right) \quad (11)$$

And thus (by the growth of the square function on \mathbb{R}^+):

$$(\mathbf{x} - \mu_l)^T \Sigma_l^{-1} (\mathbf{x} - \mu_l) \leq \left(\phi_l^{-1}\left(\frac{\sigma_\epsilon}{\tilde{\sigma}_l}\right)\right)^2 \quad (12)$$

where we can recognize the equation of an ellipsoid.

The expression is simpler in the case of locally/compactly supported functions. Indeed, the tested functions vanish starting from $r = 1$. Therefore, we can deduce the equation of the ellipsoid in this case:

$$(\mathbf{x} - \mu_l)^T \Sigma_l^{-1} (\mathbf{x} - \mu_l) \leq 1 \quad (13)$$

2.3. Tightest axis-aligned bounding box definition

The OptiX API optimizes Ray-AABB intersection. Also, our goal here is to build the BVH using the tightest axis-aligned bounding boxes enclosing previous ellipsoids to avoid unnecessary intersection calculations. In this elliptical case, the resulting axis-aligned bounding box (AABB) has a slightly complex expression. It is bounded by the following coordinates:

$$\begin{aligned} x &= \mu_x \pm \sqrt{\tilde{s}_x^2 \mathbf{R}_{1,1}^2 + \tilde{s}_y^2 \mathbf{R}_{1,2}^2 + \tilde{s}_z^2 \mathbf{R}_{1,3}^2}, \\ y &= \mu_y \pm \sqrt{\tilde{s}_x^2 \mathbf{R}_{2,1}^2 + \tilde{s}_y^2 \mathbf{R}_{2,2}^2 + \tilde{s}_z^2 \mathbf{R}_{2,3}^2}, \\ z &= \mu_z \pm \sqrt{\tilde{s}_x^2 \mathbf{R}_{3,1}^2 + \tilde{s}_y^2 \mathbf{R}_{3,2}^2 + \tilde{s}_z^2 \mathbf{R}_{3,3}^2}. \end{aligned} \quad (14)$$

where $\mathbf{R}_{i,j}$ are the coordinates of the rotation matrix \mathbf{R} associated with the ellipsoid as described in the main article. Moreover, $\tilde{s} = s \cdot \phi_l^{-1}\left(\frac{\sigma_\epsilon}{\tilde{\sigma}_l}\right)$ in the case of truncated global functions and $\tilde{s} = s$ in the case of compactly supported basis functions. Here, s refers to the diagonal of the scale matrix \mathbf{S} defined in the main paper.

In the radial case, this is straightforward as we take the cube of side $2R\phi_l^{-1}\left(\frac{\sigma_\epsilon}{\tilde{\sigma}_l}\right)$ centered on μ in the case of truncated global functions and the cube of side $2R$ centered on μ , in the case of compactly supported basis functions.

2.4. Custom intersection definition

To implement the intersection, we use the ellipsoid equations 12 and 13, depending on the basis function, and calculate the intersection of the current segment with this ellipsoid. In particular, we leverage the optimized Ray-Sphere intersection introduced in [6] and adapted for the case of ellipsoids.

3. Optix Raycasting

Here we describe the details of our RayCasting algorithm implementation using the OptiX API. This API works by allowing the definition of several custom programs that define the behavior of the rendering pipeline. The different programs of interest in our case are as follows:

- **Ray Generation:** This program is called first and is executed in parallel for each pixel, launching rays into the BVH.
- **Intersection:** Defines the ray-primitive intersection with our custom primitive, an ellipsoid in our case.
- **Any-Hit:** This program is called when the Intersection program finds a new intersection along the ray, allowing custom processing of the intersected primitives.

From these three programs, we can define our raycasting algorithm. In particular, the Ray Generation program calculates the origin and direction of the ray associated with a given pixel, then a slab size corresponding to a multiple of Δt is fixed. This slab size is the same for all rays to maintain inter-ray coherence. In practice, in most cases, we treat 8 samples per slab. Next, if a given ray intersects the axis-aligned bounding box associated with the set of primitives, then this ray intersects the scene. So we start probing the space traversed by the ray, proceeding slab by slab. For a given segment on the ray, we launch the traversal of the BVH. The intersection program computes the intersection of the segment with ellipsoids in the scene. The any-hit program collects the primitives intersecting the segment by storing their index in a large pre-allocated buffer. In practice, the buffer size is set to store at most 512 or 1024 primitives depending on the scene, thus allowing us to have more than enough primitives per buffer (between 512 and 1024 for 8 samples). This choice was made to ensure fast code execution. Once the primitives contributing locally are collected, we can calculate the value of σ and c for each of the samples in the slab, then accumulate them in the form of an intermediate color and transmittance of the ray. We can then use the early termination strategy, which ends the calculation when the current transmittance becomes lower than a threshold: $T < T_\epsilon$. When the threshold is low, this strategy allows us to disregard the samples whose contribution will be negligible compared to the overall color of the ray. If early termination isn't applied and we are still within the bounds of the scene, we can then move to the next slab and repeat the same process. We provide the ray generation and any-hit programs for performing ray casting in algorithm 1 and 2. The intersection program is not described in detail as it corresponds to the ellipsoid-segment intersection.

Algorithm 1 Any-Hit Program

Input: n_p : number of intersected primitives, n_{max} : maximum number of primitives, hitBuffer: buffer storing primitive indices, i_R : index of the current ray
Output: n_p , hitBuffer

```

1:  $i_P \leftarrow \text{optixGetPrimitiveIndex}()$   $\triangleright$  Current primitive index
2:  $\text{hitBuffer}[i_R \times n_{max} + n_p] \leftarrow i_P$ 
3:  $n_p \leftarrow n_p + 1$ 
4: if  $n_p \geq n_{max}$  then
5:    $\text{optixTerminateRay}()$   $\triangleright$  Terminate if max. primitives
6: end if
7:  $\text{optixIgnoreIntersection}()$   $\triangleright$  Continue Traversal
```

Algorithm 2 Ray Generation Program

Input: i_R : index of the current ray, $bbox_{min}$: minimum bounds of the bounding box, $bbox_{max}$: maximum bounds of the bounding box, Δt : step size, B : size of the buffer, T_ϵ : transmittance threshold, hitBuffer: buffer storing primitive indices, P: global parameters (primitive parameters and ray colors)
Output: P: update parameters

```

1:  $o, d \leftarrow \text{ComputeRay}(i_R)$   $\triangleright$  Ray origin, direction
2:  $t_0, t_1 \leftarrow \text{IntersectBBox}(o, d, bbox_{min}, bbox_{max})$ 
3:  $\Delta S \leftarrow \Delta t \times B$   $\triangleright$  Slab size
4:  $T \leftarrow 1.0$   $\triangleright$  Ray transmittance
5:  $C_R \leftarrow (0.0, 0.0, 0.0)$   $\triangleright$  Ray color
6: if  $t_0 < t_1$  then  $\triangleright$  Check if ray intersects bounding box
7:    $t_S \leftarrow t_0$   $\triangleright$  Current slab distance along the ray
8:   while  $t_S < t_1$  and  $T > T_\epsilon$  do
9:      $n_p \leftarrow 0$   $\triangleright$  Number of primitives
10:     $t_{min\_S} \leftarrow \max(t_0, t_S)$ 
11:     $t_{max\_S} \leftarrow \min(t_1, t_S + \Delta S)$ 
12:     $\triangleright$  Collect the intersected primitives
13:     $\text{Traversal}(\text{hitBuffer}, o, d, t_{min\_S}, t_{max\_S}, n_p)$ 
14:    if  $n_p == 0$  then
15:       $t_S \leftarrow t_S + \Delta S$ 
16:      continue
17:    end if
18:     $\text{densityBuffer} \leftarrow (0.0)^B$ 
19:     $\text{colorBuffer} \leftarrow (0.0, 0.0, 0.0)^B$ 
20:     $\triangleright$  Update ray color and density
21:     $\text{UpdateRay}(i_R, n_p, \Delta t, t_S, o, d, \text{densityBuffer}, \text{colorBuffer}, C_R, T, P)$ 
22:     $t_S \leftarrow t_S + \Delta S$ 
23:  end while
24: end if
25:  $\text{Pray\_colors}[i_R] \leftarrow C_R$ 
```

4. Optimization details

This section provides further details on the optimization parameters used in our experiments. We use the Adam gradient descent optimization algorithm [8]. We recall that the parameters optimized by our approaches are as follows: $P = \{(\tilde{\sigma}_l, \tilde{c}_l, \mu_l, \mathbf{q}_l, \mathbf{s}_l) \mid l = 1, \dots, N\}$ where $\tilde{\sigma}_l$ is the density parameter of the l -th primitive, \tilde{c}_l summarizes the

colorimetric parameters: lobe $\lambda_{l,j}$, lobe direction $\mathbf{p}_{l,j}$, coefficients $k_{l,j}$, for the j -th Spherical Gaussian and coefficients $\tilde{c}_{l,jm}$ for Spherical Harmonic of degree j and order m , while μ_l , \mathbf{q}_l , and \mathbf{s}_l are the mean position, quaternion, and scale parameter used to evaluate the basis function of the l -th primitive. The initial parameters are obtained from a point cloud created by Structure-From-Motion methods such as Colmap [12].

We will now describe the main parameters of our optimization, starting with the learning rates. In the context of evaluations on the Blender dataset, we employed a learning rate for density with exponential decay, starting at 5×10^{-1} and decreasing to 3×10^{-2} over 30,000 iterations. The learning rates for other parameters were set as follows: 1.0×10^{-3} for constant color parameters (RGB), 2.6×10^{-4} for Spherical Harmonics coefficients, 3.6×10^{-4} for Spherical Gaussians coefficients, 4.5×10^{-2} for lobe sharpness, 3.0×10^{-3} for lobe direction, 1.2×10^{-2} for scale parameters, 2.2×10^{-4} for quaternions, and a learning rate with exponential decay starting from 1.7×10^{-5} and reaching 1.0×10^{-6} after 30,000 iterations for mean positions. We optimize the scene using a white background to place ourselves in evaluation conditions like other state-of-the-art methods. In the case of the Mip-NeRF360 dataset, we use similar learning rates except for the density, for which we use a learning rate with exponential decay starting from 5×10^{-1} and reaching 1×10^{-4} in 30,000 iterations. This allows us to accelerate the beginning of the optimization, particularly for outdoor scenes with a large number of Gaussians. Furthermore, we train with a black background on this dataset to place ourselves in evaluation conditions similar to Gaussian Splatting and current state-of-the-art methods [7] [15].

More generally, we optimize the scene by backpropagation, with the optimization taking 30,000 iterations (1 image per iteration). Quaternions are renormalized after each optimization step to maintain a valid rotation matrix representation. The process of Adaptive Gaussian Control is similar to that used in 3D Gaussian Splatting [7], periodically increasing, every $i_{densify}$ iterations, the number of Gaussians in the scene using a heuristic based on the condition $\nabla L_{\mu_l} > \nabla L_\epsilon$, where ∇L_{μ_l} denotes the gradient on the mean position of the basis functions and ∇L_ϵ the threshold from which densification is applied. In our case, we densify every 500 iterations for Mip-NeRF360 and 300 iterations for Blender dataset, starting from the 500th iteration up to the 15,000th iteration, and we set $\nabla L_\epsilon = 0.00002$ for the Mip-NeRF 360 dataset and $\nabla L_\epsilon = 0.00004$ for the Blender dataset. Furthermore, we remove primitives whose density parameter $\tilde{\sigma}$ is below a threshold, set to 0.1 for our experiments on the Blender dataset and 0.01 on the Mip-NeRF 360 dataset. Finally, as described in the main paper, we gradually unlock the colorimetric parameters from

the lowest frequency representation to the potentially higher frequency one: harmonics of degree 0, 1, 2, and finally, 7 spherical Gaussians, resulting in 16 functions to represent the color of a primitive. The unlocking of these parameters occurs every 1000 iterations. We can specify here that the spherical harmonic and spherical gaussian coefficients, as well as the lobe sharpness, are initialized to zero, while the lobe axes are randomly initialized on the unit sphere and constrained to it during optimization. Also, one can refer to Algorithm 3 to gain a broader view of the optimization process.

Algorithm 3 RayGauss Scene Optimization

Input: $(I_i)_{i=1}^N$: N training images, i_{max} maximum number of iterations, ∇L_ϵ gradient threshold for densification, σ_ϵ density threshold for pruning

Output: optimized primitive parameters P

```

1:  $(V_i)_{i=1}^N, \tilde{P} \leftarrow \text{SFM}((I_i)_{i=1}^N)$   $\triangleright$  Camera, Sparse Point Cloud
2:  $P \leftarrow \text{InitPrim}(\tilde{P})$   $\triangleright$  Initialize primitive parameters
3:  $T \leftarrow \text{InitBVH}(P)$   $\triangleright$  Bounding Volume Hierarchy
4:  $i \leftarrow 0$ 
5: while  $i < i_{max}$  do
6:    $V, I \leftarrow \text{SampleTrainView}()$ 
7:    $\hat{I} \leftarrow \text{RayCast}(V, P, T)$ 
8:    $L \leftarrow \text{Loss}(\hat{I}, I)$ 
9:    $P \leftarrow \text{AdamOptim}(\nabla L)$ 
10:  for  $(\tilde{\sigma}_l, \tilde{c}_l, \mu_l, \mathbf{q}_l, \mathbf{s}_l)$  in  $P$  do
11:    if  $\text{IsAdaptControllter}(i)$  then
12:      if  $\nabla L_{\mu_l} > \nabla L_\epsilon$  then
13:         $P \leftarrow \text{CloneSplit}((\tilde{\sigma}_l, \tilde{c}_l, \mu_l, \mathbf{q}_l, \mathbf{s}_l))$ 
14:      end if
15:      if  $\tilde{\sigma}_l < \sigma_\epsilon$  then
16:         $\text{RemovePrimitive}()$ 
17:      end if
18:    end if
19:    if  $\text{IsUnlockIter}(i)$  then
20:       $\tilde{c}_l \leftarrow \text{UnlockColorSHSG}(i)$ 
21:    end if
22:  end for
23:   $T \leftarrow \text{UpdateBVH}(T, P)$ 
24: end while
```

Furthermore, it should be noted that the number of parameters in our representation is $87N_P$, where N_P is the number of primitives. Specifically, we have the following parameter counts: 3 for the mean position μ , 3 for the scale parameter \mathbf{s} , 4 for the quaternions \mathbf{q} , 1 for the density parameter $\tilde{\sigma}$, 27 for spherical harmonic coefficients, 21 for spherical Gaussian coefficients, 7 for lobe parameters, and 21 for lobe direction parameters. Moreover, as mentioned earlier, our current implementation requires allocating a buffer of size $N_{max,P} \times N_{ray}$, where $N_{max,P}$ is the maximum number of primitives per ray per slab allowed, and N_{ray} the number of rays launched. Therefore, the limiting factor of our implementation in terms of memory mainly

depends on the allocation of these data.

5. Uncorrelated ray casting

One of the advantages of ray casting compared to splatting lies in the fact that multiple independent rays from different cameras can be rendered simultaneously. In ray casting, which is an image-order rendering method, rays are treated independently and can originate from various viewpoints. In contrast, splatting, which is an object-order method, projects primitives onto the image plane and then sorts them. This primitive processing benefits splatting when rendering an entire image, as it allows for quickly computing the color of each pixel by summing the contributions of primitives projected onto it. However, this approach loses its advantage when considering the color of a single ray, for instance. Thus, in practice, several applications can be considered based on this observation: training is done iteratively on individual images in 3D Gaussian Splatting, whereas our training can easily use batches of rays from different images. Furthermore, supervision may require casting independent rays if the supervision data is sparse, for instance, if we want to supervise ray depth using a point cloud representing the surface as supervision data [5]. In this case, our approach is more suitable than the splatting algorithm because it can natively handle uncorrelated rays from different viewpoints. Another application would be adaptive supersampling, which consists of successively casting rays in the image plane to reduce rendering artifacts by focusing on the most challenging regions. This type of approach is more suited to ray casting as improving the rendering quality may only require casting a few additional rays. These last two applications are beyond the scope of this article. Additionally, tests were conducted by training with batches of rays. However, training in batches does not allow for the use of supervision functions such as structural similarity (SSIM), and in practice, we obtained poorer results compared to training image by image. However, conducting more experiments to explore this aspect further would be interesting.

6. OptiX API applications

Our method is supported by Nvidia OptiX, a framework designed initially for GPU ray tracing. Its flexible API allows for efficiently combining different types of primitives, associated intersections, and rendering algorithms. Consequently, our approach has the potential to be combined with more traditional rendering methods using standard primitives, such as meshes rendered by classic ray tracing, through the API used. Thus, our approach could be integrated into complex environments mixing different types of primitives that can be rendered using the single OptiX API.

7. Analysis of Δt and σ_ϵ

In this section, we present a study of the influence of two parameters on the final rendering quality, training and rendering times:

- Δt is the distance between two samples along a ray
- σ_ϵ corresponds both to the density under which a Gaussian is removed but also to the limits of the Gaussians for the calculation of intersections

Tab. 1 studies the influence of the parameter Δt on the Blender dataset. The results come from training on the Blender Dataset with PSNR averaged over the 8 scenes of this synthetic dataset. The gray line corresponds to the choice of the parameter Δt for all other experiments (main article and supplementary). We observe that increasing the space Δt makes it possible to speed up training but also rendering times at the cost of a reduction in graphic quality.

Δt	PSNR \uparrow	Training Time	Rendering Time
0.00125	34.53	46 min	17.4 FPS
0.0025	34.53	32 min	25.8 FPS
0.005	34.52	24 min	42.8 FPS
0.01	33.90	20 min	50.1 FPS

Table 1. **Study of the influence of the parameter Δt on the Blender dataset.** Δt is the distance between two samples along a ray. Values are averaged per scene. In gray, the parameter used in other experiments. Rendering time is for 800x800 pixels image.

Tab. 2 studies the influence of the parameter σ_ϵ on the Blender dataset. The results come from training on the Blender Dataset with PSNR averaged over the 8 scenes of this synthetic dataset. We can see that increasing σ_ϵ speeds up the training and rendering times of the method. On the contrary, by decreasing σ_ϵ , we increase the size of the Gaussians when calculating the intersections with the rays, which increases the training and rendering times, while improving the quality of rendering. The gray line also corresponds to the choice of the parameter σ_ϵ for all other experiments on Blender dataset (main article and supplementary), for Mip-NeRF360, we use $\sigma_\epsilon = 0.01$.

σ_ϵ	PSNR \uparrow	Training Time per scene	Rendering Time
1.0	34.52	26 min	29.4 FPS
0.1	34.53	32 min	25.8 FPS
0.01	34.54	35 min	23.5 FPS

Table 2. **Study of the influence of the parameter σ_ϵ on the Blender dataset.** σ_ϵ determines the limit of Gaussians for calculating intersections. Values are average per scene. In gray, the parameter used in other experiments. Rendering time is for 800x800 pixels image.

8. RayGauss and Anti-aliasing

Like NeRF [9], Instant-NGP [10] and Gaussian Splatting [15], RayGauss is a method that does not have an anti-aliasing mechanism, unlike the Mip-NeRF [1] and Mip-Splatting [15] methods.

To study the level of aliasing of RayGauss, we followed the protocol defined by Mip-Splatting [15] Single-Scale vs Multi-Scale with the Blender dataset, which consists of training the methods with full resolution images (800x800 pixels) then testing the rendered at different resolutions (1, $1/2$, $1/4$, $1/8$) to mimic zoom-out effects. We can see the results with the PSNR metric on Tab. 3. RayGauss manages to maintain good rendering quality on all scales compared to Gaussian Splatting (due to rasterization and dilation of Gaussians in 2D) and remains competitive compared to methods with anti-aliasing (Mip-NeRF [1] and Mip-Splatting [15]).

	Full Res.	$1/2$ Res.	$1/4$ Res.	$1/8$ Res.	Avg.
Gaussian Splatting [15]	33.33	26.95	21.38	17.69	24.84
NeRF [9]	31.48	32.43	30.29	26.70	30.23
Instant-NGP [10]	33.09	33.00	29.84	26.33	30.57
MipNeRF [1]	33.08	33.31	30.91	27.97	31.31
Mip-Splatting [15]	33.36	34.00	31.85	28.67	31.97
RayGauss (ours)	34.53	33.90	30.01	26.36	31.20

Table 3. **PSNR score for Single-scale Training and Multi-scale Testing on the Blender dataset.** All methods are trained on full-resolution images (800x800 pixels) and evaluated at four different resolutions (800x800, 400x400, 200x200 and 100x100 pixels), lower resolutions simulating zoom-out effects.

A brute-force anti-aliasing method consists in multiplying the number of rays per pixel. The basic RayGauss method launches a single ray through the center of each pixel for training and rendering. We studied the effect of casting 4 rays per pixel for training and rendering, a variant called RayGauss4x. Training and rendering times are approximately 3 times longer than the basic RayGauss method. We then calculated the PSNR scores on Blender with several scales (Tab. 4) and compared it with the Mip-Splatting method. To be fair, we also increased the rasterization resolution of Mip-splatting by 4 at each scale (during training and rendering), a variant called Mip-Splatting4x. In this configuration, RayGauss4x is superior to Mip-Splatting4x on almost all scales on the Blender dataset.

9. Detailed results

Tab. 5 and Tab. 6 show the detailed results of the main paper with metrics PSNR, SSIM, and LPIPS on the Blender and Mip-NeRF 360 datasets. Some methods have no available code, so we were not able to report information about SSIM and LPIPS (for example, for PointNeRF++).

	Full Res.	$1/2$ Res.	$1/4$ Res.	$1/8$ Res.	Avg.
Mip-Splatting4x [15]*	33.51	35.23	35.71	33.92	34.59
RayGauss4x (ours)	34.60	36.43	36.02	33.11	35.04

Table 4. **PSNR score for Single-scale Training and Multi-scale Testing on the Blender dataset with 4x Super Sampling.** Methods are trained on full-resolution images (800x800 pixels) with 4x supersampling and evaluated at four different resolutions (800x800, 400x400, 200x200 and 100x100 pixels) with 4x supersampling, lower resolutions simulating zoom-out effects.

All methods with an * have been retrained using the available code:

- NeRF and Instant-NGP with their respective model using the nefstudio framework v1.1.3: <https://github.com/nefstudio-project/nefstudio>
- Gaussian Splatting: <https://github.com/graphdeco-inria/gaussian-splatting>
- Mip-Splatting: <https://github.com/autonomousvision/mip-splatting>

References

- [1] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5835–5844, 2021. 6
- [2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5460–5469, 2022. 7
- [3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Zip-nerf: Anti-aliased grid-based neural radiance fields. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 19640–19648, 2023. 7, 8
- [4] Zhang Chen, Zhong Li, Liangchen Song, Lele Chen, Jingyi Yu, Junsong Yuan, and Yi Xu. Neurf: A neural fields representation with adaptive radial basis functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4182–4194, October 2023. 7
- [5] Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan. Depth-supervised nerf: Fewer views and faster training for free. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12872–12881, 2022. 5
- [6] Eric Haines, Johannes Günther, and Tomas Akenine-Möller. *Precision Improvements for Ray/Sphere Intersection*, pages 87–94. Apress, Berkeley, CA, 2019. 2

	PSNR \uparrow								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Avg.
NeRF [9]	34.17	25.08	30.39	36.82	33.31	30.03	34.78	29.30	31.74
Zip-NeRF [3]	34.84	25.84	33.90	37.14	34.84	31.66	35.15	31.38	33.10
Instant-NGP [10]	35.00	26.02	33.51	37.40	36.39	29.78	36.22	31.10	33.18
Mip-NeRF360 [2]	35.65	25.60	33.19	37.71	36.10	29.90	36.52	31.26	33.24
Point-NeRF [14]	35.40	26.06	36.13	37.30	35.04	29.61	35.95	30.97	33.30
Gaussian Splatting [7]*	35.85	26.22	35.00	37.81	35.87	30.00	35.40	30.95	33.39
Mip-Splatting [15]*	36.03	26.29	35.33	37.98	36.03	30.29	35.63	30.50	33.51
PointNeRF++ [13]	36.32	26.11	34.43	37.45	36.75	30.32	36.85	31.34	33.70
NeuRBF [4]*	36.54	26.38	35.01	38.44	37.35	34.12	36.16	31.73	34.47
RayGauss (ours)	37.20	27.14	35.11	38.30	37.10	31.36	38.11	31.95	34.53

	SSIM \uparrow								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Avg.
NeRF [9]	0.975	0.925	0.967	0.979	0.968	0.953	0.987	0.869	0.953
Zip-NeRF [3]	0.983	0.944	0.985	0.984	0.980	0.969	0.991	0.929	0.971
Instant-NGP [10]	-	-	-	-	-	-	-	-	-
Mip-NeRF360 [2]	0.983	0.931	0.979	0.982	0.980	0.949	0.991	0.893	0.961
Point-NeRF [14]	0.984	0.935	0.987	0.982	0.978	0.948	0.990	0.892	0.962
Gaussian Splatting [7]*	0.988	0.955	0.988	0.986	0.983	0.960	0.992	0.893	0.968
Mip-Splatting [15]*	0.988	0.956	0.988	0.987	0.984	0.962	0.992	0.900	0.970
Point-NeRF++ [13]	-	-	-	-	-	-	-	-	-
NeuRBF [4]*	0.988	0.944	0.987	0.987	0.986	0.979	0.992	0.925	0.974
RayGauss (ours)	0.990	0.960	0.988	0.988	0.986	0.969	0.995	0.914	0.974

	LPIPS \downarrow								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Avg.
NeRF [9]	0.026	0.071	0.032	0.030	0.031	0.047	0.012	0.150	0.050
Zip-NeRF [3]	0.017	0.050	0.015	0.020	0.019	0.032	0.007	0.091	0.031
Instant-NGP [10]	-	-	-	-	-	-	-	-	-
Mip-NeRF360 [2]	0.018	0.069	0.022	0.024	0.018	0.053	0.011	0.119	0.042
Point-NeRF [14]	0.023	0.078	0.022	0.037	0.024	0.072	0.014	0.124	0.049
Gaussian Splatting [7]*	0.011	0.037	0.011	0.017	0.015	0.034	0.006	0.118	0.031
Mip-Splatting [15]*	0.012	0.037	0.011	0.018	0.015	0.033	0.005	0.107	0.030
Point-NeRF++ [13]	-	-	-	-	-	-	-	-	-
NeuRBF [4]*	0.016	0.061	0.016	0.021	0.015	0.032	0.008	0.114	0.035
RayGauss (ours)	0.009	0.030	0.011	0.015	0.012	0.026	0.004	0.088	0.024

Table 5. **PSNR, SSIM and LPIPS (with VGG network) scores on the Blender dataset.** All methods are trained on the train set with full-resolution images (800x800 pixels) and evaluated on the test set with full-resolution images (800x800 pixels).

- [7] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4), jul 2023. 4, 7, 8
- [8] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015. 3
- [9] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 405–421, Cham, 2020. Springer International Publishing. 6, 7, 8
- [10] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4), jul 2022. 6, 7, 8
- [11] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4), jul 2010. 1
- [12] Johannes L. Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4104–4113, 2016. 4
- [13] Weiwei Sun, Eduard Trulls, Yang-Che Tseng, Sneha Sumbandam, Gopal Sharma, Andrea Tagliasacchi, and Kwang Moo Yi. PointNeRF++: A multi-scale, point-based Neural Radiance Field. *arXiv e-prints*, page arXiv:2312.02362, Dec.

	PSNR \uparrow									
	bonsai	counter	kitchen	room	bicycle	flowers	garden	stump	treehill	Avg.
NeRF [9]*	22.10	22.34	22.00	24.46	19.35	19.49	22.70	21.43	21.02	21.65
Instant-NGP [10]*	27.04	24.25	23.44	27.30	23.69	21.41	25.64	22.56	22.22	24.17
Gaussian Splatting [7]*	33.42	30.21	33.40	32.95	27.33	23.71	29.58	27.78	24.00	29.15
Mip-Splatting [15]*	33.44	30.43	34.30	33.30	27.62	23.79	29.78	27.89	24.25	29.42
Zip-NeRF [3]*	36.10	30.13	32.85	34.20	28.10	24.25	30.24	27.78	25.72	29.93
RayGauss (ours)	35.22	31.79	35.43	32.95	27.21	23.53	29.91	27.13	24.26	29.71

	SSIM \uparrow									
	bonsai	counter	kitchen	room	bicycle	flowers	garden	stump	treehill	Avg.
NeRF [9]*	0.652	0.690	0.658	0.815	0.371	0.462	0.653	0.482	0.506	0.588
Instant-NGP [10]*	0.923	0.769	0.736	0.920	0.658	0.604	0.829	0.563	0.611	0.735
Gaussian Splatting [7]*	0.970	0.942	0.972	0.963	0.856	0.730	0.924	0.833	0.734	0.880
Mip-Splatting [15]*	0.971	0.945	0.976	0.966	0.871	0.752	0.931	0.845	0.744	0.889
Zip-NeRF [3]*	0.978	0.932	0.951	0.965	0.865	0.754	0.918	0.829	0.769	0.885
RayGauss (ours)	0.978	0.958	0.976	0.971	0.859	0.742	0.929	0.810	0.748	0.886

	LPIPS \downarrow									
	bonsai	counter	kitchen	room	bicycle	flowers	garden	stump	treehill	Avg.
NeRF [9]*	0.127	0.217	0.207	0.119	0.360	0.320	0.161	0.326	0.387	0.247
Instant-NGP [10]*	0.076	0.207	0.199	0.094	0.315	0.308	0.143	0.212	0.389	0.216
Gaussian Splatting [7]*	0.037	0.062	0.029	0.052	0.121	0.238	0.056	0.142	0.230	0.107
Mip-Splatting [15]*	0.032	0.057	0.027	0.045	0.103	0.189	0.050	0.130	0.197	0.092
Zip-NeRF [3]*	0.021	0.059	0.034	0.037	0.113	0.168	0.061	0.145	0.158	0.088
RayGauss (ours)	0.024	0.042	0.024	0.036	0.110	0.183	0.051	0.156	0.187	0.090

Table 6. **PSNR, SSIM and LPIPS (with VGG network) scores on the Mip-NeRF 360 dataset.** All methods are trained and tested on downsampled images by a factor of 8.

2023. 7

- [14] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5428–5438, 2022. 7
- [15] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19447–19456, June 2024. 4, 6, 7, 8