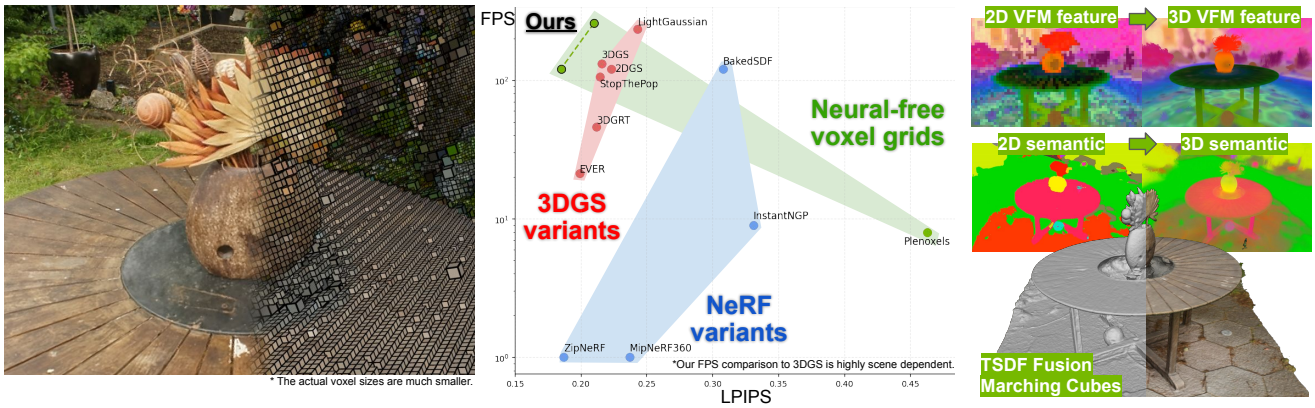
# Sparse Voxels Rasterization: Real-time High-fidelity Radiance Field Rendering

Cheng Sun[1] Jaesung Choe[1] Charles Loop[1] Wei-Chiu Ma[2] Yu-Chiang Frank Wang[1,3]
[1]NVIDIA [2]Cornell University [3]National Taiwan University




**(a) Volume rendering by rasterizing sparse voxels.** **(b) Novel-view rendering on Mip-NeRF360 scenes.** **(c) 2D-to-3D made easy.**

Figure 1. We propose **SVRaster**, a novel framework for multi-view reconstruction and novel view synthesis. **(a)** Sparse voxel representation effectively captures the volume density and radiance field of the scene, without the need for neural networks, 3D Gaussians, and sparse-points prior. **(b)** Using our customized sparse voxel rasterizer, we can learn the underlying 3D scene efficiently and achieve state-of-the-art performance in both rendering quality and speed. **(c)** Notably, lifting 2D modal to the trained sparse voxels is simple and efficient by integrating the classic Volume Fusion [7, 8, 35]. We show examples of vision foundation model feature field from RADIO [39], semantic field from Segformer [53], and signed distance field from rendered depth, making it flexible and suitable for a wide range of applications.

## Abstract

*We propose an efficient radiance field rendering algorithm that incorporates a rasterization process on adaptive sparse voxels without neural networks or 3D Gaussians. There are two key contributions coupled with the proposed system. The first is to adaptively and explicitly allocate sparse voxels to different levels of detail within scenes, faithfully reproducing scene details with $65536^3$ grid resolution while achieving high rendering frame rates. Second, we customize a rasterizer for efficient adaptive sparse voxels rendering. We render voxels in the correct depth order by using ray direction-dependent Morton ordering, which avoids the well-known popping artifact found in Gaussian splatting. Our method improves the previous neural-free voxel model by over 4db PSNR and more than 10x FPS speedup, achieving state-of-the-art comparable novel-view synthesis results. Additionally, our voxel representation is seamlessly compatible with grid-based 3D processing techniques such as Volume Fusion, Voxel Pooling, and Marching Cubes, enabling a wide range of future extensions and applications. Code: github.com/NVlabs/svraster*

## 1. Introduction

Gaussian splatting [20] has emerged as one of the most promising solutions for novel view synthesis. It has drawn wide attention across multiple communities due to its exceptional rendering speed and ability to capture the nuanced details of a scene. Nevertheless, Gaussian splatting in its base form has two key limitations: first, sorting Gaussians based on their centers does not guarantee proper depth ordering. It may result in popping artifacts [38] (*i.e.*, sudden color changes for consistent geometry) when changing views. Second, the volume density of a 3D point is ill-defined when covered by multiple Gaussians. This ambiguity makes surface reconstruction non-trivial.

On the other hand, grid representations inherently avoid these issues—both ordering and volume are well-defined. However, due to ray casting, the rendering speed of these methods [4, 5, 24, 44, 45, 52] is slower than that of Gaussian Splatting. This raises the question: is it possible to take the best of both worlds? Can we combine the efficiency of Gaussian splatting with the well-defined grid properties?

Fortunately, the answer is yes. Our key insight is to revisit voxels—a well-established primitive with decades of

history. Voxel representations are inherently compatible with modern graphics engines and can be rasterized efficiently. Additionally, previous work has demonstrated their ability to model scene volume densities, despite through volume ray casting. This positions voxels as the perfect bridge between rasterization and volumetric representation. However, naively adopting voxels does not work well [11]. Since a scene may consist of different levels of detail.

With these observations in mind, we present SVRaster, a novel framework that combines the efficiency of rasterization in 3DGS with the structured volumetric approach of grid-based representations. SVRaster leverages (1) multi-level sparse voxels to model 3D scenes and (2) implements a direction-dependent Morton order encoding that facilitates the rasterization rendering from our adaptive-sized sparse voxel representation. Our rendering is free from popping artifacts because the 3D space is partitioned into disjoint voxels, and our sorting ensures the correct rendering order. Moreover, thanks to the volumetric nature and neural-free representation of SVRaster, our sparse voxel grid can be easily and seamlessly integrated with classical grid-based 3D processing algorithms.

We show that our method is training fast, rendering fast, and achieves novel-view synthesis quality comparable to the state-of-the-art. We also integrate Volume Fusion, Voxel Pooling, and Marching Cubes operations into our adaptive sparse voxels, which showcases promising results of mesh extraction and 2D foundation feature fusion.

## 2. Related Work

Differentiable volume rendering has made significant strides in 3D scene reconstruction and novel-view synthesis tasks. Neural Radiance Fields (NeRF) [31] laid the foundation for this progress by optimizing a volumetric function, parameterized by multi-layer perceptrons (MLPs) to encode both geometry and appearance through differential rendering. Subsequent studies focus on accelerating speed by decomposing large MLPs into grid-based representations, with a shallow MLP typically still being employed. Several grid representations have been explored—dense grids [44], factorized grids [5], tri-planes [4], and hash grids [33].

In particular, our method is closely related to sparse voxel grid representations. Sparse Voxel Octrees [22] and VDB trees [34] are commonly used to manage sparse voxels and facilitate rendering. In contrast, our sparse voxels are stored in a 1D array without advanced data structures. Our rasterizer with the proposed direction-dependent Morton order ensures correct rendering order. To model volumes in sparse leaf nodes, different strategies exist, such as low-resolution 3D grid [34, 51] or implicit neural field [25, 46]. Our leaf node is a single voxel with explicit density and color parameters, like Plenoxels [11]. Regarding voxel levels, previous methods allocate voxels to a target level [11, 25, 46, 56] or use a shallow tree [34, 51]. Our voxels can adaptively fit into different levels across the entire tree depth, maximizing flexibility and scalability.

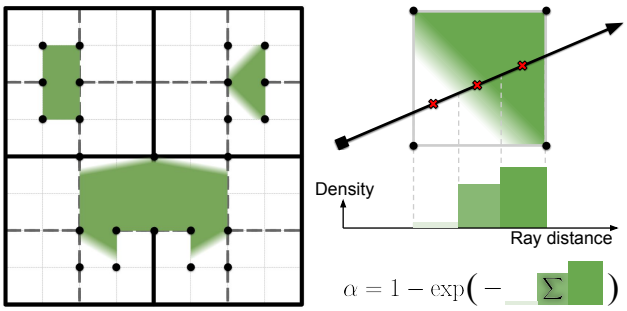
Another scalability issue of the previous grid-based methods is that they still use some sort of dense 3D grid in the field of novel-view synthesis. For instance, dense occupancy grids [5, 23, 33, 44] for free-space skipping or dense pointer grid [11] to support sparse voxel lookup. We do not use any 3D dense data structures.

3D Gaussian Splatting (3DGS) [20] takes a different approach by representing scenes with 3D Gaussian primitives and using rasterization for rendering, achieving state-of-the-art trade-off for quality and speed. Our work is inspired by the efficiency of using a rasterizer by 3DGS. However, 3DGS exhibits view-inconsistent popping artifacts due to inaccurate rendering order and primitive overlapping. Some recent works mitigate this artifact [32, 38], but completely resolving it significantly harms rendering speed [28]. Our method does not suffer from the popping artifact. Typically, 3DGS [20] initializes Gaussians from the triangulated points via COLMAP [42], which is especially critical for unbounded scenes. The sparse points geometry is also employed to guide the training of volumetric representations and has shown improvement in some NeRF-based methods [9, 12]. In this work, we do not use the sparse points.

Mesh reconstruction is an important extended topic for both volumetric-based and GS-based methods. Recent 3DGS variants [14, 16] with proper regularization have shown a good speed-accuracy trade-off. However, directly extracting meshes from 3DGS remains challenging, so these methods still rely on volumetric-based post-processing algorithms to extract meshes. NeRF variants [24, 45, 48, 49, 52, 54] improve surface quality by rendering a signed distance field (SDF) instead of density field. The well-defined volume makes it straightforward to extract a mesh from the isosurface. Our sparse voxels are also a volumetric representation. Though our voxel also follows volumetric representation focusing on the density field, we still can achieve promising accuracy with minimal time.

For efficient rendering, another line of research is to convert (or distill) the geometry and appearance of a high-quality but slow model into other efficient kind of representations such as smaller MLPs [40, 47], meshes [41, 50, 55], grids [10], or the recent Gaussians [36]. Typically, the powerful Zip-NeRF [3] is employed as the teacher model which requires hours of training per scene. Our volumetric representation could enable training-free conversion from a large implicit field, which is however out of our current scope.

Finally, the sparse voxel is also a widely-used representation for 3D processing [7, 8, 19, 26, 35, 37] and understanding [6, 13, 51]. We mainly focus on rendering in this work while we believe that adapting our method to these techniques is a promising future direction.




(a) (b)

Figure 2. **Sparse voxels scene representation. (Left)** We allocate voxel under an Octree layout. Each voxels has its own Spherical Harmonic coefficient for view-dependent appearance. The color field is approximated as a constant inside a voxel when rendering a view for efficiency. The density field is trilienarly varied inside a voxel and is modeled by the density values on the corner grid points (*i.e.*, the black dots •) of each voxel. The grid points densities are shared between adjacent voxels. **(Right)** We evenly sample $K$ points inside the segment of ray-voxel intersection to compute volume integration for its alpha value contributing to the pixel ray. See Sec. 3.1.1 for details.

# 3. Approach

We present our approach as follows. First in Sec. 3.1, we introduce our sparse voxels scene representation and our rasterizer for rendering sparse voxels into pixels. Later in Sec. 3.2, we describe the progressive scene optimization strategy, which is designed for faithfully reconstructing the scene from multi-view images using our sparse voxels.

## 3.1. Sparse Voxels Rasterization

Recent neural radiance field rendering approaches, such as NeRF [31] and 3DGS [20] variants, use the following alpha composition equation to render a pixel's color C:

$$\mathrm{C} = \sum_{i=1}^{N} T_i \cdot \alpha_i \cdot c_i \ , \quad T_i = \prod_{j=1}^{i-1} (1 - \alpha_j) \ , \quad (1)$$

where $\alpha_i \in \mathbb{R}_{\in[0,1]}$ and $c_i \in \mathbb{R}^3_{\in[0,1]}$ are alpha and view-dependent color at the $i$-th sampled point or primitive on the pixel-ray. The quantity $T_i$ is known as the transmittance. At a high-level, the difference between each method comes down to: *i)* how they find the $N$ points or primitives to composite a pixel-ray and *ii)* how the rgb and alpha values are determined from their scene representations.

Our sparse-voxel method follows the same principle. In Sec. 3.1.1, we provide details of our sparse voxels scene representation and how the $\alpha_i$ and $c_i$ in Eq. (1) are computed from a voxel. In Sec. 3.1.2, we present our rasterizer, that gathers the $N$ voxels to be composited for each pixel.

### 3.1.1. Scene Representation

We first describe the grid layout for allocating our sparse voxels and then derive the alpha value, view-dependent color, and other geometric properties needed for the composite rendering of a voxel.

**Sparse voxel grid.** Our SVRaster constructs 3D scenes using a sparse voxel representation. We allocate voxels following an Octree space partition rule (*i.e.*, **Octree layout** as illustrated in Fig. 2a) for two reasons necessary for achieving high-quality results. First, it facilitates the correct rendering order of voxels with various sizes (Sec. 3.1.2). Second, we can adaptively fit the sparse voxels to different scene level-of-details (Sec. 3.2). Note that our representation does not replicate a traditional **Octree data structure** with parent-child pointers or linear Octree. Specifically, we only keep voxels at the Octree leaf nodes without any ancestor nodes. Our sorting-based rasterizer will project voxels to image space and guarantee all voxels are in the correct order when rendering. In sum, we store individual voxels in arbitrary order without the need to maintain a more complex data structure, thanks to the flexibility provided by our rasterizer.

We choose a maximum level of detail $L$ ($= 16$ in this work) that defines a maximum grid resolution at $(2^L)^3$. Let $\mathbf{w}_\mathrm{s} \in \mathbb{R}$ be the Octree size and $\mathbf{w}_\mathrm{c} \in \mathbb{R}^3$ be the Octree center in the world space. The voxel index $v = \{i, j, k\} \in [0, \dots, 2^L\!-\!1]^3$ together with a Octree level $l \in [1, L]$ ($l = 0$ represent root node and is not used) define voxel size $\mathbf{v}_\mathrm{s}$ and voxel center $\mathbf{v}_\mathrm{c}$ as:

$$\mathbf{v}_\mathrm{s} = \mathbf{w}_\mathrm{s} \cdot 2^{-l} \ , \quad \mathbf{v}_\mathrm{c} = \mathbf{w}_\mathrm{c} - 0.5 \cdot \mathbf{w}_\mathrm{s} + \mathbf{v}_\mathrm{s} \cdot v \ . \quad (2)$$

Internally, we map the grid index to its Morton code using a well-known bit interleaving operation in the low-level CUDA implementation. Please see supplementary materials for more details.

**Voxel alpha from density field.** Next, we present details for the geometry and appearance modeling of each voxel primitive. For scene geometry, we use eight parameters corresponding to the voxel corners to model a trilinear density field inside each voxel, denoted as $\mathbf{v}_\mathrm{geo} \in \mathbb{R}^{2\times2\times2}$. Sharing corners among adjacent voxels results in a continuous density field.

We also need an activation function to ensure a non-negative density value for the raw density from $\mathbf{v}_\mathrm{geo}$. For this purpose, we use exponential-linear activation:

$$\mathrm{explin}(x) = \begin{cases} x & \text{if } x > 1.1 \\ \exp\left(\frac{x}{1.1} - 1 + \ln 1.1\right) & \text{otherwise} \end{cases} \ , \quad (3)$$

which approximates softplus but is more efficient to compute. For a sharp density field inside a voxel, we apply the non-linear activation after trilinear interpolation [18, 44].

To derive the alpha value of a voxel contributing to the alpha composition formulation in Eq. (1), we evenly sample $K$ points in the ray segment of ray-voxel intersection as depicted in Fig. 2b. The equation follows the numerical

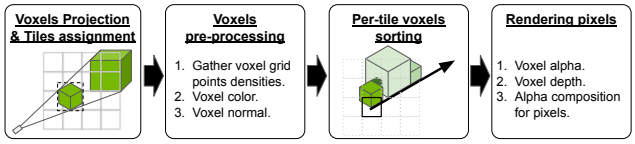



Figure 3. **Rasterization procedure.** Refer to Sec. 3.1.2 for details.

integration for volume rendering as in NeRF [29, 31]:

$$\alpha = 1 - \exp\left(-\frac{l}{K}\sum_{k=1}^{K}\text{explin}\left(\text{interp}\left(\mathbf{v}_{\text{geo}}, \mathbf{q}_k\right)\right)\right) , \quad (4)$$

where $l$ is the ray segment length, $\mathbf{q}_k$ is the local voxel coordinate of the $k$-th sample point, and $\text{interp}(\cdot)$ indicates trilinear interpolation.

**Voxel view-dependent color from SHs.** To model view-dependent scene appearance, we use $N_{\text{shd}}$ degree SH. For increased efficiency, we assume the SH coefficients stay constant inside a voxel, denoted as $\mathbf{v}_{\text{sh}} \in \mathbb{R}^{(N_{\text{shd}}+1)^2 \times 3}$. We approximate voxel colors as a function of the direction from the camera position $\mathbf{r}_{\text{o}}$ to the voxel center $\mathbf{v}_{\text{c}}$ instead of individual ray direction $\mathbf{r}_{\text{d}}$ for the sake of efficiency following 3DGS:

$$c = \max(0, \text{sh\_eval}(\mathbf{v}_{\text{sh}}, \text{normalize}(\mathbf{v}_{\text{c}} - \mathbf{r}_{\text{o}}))) , \quad (5)$$

which is the view-dependent color intensity of the voxel contributing to the pixel composition Eq. (1). Due to the approximation, the resulting SH color of a voxel can be shared by all covered pixels in the image rather than evaluating the SH for each intersecting ray.

**Voxel normal.** The rendering of other features or properties is similar to rendering a color image by replacing the color term $c$ in Eq. (1) with the target modality like the normal of a voxel density field. For rendering efficiency, we assume the normal stays constant inside a voxel, which is represented by the analytical gradient of the density field at the voxel center:

$$\vec{\mathbf{n}} = \text{normalize}\left(\nabla_{\mathbf{q}}\, \text{interp}\left(\mathbf{v}_{\text{geo}}, \mathbf{q}_{\text{c}}\right)\right) , \quad (6)$$

where $\mathbf{q}_{\text{c}} = (0.5, 0.5, 0.5)$ and the closed-form equations for forward and backward passes are in the supplementary material. Similar to the SH colors, the differentiable voxel normals are computed once in pre-processing and are shared by all the intersecting rays in the image.

**Voxel depth.** Unlike colors and normals, the point depths to composite is efficient to compute so we do the same $K$ points sampling as in the voxel alpha value in Eq. (4) for a more precise depth rendering. We manually expand and simplify the forward and backward computation for small number of $K$ in our CUDA implementation. Please refer to supplementary materials for details.

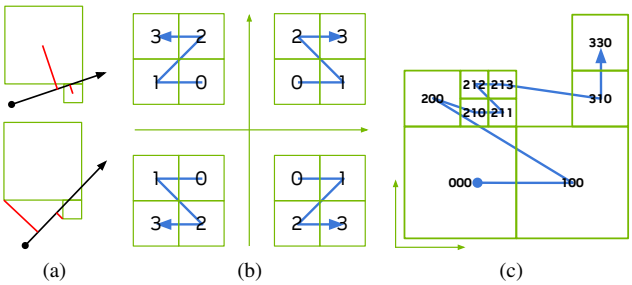



Figure 4. **Illustration of the rendering order. (a)** In both cases, the smaller voxels should be rendered first but they will arranged behind the larger voxels if using voxel centers or the nearest corners as the sorting order. **(b)** We show the four types of Morton order under the 2D world. The voxel rendering order under an Octree node is depend on which world quadrant the ray direction is pointing to. **(c)** An toy example of sorting the Morton order encoding. All the ray directions going toward the up-right quadrant can use the sorted voxels for a correct rendering order. See Sec. 3.1.2 for details.

### 3.1.2. Rasterization Algorithm

The overview of our sparse voxel rasterization procedure is depicted in Fig. 3. We build our sorting-based rasterizer based on the highly efficient CUDA implementation of 3DGS [20]. The procedure is detailed in the following.

**Projection to image space.** The first step of rasterization is projecting sparse voxels onto image space and assigning the voxels to the tiles (*i.e.*, image patches) they cover. In practice, we project the eight corner points of each voxel. The voxel is assigned to all tiles overlapped with the axis-aligned bounding box formed by the projected eight points.

**Pre-processing voxels.** For active voxels assigned to tiles of the target view, we gather their densities $\mathbf{v}_{\text{geo}}$ from the grid points, compute view-dependent colors from their spherical harmonic coefficients with Eq. (5), and derive voxel normals by Eq. (6). The pre-processed voxel properties are shared among all pixels during rendering.

**Sorting voxels.** For accurate rasterization, primitive rendering order is important. Similar to the challenge in 3DGS [20], using primitive centers or their closest distance to the camera can produce incorrect ordering, producing popping artifacts [38]. We show two incorrect ordering results using naive sorting criteria in Fig. 4a. Thanks to the Octree layout (Sec. 3.1.1), we can sort by Morton order using our sparse voxels representation. As illustrated in Fig. 4b, we can follow certain types of Morton order to render the voxels under an Octree node for correct ordering. The type of Morton order to follow is solely dependent on the positive/negative signs of the ray direction (the ray origin doesn't matter). That is to say, we have eight permutations of Morton order for different ray directions in the 3D space. Finally, the generalization to multi-level voxels can

be proved by induction. An ordering example in 2D with three levels is depicted in Fig. 4c.

The sorting is applied for each image tile. In case all the pixels in a tile share the same ray direction signs, we can simply sort the assigned voxels by their type of Morton order. We handle the corner case when multiple Morton orders are required in supplementary materials.

**Rendering pixels.** Finally, we proceed with alpha composition, Eq. (1), to render pixels. In our case, the $N$ primitives blend of a pixel-ray depends on the number of sparse voxels assigned to the tile that the pixel-ray belongs to. The computation of the alpha, color, and other geometric properties from our sparse voxels are described in Sec. 3.1.1. When rendering sparse voxels for a pixel-ray, we compute ray-aabb intersection to determine the ray segment to sample (for voxel alpha in Eq. (4)) and skip some non-intersected sparse voxels. We do early termination of the alpha composition if the transmittance of a sparse voxel is below a threshold $T_i{<}h_{\text{T}}$.

**Anti-aliasing.** To mitigate aliasing artifacts, we render in $h_{\text{ss}}$ times higher resolution and then apply image downsampling to the target resolution with an anti-aliasing filter.

## 3.2. Progressive Sparse Voxels Optimization

In this section, we describe the procedure to optimize a 3D scene from the input frames with known camera parameters using our SVRaster presented in Sec. 3.1.

**Voxel max sampling rate.** We first define the maximum sampling rate $\mathbf{v}_{\text{rate}}$ of each voxel on the training images, which reflects the image region a voxel can cover. A smaller $\mathbf{v}_{\text{rate}}$ indicates that the voxel is more prone to overfitting due to less observation. We use $\mathbf{v}_{\text{rate}}$ in our voxel initialization and subdivision process described later. Given $N_{\text{cam}}$ training cameras, we estimate the maximum sampling rate of a voxel as follows with visualization in Fig. 5a:

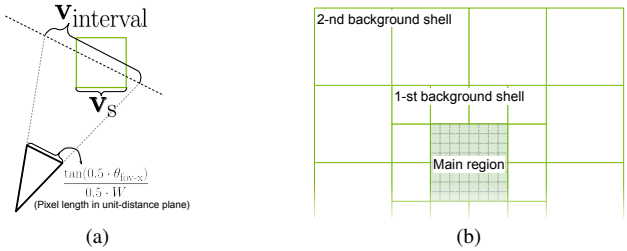
$$\mathbf{v}_{\text{rate}} = \max_i^{N_{\text{cam}}} \frac{\mathbf{v}_{\text{s}}}{\mathbf{v}_{\text{interval}}^{(i)}}, \tag{7a}$$

$$\mathbf{v}_{\text{interval}}^{(i)} = \underbrace{(\mathbf{v}_{\text{c}} - \mathbf{r}_{\text{o}}^{(i)})^\intercal \ell^{(i)}}_{\text{Voxel z-distance}} \cdot \underbrace{\frac{\tan\left(0.5 \cdot \theta_{\text{fov-x}}^{(i)}\right)}{0.5 \cdot W^{(i)}}}_{\text{Unit-distance pixel size}}, \tag{7b}$$

where $\ell$ is camera lookat vector, $\theta_{\text{fov-x}}$ is camera horizontal field of view, and $W$ is image width. The sampling rate indicates the estimated number of rays along the image's horizontal axis direction that may hit the voxel.

### 3.2.1. Scene Initialization

Without employing an additional prior, we initialize all the parameters to constant. We start with volume density approaching zero by setting voxel raw density to a negative number $h_{\text{geo}}$ so that the initial activated density $\text{exp}\text{lin}(h_{\text{geo}}){\approx}0$. We set the SH coefficients to zero for non-zero degrees and set the view-independent zero-degree component to yield gray color (*i.e.*, intensity equal 0.5). We detail the Octree grid layout initialization in the following.




(a) (b)

Figure 5. **Visualization of voxel sampling rate and grid layout initialization. (a)** We visualize the voxel sampling rate defined in Eq. (7). **(b)** We depict the foreground main region and the background region under different shell levels. In unbounded scenes, we apply different grid layout initialization strategies for foreground and background regions. See Sec. 3.2.1 for details.

**Bounded scenes.** In case the scenes or the objects to reconstruct are enclosed in a known bounded region, we simply initialize the layout as a dense grid with $h_{\text{lv}}$ levels and remove voxels unobserved by any training images. The number of voxels is $\leq (2^{h_{\text{lv}}})^3$ after initialization.

**Unbounded scenes.** For the unbounded scenes, we first split the space into the main and the unbounded background regions, depicted in Fig. 5b, each with a different heuristic. We use the training camera positions to determine a cuboid for the main region. The cuboid center is set to the average camera positions and the radius is set to the median distance between the cuboid center and the cameras. The same as the bounded scenes, we initialize a dense grid with $h_{\text{lv}}$ levels for the main region. For the background region, we allocate $h_{\text{out}}$ level of background shells enclosing the main region, which means that the radius of the entire scene is $2^{h_{\text{out}}}$ of the main region. In each background shell level, we start with the coarsest voxel size, *i.e.*, $4^3{-}2^3{=}56$ voxels in each shell level. We then iteratively subdivide shell voxels with the highest sampling rate and remove voxels unobserved by any training cameras. The process repeats until the ratio of the number of voxels in the background and the main regions is $h_{\text{ratio}}$. The number of voxels is $\leq (1 + h_{\text{ratio}})(2^{h_{\text{lv}}})^3$ after initialization.

### 3.2.2. Adaptive Pruning and Subdivision

The initialized grid layout only coarsely covers the entire scene that should be adaptively aligned to different levels-of-detail for the scene during the training progress. We apply the following two operations every $h_{\text{every}}$ training iterations to achieve this purpose.

**Pruning.** We compute the maximum blending weight $(T_i\alpha_i)$ from Eq. (1) of each voxel using all the training cameras. We remove voxels with maximum blending weight

lower than $h_{\text{prune}}$.

**Subdivision.** Our heuristic is that a voxel with a larger training loss gradient indicates that the voxel region requires finer voxels to model. Specifically, we accumulate the subdivision priority as the following:

$$\mathbf{v}_{\text{priority}} = \sum\nolimits_{\mathbf{r}\in R} \left\| \alpha(\mathbf{r}) \cdot \frac{\partial \mathcal{L}(\mathbf{r})}{\partial \alpha(\mathbf{r})} \right\| , \tag{8}$$

where $R$ is the set of all training pixel rays throughout the $h_{\text{every}}$ iterations and $\mathcal{L}(\mathbf{r})$ is the training loss of the ray. The gradient is weighted by alpha values contributed from the voxel to the ray. Higher $\mathbf{v}_{\text{priority}}$ indicates higher subdivision priority. To prevent voxels from overfitting few pixels, we set the priority to zero for voxels with maximum sampling rate lower than a sampling rate threshold $\mathbf{v}_{\text{rate}} < 2h_{\text{rate}}$. Finally, we select the voxels with priority above the top $h_{\text{percent}}$ percent to subdivide, *i.e.*, the total number of voxels is increased by $(h_{\text{percent}} \cdot (8-1))$ percent. Note that we only keep the leaf nodes in the Octree layout so we remove the source voxels once they are subdivided.

When voxels are pruned and subdivided, the voxel Spherical Harmonic (SH) coefficients and the grid point densities need to be updated accordingly. The SH coefficients are simply pruned together with voxels and duplicated to the subdivided children voxels. Grid point densities are slightly more complex as the eight voxel corner grid points are shared between adjacent voxels (Fig. 2). We remove a grid point only when it does not belong to any voxel corners. When subdividing, we use trilinear interpolation to compute the densities of the new grid points. The duplicated grid points are merged and their densities are averaged.

#### 3.2.3. Optimization objectives

We use MSE and SSIM as the photometric loss between the rendered and the ground truth images. The overall training objective is summarized as:

$$\begin{aligned}\mathcal{L} = \mathcal{L}_{\text{mse}} + \lambda_{\text{ssim}}\mathcal{L}_{\text{ssim}} \\ + \lambda_{\text{T}}\mathcal{L}_{\text{T}} + \lambda_{\text{dist}}\mathcal{L}_{\text{dist}} + \lambda_{\text{R}}\mathcal{L}_{\text{R}} + \lambda_{\text{tv}}\mathcal{L}_{\text{tv}} ,\end{aligned} \tag{9}$$

where $\lambda$ are the loss weights, $\mathcal{L}_{\text{T}}$ encourages the final ray transmittances to be either zero or one, $\mathcal{L}_{\text{dist}}$ is the distortion loss [2], $\mathcal{L}_{\text{R}}$ is the per-point rgb loss [44], and $\mathcal{L}_{\text{tv}}$ is the total variation loss on the sparse density grid. In mesh extraction task, we also add the depth-normal consistency loss from 2DGS [16]:

$$\mathcal{L}_{\text{mesh}} = \lambda_{\text{n-dmean}}\mathcal{L}_{\text{n-dmean}} + \lambda_{\text{n-dmed}}\mathcal{L}_{\text{n-dmed}} , \tag{10}$$

where both losses encourage alignment between rendered normals and depth-derived normals from mean and median depth. More details are in the supplementary materials.

#### 3.2.4. Sparse-voxel TSDF Fusion and Marching Cubes

Our sparse voxels can be seamlessly integrated with grid-based algorithms. To extract a mesh, we implement Marching Cubes [27] to extract the triangles of an isosurface over density from the sparse voxels. The duplicated vertices from adjacent voxels are merged to produce a unique set of vertices. When the adjacent voxels belong to different Octree levels, the extracted triangle may not be connected as the density field is not continuous for voxels in different levels. Such discontinuities can be removed by simply subdividing all voxels to the finest level.

Deciding the target level set for extracting the isosurface can be tricky for the density field. Instead, we implement sparse-voxel TSDF-Fusion [7, 8, 37] to compute the truncated signed distance values of the sparse grid points. We can then directly extract the surface of the zero-level set using the former sparse-voxel Marching Cubes. Future extensions of our method could directly model signed distance fields following NeuS [48] and VolSDF [54]. The sparse-voxel TSDF-Fusion can still be beneficial to directly initialize our sparse voxel representation from sensor depth.

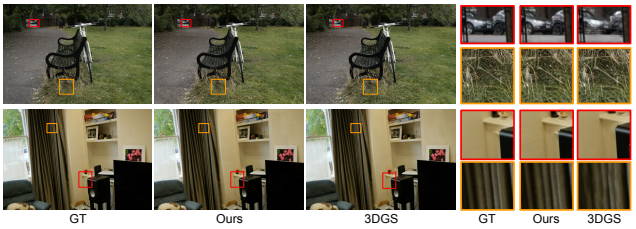



Figure 6. **A qualitative comparison with 3DGS [20].** Our result here corresponding to the base version in Tab. 1. We achieve similar visual quality comparing to 3DGS. Note that 3DGS use the coarse geometry from SfM while we do not rely on this prior.

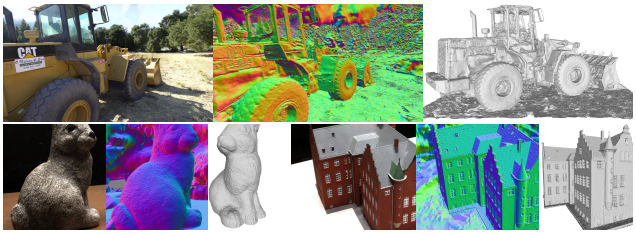


Figure 7. **Visualization of the reconstructed surface.** We show the rendering images, normal maps, and the final meshes on Tanks&Temples and DTU datasets. Note that we only model the scene with density field and do not use the coarse geometry prior from SfM sparse points in this work.

## 4. Experiments

### 4.1. Implementation Details

We use the following implementation details except stated otherwise. We start the optimization from empty space with raw density set to $h_{\text{geo}}{=}{-}10$. The initial Octree level is $h_{\text{lv}}{=}6$ (*i.e.*, $64^3$ voxels) for the bounded scenes and the foreground main region of the unbounded scenes. To model

| Method | Mip-NeRF360 dataset | | | | |
|---|---|---|---|---|---|
| | FPS↑ | Tr. Time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
| NeRF [31] | <1 | ∼day | 0.451 | 23.85 | 0.605 |
| M-NeRF [1] | <1 | ∼day | 0.441 | 24.04 | 0.616 |
| M-NeRF360 [2] | <1 | ∼day | 0.237 | 27.69 | 0.792 |
| Zip-NeRF [3] † | <1 | ∼hrs | 0.187 | **28.55** | **0.828** |
| Plenoxels [11] | <10 | ∼30m | 0.463 | 23.08 | 0.626 |
| INGP [33] | ∼10 | ∼5m | 0.302 | 25.68 | 0.705 |
| 3DGS [20] † | 131 | 24m | 0.216 | 27.45 | 0.815 |
| **Ours fast-rend** | **258** | 9m | 0.210 | 26.87 | 0.804 |
| **Ours fast-train** | 131 | **4.5m** | 0.199 | 27.08 | 0.816 |
| **Ours** | 121 | 15m | **0.185** | 27.33 | 0.822 |

† Re-evaluate on our machine.

Table 1. **Novel-view synthesis results comparison on Mip-NeRF360 dataset [2].** The results are averaged from 4 indoor scenes and 5 outdoor scenes. 3DGS uses the sparse points prior from COLMAP [42], whereas the other methods and ours do not.

| Method | Tanks&Temples | | | Deep Blending | | |
|---|---|---|---|---|---|---|
| | FPS↑ | LPIPS↓ | PSNR↑ | FPS↑ | LPIPS↓ | PSNR↑ |
| Plenoxels [11] | 13 | 0.379 | 21.08 | 11 | 0.510 | 23.06 |
| INGP [33] | 14 | 0.305 | 21.92 | 3 | 0.390 | 24.96 |
| M-NeRF360 [2] | <1 | 0.257 | 22.22 | <1 | 0.245 | 29.40 |
| 3DGS [20] † | **180** | 0.176 | **23.75** | 140 | 0.244 | 29.60 |
| **Ours** | 125 | **0.144** | 23.04 | **366** | **0.228** | **29.84** |

† Re-evaluate on our machine.

Table 2. **Comparison on Tanks&Temples [21] and Deep Blending [15] datasets.** We follow 3DGS [20] to use two outdoor scenes from Tanks&Temples and two indoor scenes from Deep Blending.


Figure 8. **Failuare case.** On scenes with severe exposure variation of training views, our method struggles and produces clear boundary of different brightnesses and allocates many floaters. 3DGS on the other hand is less sensitive to photometric variation of GT. This explains our worse PSNR and FPS on Tanks&Temples in Tab. 2.

unbounded scenes, we use $h_{\text{out}}$=5 background shell levels with $h_{\text{ratio}}$=2 times the number of foreground voxels (Sec. 3.2.1 and Fig. 5). The early ray stopping threshold is set to $h_{\text{T}}$=1e−4 and the supersampling scale is set to $h_{\text{ss}}$=1.5. Inside each voxel, we sample $K$=1 point for novel-view synthesis and $K$=3 points for mesh reconstruction task. We train our model for 20K iterations. The voxels are pruned and subdivided every $h_{\text{every}}$=1,000 iterations. The pruning threshold $h_{\text{prune}}$ is linearly scale from 0.0001 to 0.05 and the subdivision percentage is $h_{\text{percent}}$=5. More details are provided in the supplement.

| Method | Peak GPU mem.↓ | Model size↓ | FPS at higher res.↑ | | |
|---|---|---|---|---|---|
| | | | 1x | 2x | 3x |
| 3DGS | **1.8GB** | **0.7GB** | **131** | 69 | 39 |
| Ours | 3.9GB | 1.8GB | 121 | **103** | **69** |

Table 3. **Memory, model size, and high-res FPS.** The results are averaged on Mip-NeRF360 dataset. The standard 1x evaluation resolution have about 1–1.6M pixels per frame.

We use the standard LPIPS [57], SSIM, and PSNR metrics to evaluate novel-view quality. We directly employ the LPIPS implementation from prior work [20], which uses $[0, 1]$ image intensity range with VGG [43] network. For mesh accuracy, we follow the benchmarks to use F-score and chamfer distance. We use the test-set images to measure the FPS on a desktop computer with a 3090 Ti GPU.

### 4.2. Novel-view Synthesis

In Tab. 1, we show the quantitative comparison on the MipNeRF-360 dataset. We also provide a fast-rendering variant with >2x FPS by setting $h_{\text{ss}}$=1.5, $h_{\text{prune}}$=0.15, and a fast-training variant with >3x training speedup by scaling all iteration related hyperparameter by 0.3. Both fast variants only reduce quality moderately.

Our rendering speed is comparable to 3DGS on average. However, FPS varies significantly across scenes (see supplementary for detailed per-scene results). Several differences impact speed between our method and 3DGS. For instance, 3DGS sorts 32-bit floats, while we sort 48-bit Morton codes (16 levels, each with 3 bits) for primitive ordering. We also avoid the overhead of computing inverse covariance matrices from quaternions and scaling parameters. We directly intersect rays with voxels in 3D space, whereas 3DGS approximates the projection of 3D Gaussians to 2D via a linear affine transform. Finally, Gaussian distributions decay gradually to zero, while our post-activation [18, 44] voxel density field can be arbitrarily sharp. As a result, the average numbers of primitives contributing to a pixel intensity is 63 for 3DGS and 27 for us. The influence of these factors on rendering speed is scene-dependent.

Regarding rendering quality metrics, our method achieves significantly better LPIPS than 3DGS and even surpasses Zip-NeRF. This can be explained by the qualitative comparison in Fig. 6 where our method recover finer details. The SSIM comparison results with 3DGS is more scene-dependent. SSIM comparisons with 3DGS are more scene-dependent, while PSNR tends to favor 3DGS, as it prefers smoother rendering in uncertain regions. More visual comparisons are in the supplementary.

Our method uses much more voxel primitives than Gaussian, resulting in larger model size and requires more GPU memory as shown in Tab. 3. Interestingly, our high FPS is

| Resolution of main | $256^3$ | $512^3$ | $1024^3$ | adaptive | Plenoxels $640^3$ |
|---|---|---|---|---|---|
| LPIPS↓ | 0.444 | 0.326 | | **0.200** | 0.452 |
| PSNR↑ | 23.98 | 25.37 | OOM | **28.01** | 23.29 |
| FPS↑ | **457** | 190 | | 171 | <10 |

Table 4. **Ablation experiments of adaptive and uniform voxel sizes.** The results are evaluated on the indoor *bonsai* and the outdoor *bicycle* scenes from MipNeRF-360 dataset. The resolutions at the first row indicate the final grid resolution of the main foreground cuboid. Plenxoels is the previous fully-explicit voxel grid approach. Please refer to Sec. 4.3 for more discussion.

more scalable to higher resolution rendering perhaps due to the fewer contributing primitives per pixel.

We compare results on two more datasets in Tab. 2. On Deep Blending [15] dataset, we achieve much better speed and quality than 3DGS. On Tanks&Temples dataset [21], despite having better LPIPS, our PSNR and FPS are worse than 3DGS. The failure results are shown in Fig. 8 where the training views have large exposure variation. As a result, our method produces a clear brightness boundary in the scene with many floaters that slow down our rendering.

### 4.3. Ablation Studies

The rasterization process improves the rendering speed of sparse voxels, while the key to achieve high-quality results is the adaptive voxel size for different levels of detail. We show an ablation experiment in Tab. 4. Instead of subdividing voxels adaptively, the uniform voxel size variant subdivides all the voxels at certain training iterations until the grid resolution of the foreground main region reaches $256^3$, $512^3$, or $1024^3$. We also align the background and the pruning setup where the details are deferred to the supplementary. As shown in the table, the quality of uniform voxel size in $512^3$ resolution is much worse than the adaptive voxel size under similar FPS. Our machine with 24GB GPU memory fails to reach $1024^3$ grid resolution despite the voxels being pruned and sparse. Plenoxels [11] renders sparse voxels by ray casting and sampling following NeRF. Their FPS is significantly lower than our rasterization-based sparse voxel rendering. As they use a dense 3D pointer grid to support point sampling on rays, the model scalability for the unbounded background region is thus limited. To workaround, they apply a multi-sphere image with 64 layers to model the background. Conversely, we set the scene size as 32x larger ($h_{\text{out}}$=5) than the foreground cuboid and direct model the entire scene by our sparse voxels. As a result, our uniform size variant with $512^3$ foreground resolution still outperforms Plenoxels with $640^3$ grid.

In our rasterizer, we use direction-dependent Morton order to ensure the sorting of voxels from different levels is correct. However, like the case in 3DGS, the popping artifact by the incorrect order is not a major factor in the numerical results. Instead, we provide more visualization in the supplementary videos to show its effect. See more ablation studies in the supplementary materials.

| | | | Tansk&Temples | | DTU | |
|---|---|---|---|---|---|---|
| Method | Geom. | Init. | F-score↑ | Tr. time↓ | Cf.↓ | Tr. time↓ |
| NeRF [31] | Density | Random | - | - | 1.49 | hrs |
| NeuS [48] | SDF | Sphere | 0.38 | hrs | 0.84 | hrs |
| Voxurf [52] | SDF | Sphere | - | - | 0.76 | 15m |
| Neuralangelo [24] | SDF | Sphere | **0.50** | hrs | **0.61** | hrs |
| 3DGS [20] | Gaussians | SfM pts. | 0.19 | 14m | 1.96 | 11m |
| SuGaR [14] | Gaussians | SfM pts. | 0.09 | ∼1h | 1.33 | ∼1h |
| 2DGS [16] | Gaussians | SfM pts. | 0.32 | 16m | 0.80 | 11m |
| **Ours** | Density | Constant | 0.40 | **11m** | 0.76 | **5m** |

Table 5. **Mesh results comparison on the Tanks&Temples [21] and DTU [17] datasets.** Our method with volume density and constant field initialization achieves good accuracy-time trade-off on the unbounded-level and object-level datasets.

### 4.4. Mesh Reconstruction

We extract mesh by adapting TSDF-Fusion [35] and Marching Cubes [27] into our sparse voxels. The results comparison on the large-scale Tanks&Temples and the object-scale DTU datasets is provided in Tab. 5, where our method achieves a good balance of accuracy and training time on both datasets. See qualitative examples in Fig. 7 and the supplementary materials. We find that our current method tends to produce unnecessary geometric bumps for texture details. Future work could improve surface smoothness by adapting our density field for direct signed distance field modeling or incorporating SfM sparse points prior.

### 4.5. 2D Feature Fusion

We showcase some results in Fig. 1, where we fuse 2D semantic segmentation and high-dimensional foundation features into our sparse voxels using Voxel Pooling and Volume Fusion. The multi-view ensemble smooths the inconsistent 2D predictions, while our detailed 3D geometry enables higher-resolution rendering of 2D features. See more results in the released code.

## 5. Conclusion

This work presents a novel differentiable radiance field rendering system that integrates an efficient rasterizer with a multi-level sparse voxel scene representation. We reconstruct a scene from the multi-view input images by adaptively fitting the sparse voxels into different levels-of-detail. The results reveal that fully explicit voxel models, without neural networks or Gaussians, can achieve state-of-the-art comparable novel-view rendering speed and quality. The key breakthrough is overcoming the scalability constraint of voxel grids with adaptive sparse voxels and improving rendering speed through rasterization. We believe that incorporating our method with classical 3D processing algorithm [7, 8, 19, 26, 37] and 3D neural network [6, 13, 51] are promising future directions.

**Acknowledgements.** We thank Min-Hung Chen and Yueh-Hua Wu for helpful paper proofreading.

**Updates.**

- Mar 2025: Revise literature review.
- Feb 2025: (1) Code release. (2) Add more volume fusion examples. (3) Novel-view synthesis quality improved. (4) Provide fast rendering and fast training variants. (5) Discuss results more.
- Dec 2024: Paper release on arxiv.

## 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 ICCV, 2021. 7

[2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In CVPR, 2022. 6, 7, 15, 17, 18, 19

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Zip-nerf: Anti-aliased grid-based neural radiance fields. In ICCV, 2023. 2, 7

[4] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas J. Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3d generative adversarial networks. In CVPR, 2022. 1, 2

[5] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In ECCV, 2022. 1, 2, 16

[6] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In CVPR, 2019. 2, 8

[7] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. ACM TOG, 1996. 1, 2, 6, 8

[8] Angela Dai, Matthias Nießner, Michael Zollöfer, Shahram Izadi, and Christian Theobalt. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. ACM TOG, 2017. 1, 2, 6, 8

[9] Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan. Depth-supervised nerf: Fewer views and faster training for free. In CVPR, 2022. 2

[10] Daniel Duckworth, Peter Hedman, Christian Reiser, Peter Zhizhin, Jean-François Thibert, Mario Lucic, Richard Szeliski, and Jonathan T. Barron. SMERF: streamable memory efficient radiance fields for real-time large-scene exploration. ACM TOG, 2024. 2

[11] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In CVPR, 2022. 2, 7, 8, 16

[12] Qiancheng Fu, Qingshan Xu, Yew Soon Ong, and Wenbing Tao. Geo-neus: Geometry-consistent neural implicit surfaces learning for multi-view reconstruction. In NeurIPS, 2022. 2

[13] Benjamin Graham and Laurens Van der Maaten. Submanifold sparse convolutional networks. arXiv preprint arXiv:1706.01307, 2017. 2, 8

[14] Antoine Guédon and Vincent Lepetit. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. In CVPR, 2024. 2, 8

[15] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel J. Brostow. Deep blending for free-viewpoint image-based rendering. ACM TOG, 2018. 7, 8, 19

[16] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2d gaussian splatting for geometrically accurate radiance fields. ACM TOG, 2024. 2, 6, 8

[17] Rasmus Ramsbøl Jensen, Anders Lindbjerg Dahl, George Vogiatzis, Engin Tola, and Henrik Aanæs. Large scale multi-view stereopsis evaluation. In CVPR, 2014. 8, 15, 17, 18, 19

[18] Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy J. Mitra. Relu fields: The little non-linearity that could. In ACM TOG, 2022. 3, 7, 11

[19] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In SGP, 2006. 2, 8

[20] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. ACM TOG, 2023. 1, 2, 3, 4, 6, 7, 8, 13, 18, 19

[21] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: benchmarking large-scale scene reconstruction. ACM TOG, 2017. 7, 8, 15, 17, 19

[22] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In ACM I3D, 2010. 2

[23] Ruilong Li, Hang Gao, Matthew Tancik, and Angjoo Kanazawa. Nerfacc: Efficient sampling accelerates nerfs. In ICCV, 2023. 2

[24] Zhaoshuo Li, Thomas Müller, Alex Evans, Russell H. Taylor, Mathias Unberath, Ming-Yu Liu, and Chen-Hsuan Lin. Neuralangelo: High-fidelity neural surface reconstruction. In CVPR, 2023. 1, 2, 8

[25] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. In NeurIPS, 2020. 2

[26] Charles Loop, Qin Cai, Sergio Orts-Escolano, and Philip A Chou. A closed-form bayesian fusion equation using occupancy probabilities. In 3DV, 2016. 2, 8

[27] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In ACM TOG, 1987. 6, 8

[28] Alexander Mai, Peter Hedman, George Kopanas, Dor Verbin, David Futschik, Qiangeng Xu, Falko Kuester, Jonathan T. Barron, and Yinda Zhang. Ever: Exact volumetric ellipsoid rendering for real-time view synthesis. arXiv preprint arXiv:2410.01804, 2024. 2, 18

[29] Nelson L. Max. Optical models for direct volume rendering. IEEE TVCG, 1995. 4, 12

[30] Duane Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In ACM PACT, 2010. 13

[31] 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 ECCV, 2020. 2, 3, 4, 7, 8, 12, 17, 18, 19

[32] Nicolas Moenne-Loccoz, Ashkan Mirzaei, Or Perel, Riccardo de Lutio, Janick Martinez Esturo, Gavriel State, Sanja Fidler, Nicholas Sharp, and Zan Gojcic. 3d gaussian ray tracing: Fast tracing of particle scenes. ACM TOG, 2024. 2, 18

[33] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. ACM TOG, 2022. 2, 7

[34] Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. ACM TOG, 2013. 2

[35] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In IEEE ISMAR, 2011. 1, 2, 8

[36] Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ FPS. arXiv preprint arXiv:2403.13806, 2024. 2

[37] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. ACM TOG, 2013. 2, 6, 8

[38] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. Stopthepop: Sorted gaussian splatting for view-consistent real-time rendering. ACM TOG, 2024. 1, 2, 4, 18

[39] Mike Ranzinger, Greg Heinrich, Jan Kautz, and Pavlo Molchanov. Am-radio: Agglomerative vision foundation model reduce all domains into one. In CVPR, 2024. 1

[40] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In ICCV, 2021. 2

[41] Christian Reiser, Stephan J. Garbin, Pratul P. Srinivasan, Dor Verbin, Richard Szeliski, Ben Mildenhall, Jonathan T. Barron, Peter Hedman, and Andreas Geiger. Binary opacity grids: Capturing fine geometric detail for mesh-based view synthesis. ACM TOG, 2024. 2

[42] Johannes L. Schönberger, Enliang Zheng, Jan-Michael Frahm, and Marc Pollefeys. Pixelwise view selection for unstructured multi-view stereo. In ECCV, 2016. 2, 7

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015. 7

[44] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In CVPR, 2022. 1, 2, 3, 6, 7, 11, 16, 17

[45] Cheng Sun, Guangyan Cai, Zhengqin Li, Kai Yan, Cheng Zhang, Carl S. Marshall, Jia-Bin Huang, Shuang Zhao, and Zhao Dong. Neural-pbir reconstruction of shape, material, and illumination. In ICCV, 2023. 1, 2

[46] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles T. Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In CVPR, 2021. 2

[47] Haithem Turki, Vasu Agrawal, Samuel Rota Bulò, Lorenzo Porzi, Peter Kontschieder, Deva Ramanan, Michael Zollhöfer, and Christian Richardt. Hybridnerf: Efficient neural rendering via adaptive volumetric surfaces. In CVPR, 2024. 2

[48] Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. In NeurIPS, 2021. 2, 6, 8

[49] Yiming Wang, Qin Han, Marc Habermann, Kostas Daniilidis, Christian Theobalt, and Lingjie Liu. Neus2: Fast learning of neural implicit surfaces for multi-view reconstruction. In ICCV, 2023. 2

[50] Zian Wang, Tianchang Shen, Merlin Nimier-David, Nicholas Sharp, Jun Gao, Alexander Keller, Sanja Fidler, Thomas Müller, and Zan Gojcic. Adaptive shells for efficient neural radiance field rendering. ACM TOG, 2023. 2

[51] Francis Williams, Jiahui Huang, Jonathan Swartz, Gergely Klár, Vijay Thakkar, Matthew Cong, Xuanchi Ren, Ruilong Li, Clement Fuji-Tsang, Sanja Fidler, Eftychios Sifakis, and Ken Museth. fvdb : A deep-learning framework for sparse, large scale, and high performance spatial intelligence. ACM TOG, 2024. 2, 8

[52] Tong Wu, Jiaqi Wang, Xingang Pan, Xudong Xu, Christian Theobalt, Ziwei Liu, and Dahua Lin. Voxurf: Voxel-based efficient and accurate neural surface reconstruction. In ICLR, 2023. 1, 2, 8

[53] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, José M. Álvarez, and Ping Luo. Segformer: Simple and efficient design for semantic segmentation with transformers. In NeurIPS, 2021. 1

[54] Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. Volume rendering of neural implicit surfaces. In NeurIPS, 2021. 2, 6

[55] Lior Yariv, Peter Hedman, Christian Reiser, Dor Verbin, Pratul P. Srinivasan, Richard Szeliski, Jonathan T. Barron, and Ben Mildenhall. Bakedsdf: Meshing neural sdfs for real-time view synthesis. In ACM TOG, 2023. 2

[56] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. Plenoctrees for real-time rendering of neural radiance fields. In ICCV, 2021. 2

[57] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In CVPR, 2018. 7

# Sparse Voxels Rasterization: Real-time High-fidelity Radiance Field Rendering

## Supplementary Material

We provide more details of our method and implementation in Secs. A to C. Some content overlaps with the main paper in the interest of being self-contained. More results and discussion are found in Secs. D and E.

## A. More Details of Our Representation

### A.1. Details of Sparse Voxels Grid

Recall that our SVRaster allocates voxels following an **Octree layout** but does not replicate a traditional **Octree data structure** with parent-child pointers or linear Octree. We only keep voxels at the Octree leaf nodes without any ancestor nodes and store individual voxels in arbitrary order without the need to maintain a more complex data structure.

The maximum level of detail is set to $L$=16 that defines the finest grid resolution at $65536^3$. Note that this is only for our CUDA-level implementation convenience. We leave it as future work to extend to an arbitrary number of levels as we find that 16 levels are adequate for the scenes we experimented with in this work.

Let $\mathbf{w}_\text{s} \in \mathbb{R}$ be the Octree size and $\mathbf{w}_\text{c} \in \mathbb{R}^3$ be the Octree center in the world space. The voxel index $v = \{i, j, k\} \in [0, \dots, 2^L\text{–}1]^3$ together with an Octree level $l \in [1, L]$ ($l = 0$ represent root node and is not used) define voxel size $\mathbf{v}_\text{s}$ and voxel center $\mathbf{v}_\text{c}$ as:

$$\mathbf{v}_\text{s} = \mathbf{w}_\text{s} \cdot 2^{-l}\,, \quad \mathbf{v}_\text{c} = \mathbf{w}_\text{c} - 0.5 \cdot \mathbf{w}_\text{s} + \mathbf{v}_\text{s} \cdot v\,. \tag{11}$$

Internally, we map the grid index to its Morton code by a well-known bit interleaving operation, which is helpful to implement our rasterizer detailed later. A Python pseudocode is provided in Listing 1.

### A.2. Details of Voxel Alpha from Density

A voxel density field is parameterized by eight parameters attached to its corners $\mathbf{v}_\text{geo} \in \mathbb{R}^{2\times2\times2}$, which is denoted as $\mathbf{V}$ for brevity in the later equations. We use the exponential-linear activation function to map the raw density to non-negative volume density. We visualize exponential-linear and Softplus in Fig. 9. Exponential-linear is similar to Softplus but more efficient to compute on a GPU. For a sharp density field inside a voxel, we apply the non-linear activation after trilinear interpolation [18, 44].

We evenly sample $K$ points in the ray segment of ray-voxel intersection to derive the voxel alpha value contributing to the pixel ray. First, we compute the ray voxel intersection point by Listing 2, which yields the ray distances $a$ and $b$ for the entrance and exit points along the ray with ray origin $\mathbf{r}_\text{o} \in \mathbb{R}^3$ and ray direction $\mathbf{r}_\text{d} \in \mathbb{R}^3$. The coordinate

```
MAX_NUM_LEVELS = 16

def to_octpath(i, j, k, lv):
    # Input
    #   (i,j,k): voxel index.
    #   lv:      Octree level.
    # Output
    #   octpath: Morton code
    octpath: int = 0
    for n in range(lv):
        bits = 4*(i&1) + 2*(j&1) + (k&1)
        octpath |= bits << (3*n)
        i = i >> 1
        j = j >> 1
        k = k >> 1
    octpath = octpath << (3*(MAX_NUM_LEVELS-lv))
    return octpath

def to_voxel_index(octpath, lv):
    # Input
    #   octpath: Morton code
    #   lv:      Octree level.
    # Output
    #   (i,j,k): voxel index.
    i: int = 0
    j: int = 0
    k: int = 0
    octpath = octpath >> (3*(MAX_NUM_LEVELS-lv))
    for n in range(lv):
        i |= ((octpath&0b100)>>2) << n
        j |= ((octpath&0b010)>>1) << n
        k |= ((octpath&0b001))    << n
        octpath = octpath >> 3
    return (i, j, k)
```

Listing 1. Pseudocode for conversion between voxel index and Morton code. See Sec. A.1 for details.

```
def ray_aabb(vox_c, vox_s, ro, rd):
    # Input
    #   vox_c: Voxel center position.
    #   vox_s: Voxel size.
    #   ro:    Ray origin.
    #   rd:    Ray direction.
    # Output
    #   a:     Ray enter at (ro + a * rd).
    #   b:     Ray exit at (ro + b * rd).
    #   valid: If ray hit the voxel.
    c0 = (vox_c - 0.5 * vox_s - ro) / rd
    c1 = (vox_c + 0.5 * vox_s - ro) / rd
    a = torch.minimum(c0, c1).max()
    b = torch.maximum(c0, c1).min()
    valid = (a <= b) & (a > 0)
    return a, b, valid
```

Listing 2. Pseudocode for intersecting ray and a axis-aligned voxel. See Sec. A.2 for details.

of $k$-th of the K sample points is:

$$t_k = a + \frac{k-0.5}{K} \cdot (b-a) \tag{12a}$$

$$\mathbf{p}_k = \mathbf{r}_\mathrm{o} + t_k \cdot \mathbf{r}_\mathrm{d} \tag{12b}$$

$$\mathbf{q}_k = (\mathbf{p}_k - (\mathbf{v}_\mathrm{c} - 0.5 \cdot \mathbf{v}_\mathrm{s})) \cdot \frac{1}{\mathbf{v}_\mathrm{s}} , \tag{12c}$$

where $\mathbf{p}_k \in \mathbb{R}^3$ is in the world coordinate and $\mathbf{q}_k \in \mathbb{R}^3_{\in[0,1]}$ is in the local voxel coordinate. The local coordinate $\mathbf{q}$ is used to sample voxel by trilinear interpolation:

$$\mathrm{interp}(\mathbf{V}, \mathbf{q}) = \begin{bmatrix} (1-\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(\mathbf{q}_z) \end{bmatrix}^\mathsf{T} \begin{bmatrix} \mathbf{V}_{000} \\ \mathbf{V}_{001} \\ \mathbf{V}_{010} \\ \mathbf{V}_{011} \\ \mathbf{V}_{100} \\ \mathbf{V}_{101} \\ \mathbf{V}_{110} \\ \mathbf{V}_{111} \end{bmatrix} , \tag{13}$$

where the subscript in this equation indicates the $x, y, z$ components of the vector $\mathbf{q}$ and the sample index is omitted. Following NeRF [29, 31], we use quadrature to compute the integrated volume density for alpha value:

$$\alpha = 1 - \exp\left( -\frac{l}{K} \sum_{k=1}^{K} \mathrm{explin}\left(v_k\right) \right) \tag{14a}$$

$$v_k = \mathrm{interp}\left(\mathbf{V}, \mathbf{q}_k\right) \tag{14b}$$

$$l = (b-a) \cdot \left\| \mathbf{r}_\mathrm{d} \right\| , \tag{14c}$$

where $l$ is the ray segment length. The gradient with respect to the voxel density parameters is:

$$\nabla_\mathbf{V}\, \alpha = (1-\alpha) \cdot \frac{l}{K} \cdot \sum_{k=1}^{K} \left( \frac{\mathrm{d}}{\mathrm{d}v_k} \mathrm{explin}(v_k) \cdot \nabla_\mathbf{V} v_k \right) . \tag{15}$$

## A.3. Details of Voxel Normal

Recall that we approximate the normal field as constant inside a voxel for efficiency, which is represented by the analytical gradient of the density field at the voxel center $\mathbf{q}^{(\mathrm{c})}$. Thanks to the neural-free representation, we derive closed-form equations for forward and backward passes instead of relying on double backpropagation of autodiff. The unnormalized voxel normal in the forward pass is:

$$\nabla_\mathbf{q} \mathrm{interp}(\mathbf{V}, \mathbf{q}^{(\mathrm{c})}) = 0.25 \cdot \begin{bmatrix} (\mathbf{V}_{100}+\mathbf{V}_{101}+\mathbf{V}_{110}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{001}+\mathbf{V}_{010}+\mathbf{V}_{011}) \\ (\mathbf{V}_{010}+\mathbf{V}_{011}+\mathbf{V}_{110}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{001}+\mathbf{V}_{100}+\mathbf{V}_{101}) \\ (\mathbf{V}_{001}+\mathbf{V}_{011}+\mathbf{V}_{101}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{010}+\mathbf{V}_{100}+\mathbf{V}_{110}) \end{bmatrix} \tag{16}$$

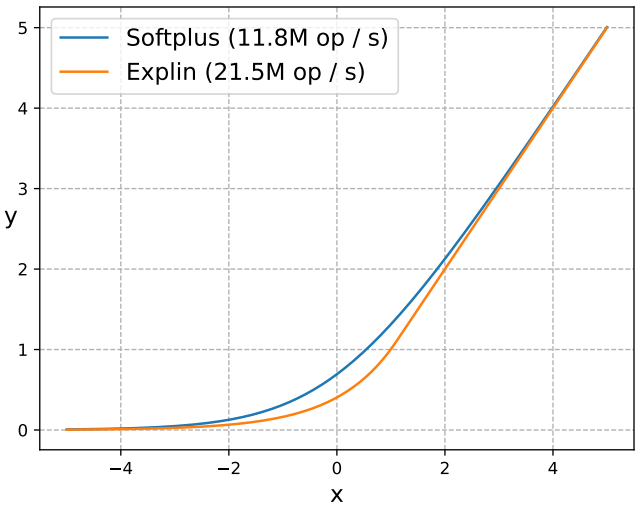



Figure 9. **Activation functions.** We use exponential-linear activation to softly map raw density to non-negative volume density. Exp-lin activation is about two times faster to compute in CUDA, which is 21.5M operations per second in a CUDA thread comparing to 11.8M of Softplus. See Sec. A.2 for more details.

For the backward pass, the gradient with respect to a density parameter is:

$$\nabla_{\mathbf{V}_{ijk}} \nabla_\mathbf{q} \mathrm{interp}(\mathbf{V}, \mathbf{q}^{(\mathrm{c})}) = 0.25 \cdot \begin{bmatrix} 2i-1 \\ 2j-1 \\ 2k-1 \end{bmatrix} . \tag{17}$$

## A.4. Details of Voxel Depth

Voxel depths are efficient to compute compared to view-dependent colors and normals so we do the same $K$ points sampling as in the voxel alpha value. Unlike colors and normals, which are approximated by constant inside each voxel, the depth values of each sample point inside a voxel are different so we need to incorporate the point depth in Eq. (12a) into the local alpha composition in Eq. (14). Let

$$\alpha_k = 1 - \exp\left( -\frac{l}{K} \cdot \mathrm{explin}\left(\mathrm{interp}(\mathbf{V}, \mathbf{q}_k)\right) \right) \tag{18}$$

be the alpha value of the $k$-th sampled point. The voxel local depth is:

$$d = \sum_{k=1}^{K} \left( \prod_{j=1}^{k-1} (1-\alpha_j) \right) \cdot \alpha_k \cdot t_k . \tag{19}$$

Finally, the pixel depth is composited by $\mathrm{D} = \sum_{i=1}^{N} T_i d_i$ from the $N$ voxels, where $T_i$ is the ray transmittance when reaching the $i$-th voxel described in the main paper.

We only experiment with $K \leq 3$ in this work, where the forward and backward equation of each case is summarized as follows. The $K{=}1$ is trivial with $d{=}\alpha_1 t_1$ and $\frac{\mathrm{d}d}{\mathrm{d}\alpha_1}{=}t_1$. In case $K{=}2$, the backward equations with voxel

depth $d{=}\alpha_1 t_1 + (1-\alpha_1)\alpha_2 t_2$ are:

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_1} = t_1 - \alpha_2 t_2, \quad \frac{\mathrm{d}d}{\mathrm{d}\alpha_2} = t_2 - \alpha_1 t_2 \,. \tag{20}$$

The voxel depth when $K{=}3$ is:

$$d = \alpha_1 t_1 + (1-\alpha_1)\alpha_2 t_2 + (1-\alpha_1)(1-\alpha_2)\alpha_3 t_3 \,. \tag{21}$$

The backward equations are:

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_1} = t_1 + \alpha_2\alpha_3 t_3 - \alpha_2 t_2 - \alpha_3 t_3 \tag{22a}$$

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_2} = t_2 + \alpha_1\alpha_3 t_3 - \alpha_1 t_2 - \alpha_3 t_3 \tag{22b}$$

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_3} = t_3 + \alpha_1\alpha_2 t_3 - \alpha_1 t_3 - \alpha_2 t_3 \,. \tag{22c}$$

# B. More Details of Voxel Rendering Order

Our sorting-based rasterizer is based on the efficient CUDA implementation done by 3DGS [20]. In the following, we first describe how the overall sorting pipeline works in Sec. B.1. We then dive more into the implementation of the direction-dependent Morton order in Sec. B.2 and its correctness proof in Sec. B.3.

A supplementary video is provided to show the effect of correct ordering and a few popping artifacts in comparison with 3DGS [20].

## B.1. Overview

The goal in the sorting stage of the rasterizer is to arrange a list of voxels in near-to-far order for each image tile. To this end, 3DGS's rasterizer duplicates a Gaussian for each image tile the Gaussian covers. A key-value pair is attached to each Gaussian duplication, where the tile index is assigned as the most significant bits of the sorting key. The bit field of the key-value pair is as follows:

$$\text{key} = |\underbrace{\text{tile id}}_{\text{32 bits}}|\underbrace{\text{Gaussian z-depth}}_{\text{32 bits}}| \tag{23a}$$

$$\text{value} = |\underbrace{\text{Gaussian id}}_{\text{32 bits}}| \tag{23b}$$

By doing so, all the duplicated Guassians assigned to the same image tile will be in the consecutive array segment after sorting with near-to-far z-depth ordering. In the later rendering stage, each pixel only iterates through the list of Gaussians of its tile for alpha composition.

In our case, we replace the primitive z-depth with a direction-dependent Morton order of voxels to ensure the rendering order is always correct. As there are eight different Morton orders to follow depending on the positive/negative signs of ray directions, dubbed *ray sign bits*, we further duplicate each voxel by the numbers of different ray sign bits it covers. The ray sign bits are also attached to each duplicated voxel. In the rendering stage, a pixel only composites voxels with the same attached ray sign bits when there are multiple ray sign bits in an image tile. Our bit field of the key-value pair is:

$$\text{key} = |\underbrace{\text{tile id}}_{\text{16 bits}}|\underbrace{\text{Morton order}}_{\text{48 (=3}L\text{) bits}}| \tag{24a}$$

$$\text{value} = |\underbrace{\text{ray sign bits}}_{\text{3 bits}}|\underbrace{\text{voxel id}}_{\text{29 bits}}| \tag{24b}$$

where $L{=}16$ is the maximum number of Octree levels. Note that the "voxel id" here is indexed to the 1D array location where we store the voxel. Not to be confused the grid $(i,j,k)$ index in Sec. A.1. The bit field arrangement is mainly for our implementation convenient to squeeze everything into 64 and 32 bits unsigned integers. In our current implementation, the maximum number of tiles is $2^{16}{=}65536$, which is $4096{\times}4096$ maximum image resolution with $16{\times}16$ tile size; the maximum grid resolution is $(2^{16})^3{=}65536^3$; the maximum number of voxels is $2^{29}{\approx}500M$. We find this is more than enough for the scenes in our experiments. Future work can define custom data types with extra bits for GPU Radix sort [30] to increase the resolution limit.

## B.2. Direction-dependent Morton Order

As described and illustrated in the main paper, there are eight types of Morton order to follow, each of which is for a certain type of positive/negative signs pattern of ray directions. We hard-code the eight types of Morton orders, which is used to remap every non-overlapping three bits (corresponding to different Octree levels) in the Octree Morton code of voxels (Sec. A.1):

$$\ldots\, b_x b_y b_z\; a_x a_y a_z \mapsto \ldots\, f^{(\mathrm{k})}(b_x b_y b_z)\; f^{(\mathrm{k})}(a_x a_y a_z) \,, \tag{25}$$

where $f^{(\mathrm{k})} : [0\cdots 7] \mapsto [0\cdots 7]$ is one of the eight permutation mappings. The pseudocode for computing the ray sign bits and the mapping function from Octree Morton code to direction-dependent Morton order is provided in Listing 3.

## B.3. Proof of Correct Ordering

We prove the ordering correctness by induction. We focus on the case for $(+,+,+)$ ray directions. The proof can be generalized to the other types of ray direction signs by flipping the scene. The Morton order of the eight voxels in the first Octree level is illustrated in Fig. 10.

Recap that our sparse voxels only consist of the Octree leaf nodes without any ancestor nodes. Let $V^\ell$ be the space of all valid sparse voxel sets with maximum Octree level equal to $\ell$. Let $S(\ell)$ be the statement that:

> "For all sparse voxel sets in $V^\ell$, their direction-dependent Morton order is always aligned with the near-to-far rendering order for all rays with $(+,+,+)$ direction signs."

```
MAX_NUM_LEVELS = 16
order_tables = [
    [0, 1, 2, 3, 4, 5, 6, 7],
    [1, 0, 3, 2, 5, 4, 7, 6],
    [2, 3, 0, 1, 6, 7, 4, 5],
    [3, 2, 1, 0, 7, 6, 5, 4],
    [4, 5, 6, 7, 0, 1, 2, 3],
    [5, 4, 7, 6, 1, 0, 3, 2],
    [6, 7, 4, 5, 2, 3, 0, 1],
    [7, 6, 5, 4, 3, 2, 1, 0],
]

def to_rd_signbits(rd):
    # Input
    #   rd:       Ray direction.
    # Output
    #   signbits: Ray sign bits.
    return 4*(rd[0]<0) + 2*(rd[1]<0) + (rd[2]<0)

def to_dir_dep_morton_order(octpath, signbits):
    # Input
    #   octpath:  Voxel Octree Morton code.
    #   signbits: The signbits the voxel care.
    # Output
    #   order:    The order for sorting.
    table = order_tables[signbits]
    order = 0
    for i in range(MAX_NUM_LEVELS):
        order |= table[octpath & 0b111] << (3*i)
        octpath = octpath >> 3
    return order
```

Listing 3. Pseudocode for direction-dependent Morton order. The mapping between voxel grid $(i, j, k)$ index and Octree Morton code `octpath` is detailed in Listing 1. In practice, the mapping from Octree Morton code to direction-dependent Morton order is done by a single bitwise xor operation instead of for-loop. More details in Sec. B.2.

**Base case.** When $\ell$=1, there is only one Octree level. The direction-dependent Morton order of the eight voxels for $(+,+,+)$ ray directions is illustrated in Fig. 10. The bit field from most to least significant bit is for $x$, $y$, and $z$ directions, respectively. As the ray is going toward $+x$ direction, we can always render the voxels in the $-x$ side $(000, 001, 010, 011)$ first before the voxels in the $+x$ side $(100, 101, 110, 111)$, which is aligned with the most significant bit of the Morton order. Similarly, for the voxels in the $-x$ side, we can render the voxels in the $-y$ side $(000, 001)$ before the $+y$ side $(010, 011)$ as the ray is going toward $+y$. Finally, we can see that the rendering order is correct if we iterate the voxels following the assigned Morton order for ray with $(+,+,+)$ directions.

**Induction hypothesis.** Assume that $S(\ell)$ is true for some positive integer $\ell$.

**Induction step.** We want to show $S(\ell) \implies S(\ell+1)$ is true. For any sparse voxel set $w \in V^{\ell+1}$, there exists a sparse voxel set $v \in V^{\ell}$ that can evolve into $w$ by: *i)* selecting a subset of voxels in $v$ to subdivide with the source

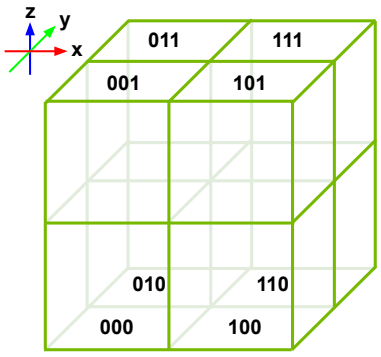



Figure 10. **Base case.** Direction-dependent Morton order for $(+,+,+)$ ray direction signs under the base case with 1 Octree level. The three bits from left to right is for the $x$, $y$, and $z$ directions respectively. The rendering order is correct for all rays going toward $(+,+,+)$ direction. See Sec. B.3 for more details.

voxels removed and *ii)* removing some of the voxels. The $S(\ell)$ indicates that the direction-dependent Morton order of $v$ has the correct rendering order. To extend for a new Octree level, three zero bits are first append to the least significant bit of the Morton order of every voxel in $v$, which does not affect the ordering. When subdividing a voxel, the eight child voxels share the same most significant $3\ell$ bits as the source voxel, while the least significant 3 bits follow the same direction-dependent Morton order as in the base case Fig. 10. This reflects the fact that the new child voxels should keep the same relative order to the other voxels as their source parent voxels as the child voxels are all in the 3D space of the source voxels. The rendering ordering of the eight child voxels can also follow the same Morton order as the base case. That is the Morton order is still rendering-order correct after subdividing some voxels in $v$. Finally, removing voxels does not affect the ordering of the remaining others. In sum, the Morton order of $w$ also has the correct rendering order so $S(\ell)$ implies $S(\ell+1)$. By induction, $S(\ell)$ is true for all positive integer $\ell$.

## C. Additional Implementation Details

We start the optimization from empty space with raw density set to $h_{\text{geo}}$=$-10$. We use spherical harmonic (SH) with $N_{\text{shd}}$=3 degrees. The learning rate is set to 0.025 for the grid point densities, 0.01 for zero-degree SH coefficients, and 0.00025 for higher-degree SH coefficients. We decay all learning rates by 0.1 at the 19K iteration. The momentum and the epsilon value of the Adam optimizer are set to $(0.1, 0.99)$ and 1e$-$15. The initial Octree level is $h_{\text{lv}}$=6 (*i.e.*, $64^3$ voxels) for the bounded scenes and the foreground main region of the unbounded scenes. To model unbounded scenes, we use $h_{\text{out}}$=5 background shell levels with $h_{\text{ratio}}$=2 times the number of foreground voxels. We use average frame color as the color coming from infi-

| Resolution of main | $256^3$ | $512^3$ | $1024^3$ | adaptive |
|---|---|---|---|---|
| LPIPS↓ | 0.444 | 0.326 | | **0.200** |
| PSNR↑ | 23.98 | 25.37 | OOM | **28.01** |
| FPS↑ | **457** | 190 | | 171 |

Table 6. **Ablation experiments of adaptive and uniform voxel sizes.** The resolutions at the first row indicate the final grid resolution of the main foreground cuboid. Note that OOM is abbreviation of the term, 'out-of-memory'.

nite far away for unbounded scenes. The early ray stopping threshold is set to $h_{\mathrm{T}}$=1e−4 and the supersampling scale is set to $h_{\mathrm{ss}}$=1.5. Inside each voxel, we sample $K$=1 point for novel-view synthesis and $K$=3 points for the mesh reconstruction task.

We train our model for $20K$ iterations. The voxels are subdivided every $h_{\mathrm{every}}$=$1K$ iterations until $15K$ iterations, where the voxels with top $h_{\mathrm{percent}}$=5 percent priority are subdivided each time. We set $h_{\mathrm{rate}}$=1 and skip subdividing voxels with a maximum sampling rate below $2h_{\mathrm{rate}}$. The voxels are pruned every $h_{\mathrm{every}}$=$1K$ iterations until $18K$ iterations, where voxels with maximum blending weights less than a pruning threshold are removed. The pruning threshold is linearly increased from 0.0001 at the first pruning to $h_{\mathrm{prune}}$=0.05 at the last pruning.

The loss weights are set to $\lambda_{\mathrm{ssim}}$=0.02, $\lambda_{\mathrm{T}}$=0.01, $\lambda_{\mathrm{dist}}$=0.1 after $10K$ iterations, $\lambda_{\mathrm{R}}$=0.01, $\lambda_{\mathrm{tv}}$=1e−10 until $10K$ iterations. For the mesh reconstruction task, the weights for normal-depth alignment self-consistency loss are set to $\lambda_{\text{n-dmean}}$=0.001 and $\lambda_{\text{n-dmed}}$=0.001 for mean and median depth respectively. The initial depths and normals are bad so the two normal-depth consistency loss is activated at the later training iterations. We find the median depth converges the fastest so we activate median depth-normal consistency loss at $3K$ iterations, which also only regularizes the rendered depth as median depth is not differentiable. The mean depth-normal consistency loss is activated at $10K$ iterations.

# D. Additional Ablation Studies

## D.1. Novel-View Synthesis

We conduct comprehensive ablation experiments of our method using the indoor **bonsai** and the outdoor **bicycle** scenes from the MipNeRF-360 [2] dataset.

**Adaptive voxel sizes.** In the main paper, we show that adaptive voxel size for different levels of detail is crucial to achieve high-quality results. The results are recapped in Tab. 6. We provide experiment details here. The starting point of the main foreground region is the same for all variants with $64^3$ dense voxels. Regarding the background region, using the same voxel size as the foreground region is impracticable for uniform-sized variants. Instead, each of the 5 background shell voxels is uniformly subdivided by 4 times as initialization for all the variants. The difference is that the uniform-sized variants subdivide all voxels each time until the grid resolution of the main region reaches $256^3$, $512^3$, or $1024^3$ instead of subdividing voxels adaptively as described in the main paper. The pruning setup remains the same for all variants. The result in Tab. 6 shows that adaptive voxel sizes are the key to solve the scalability issue of uniform-sized voxel, which achieves much better rendering quality with high render FPS.

**More ablation studies for the hyperparameters.** We conduct more ablation experiments to show the effectiveness of the hyperparameters in Tabs. 7 to 17. We mark the adopted hyperparameter setup by "*" in the table rows. The setup of the marked rows across different tables can be different as we update the base setups in a rolling manner during the hyperparameter tuning stage. In each table, the other hyperparameter setups except the ablated one are the same. We discuss the experiments directly in the table captions to avoid the need for cross-referencing between the tables and the main text.

## D.2. Mesh Reconstruction

To show the effectiveness of the mesh regularization losses, we use the **Ignatius** and the **Truck** scenes from TnT [21] dataset and three scans with id **24, 69, 122** from DTU [17] dataset for ablation studies. The results are shown in Tab. 18. While the normal-depth self-consistency losses do not improve novel-view synthesis quality in Tab. 17, the mesh accuracy is improved by an obvious margin with the regularizations.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| no ss | **111** | **13.5m** | 0.201 | 27.69 | 0.830 |
| $h_{\mathrm{ss}}$=1.01 | 108 | **13.5m** | 0.193 | 28.24 | 0.845 |
| $h_{\mathrm{ss}}$=1.10* | 107 | **13.5m** | 0.190 | 28.32 | 0.848 |
| $h_{\mathrm{ss}}$=1.20 | 99 | 13.6m | 0.188 | 28.36 | 0.849 |
| $h_{\mathrm{ss}}$=1.30 | 100 | 13.7m | 0.187 | 28.39 | 0.850 |
| $h_{\mathrm{ss}}$=1.50 | 92 | 13.8m | 0.186 | 28.42 | 0.851 |
| $h_{\mathrm{ss}}$=2.00 | 75 | 14.2m | **0.185** | **28.46** | **0.853** |

Table 7. **Supersampling rate.** Our rendering suffers from aliasing artifact so we render the image in $h_{\mathrm{ss}}\times$ higher resolution and apply image downsampling with anti-aliasing filter. The quality without supersampling is much worse than the others. Resampling the image with a very small $h_{\mathrm{ss}} = 1.01$ can already boost quality significantly. We find the quality can keep going better with higher $h_{\mathrm{ss}}$ but the FPS drops by more than 30% at $h_{\mathrm{ss}} = 2$. More future development is needed for a more efficient anti-aliasing rendering of our method. We use $h_{\mathrm{ss}} = 1.1$ for speed-quality trade-off.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $N_{\text{shd}}$=1 | **118** | **11.2m** | 0.201 | 27.43 | 0.840 |
| $N_{\text{shd}}$=2 | 114 | 12.1m | 0.193 | 27.94 | 0.847 |
| $N_{\text{shd}}$=3* | 107 | 13.5m | **0.190** | **28.32** | **0.848** |

Table 8. **Degree of Spherical Harmonic (SH).** The rendering time with higher SH degree is similar but the quality is much better. We use $N_{\text{shd}}$=3 as our final setup. However, about 80% of the parameters and the disk space is occupied by the SH coefficient with $N_{\text{shd}}$=3. Future work may want to design a more parameters efficient representation for view-dependent colors.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $K$=1* | **107** | **13.5m** | 0.190 | 28.32 | 0.848 |
| $K$=2 | 102 | 13.8m | **0.189** | 28.32 | **0.849** |
| $K$=3 | 99 | 13.9m | **0.189** | **28.33** | **0.849** |

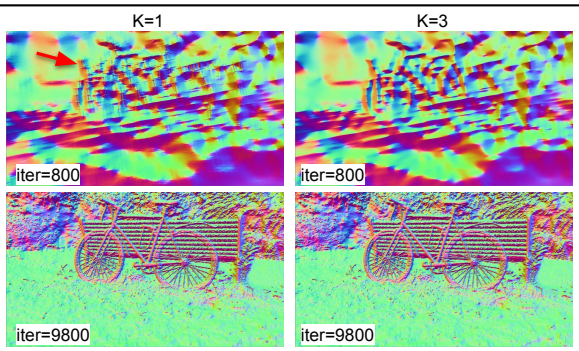



Table 9. **Number of sample points in a voxel when rendering.** The effect of sampling more point inside a voxel is marginal as the voxels are typically subdivided into fine level with small size. It mainly affects the depth rendering for larger voxels. The figure shows the normal derived from the rendered depth. $K$=1 at the early training stage produce noisy depth as highlighted by the red arrow, while the depth noisy is mitigated when the voxels is subdivided into finer level. We suggest to use $K$>1 only when the sub voxel depth accuracy is required.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $h_{\text{prune}}$=0.01 | 96 | 16.7m | 0.191 | **28.41** | 0.847 |
| $h_{\text{prune}}$=0.03 | 107 | 13.5m | **0.190** | 28.32 | **0.848** |
| $h_{\text{prune}}$=0.05* | 119 | 11.6m | 0.192 | 28.20 | 0.846 |
| $h_{\text{prune}}$=0.10 | 158 | 9.1m | 0.199 | 27.95 | 0.840 |
| $h_{\text{prune}}$=0.15 | 188 | 7.7m | 0.212 | 27.72 | 0.831 |
| $h_{\text{prune}}$=0.20 | 213 | 6.8m | 0.224 | 27.53 | 0.823 |
| $h_{\text{prune}}$=0.30 | **241** | **5.9m** | 0.248 | 27.22 | 0.806 |

Table 10. **Pruning threshold.** We prune voxels with maximum blending weights below $h_{\text{prune}}$. Higher FPS and faster processing time can be achieved by pruning more voxels but with loss in quality. We finally use $h_{\text{prune}}$=0.05 to balance speed and quality.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| 0.33 | **159** | **9.9m** | 0.210 | 27.98 | 0.833 |
| 0.50 | 130 | 11.2m | 0.197 | 28.16 | 0.843 |
| 1.00* | 107 | 13.5m | **0.190** | 28.32 | **0.848** |
| 2.00 | 101 | 15.0m | **0.190** | **28.36** | **0.848** |
| 3.00 | 101 | 15.7m | **0.190** | **28.36** | **0.848** |

Table 11. **Subdivision scale.** We subdivide $h_{\text{percent}}$=5 percent of the voxels with the highest priority 15 times during the training. As the number of voxels become $(1 + 0.07h_{\text{percent}})$ at each subdivision, the subdivision scales in above table shows their $\frac{(1+0.07h_{\text{percent}})^{15}}{1.35^{15}}$, which indicate the expected relative number of voxels comparing to the base setup. The merit of subdividing more voxels each time is marginal comparing to the base setup.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $h_{\text{ratio}}$=1.0 | **120** | **11.6m** | 0.195 | 28.17 | 0.844 |
| $h_{\text{ratio}}$=2.0* | 107 | 13.5m | 0.190 | 28.32 | 0.848 |
| $h_{\text{ratio}}$=3.0 | 103 | 14.7m | **0.189** | 28.35 | **0.849** |
| $h_{\text{ratio}}$=4.0 | 103 | 15.5m | **0.189** | **28.38** | **0.849** |

Table 12. **Initial ratio of the number of voxels in background and main regions.** At the initialization stage, we heuristically subdivide voxel in the background region until the ratio of the number of voxel is $h_{\text{ratio}}$ to the foreground region. The overall result quality are similar for different $h_{\text{ratio}}$. It affects training time more than testing FPS as the training iterations per second before any pruning is depend on the initial number of voxels.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $\lambda_{\text{tv}}$=0 | 102 | 14.0m | 0.202 | 27.77 | 0.832 |
| $\lambda_{\text{tv}}$=1e−11 | 106 | 13.6m | 0.196 | 27.97 | 0.840 |
| $\lambda_{\text{tv}}$=1e−10* | **107** | **13.5m** | **0.190** | **28.32** | **0.848** |
| $\lambda_{\text{tv}}$=1e−9 | 99 | 15.5m | 0.213 | 27.85 | 0.822 |

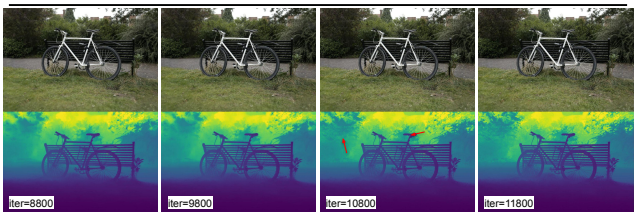



Table 13. **Total Variation (TV) loss.** Similar to previous grid-based approaches [5, 11, 44], TV loss is also important in our method. We apply TV loss on density grid only for the first half 10,000 iterations as applying TV for all iterations leads to blurrier rendering. TV with proper loss weighting leads to better quantitative results without loss of speed. The effect of TV loss is also visualized in above figure, where many geometric details emerge after the TV loss is turned off. The employed TV loss scheduling entails the coarse-to-fine optimization strategy.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $\lambda_{\mathrm{R}}$=0 | 111 | 14.6m | 0.200 | 28.11 | 0.843 |
| $\lambda_{\mathrm{R}}$=1e−4 | 110 | 14.6m | 0.199 | 28.11 | 0.843 |
| $\lambda_{\mathrm{R}}$=1e−3 | 107 | 14.5m | 0.196 | 28.20 | 0.845 |
| $\lambda_{\mathrm{R}}$=1e−2* | 107 | 13.5m | **0.190** | **28.32** | **0.848** |
| $\lambda_{\mathrm{R}}$=1e−1 | **118** | **10.9m** | 0.205 | 27.77 | 0.830 |

Table 14. **Color concentration loss.** We find it helpful to apply L2 loss directly between observed pixel color and the individual voxel color of each voxel passing by the ray [44], which slightly improve training time and result quality.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $\lambda_{\mathrm{dist}}$=0 | 105 | 14.9m | 0.199 | 27.27 | 0.839 |
| $\lambda_{\mathrm{dist}}$=1e−4 | 106 | 15.4m | 0.199 | 27.48 | 0.839 |
| $\lambda_{\mathrm{dist}}$=1e−3 | 105 | 15.1m | 0.195 | 27.97 | 0.842 |
| $\lambda_{\mathrm{dist}}$=1e−2 | 107 | 13.5m | 0.190 | **28.32** | **0.848** |
| $\lambda_{\mathrm{dist}}$=1e−1 | **137** | **9.9m** | 0.256 | 26.34 | 0.760 |
| $\lambda_{\mathrm{dist}}$=1e−4 from 10K | 104 | 15.1m | 0.199 | 27.40 | 0.839 |
| $\lambda_{\mathrm{dist}}$=1e−3 from 10K | 105 | 15.0m | 0.197 | 27.76 | 0.842 |
| $\lambda_{\mathrm{dist}}$=1e−2 from 10K | 105 | 14.6m | 0.193 | 28.08 | 0.845 |
| $\lambda_{\mathrm{dist}}$=1e−1 from 10K* | 113 | 13.7m | **0.188** | 28.11 | **0.848** |

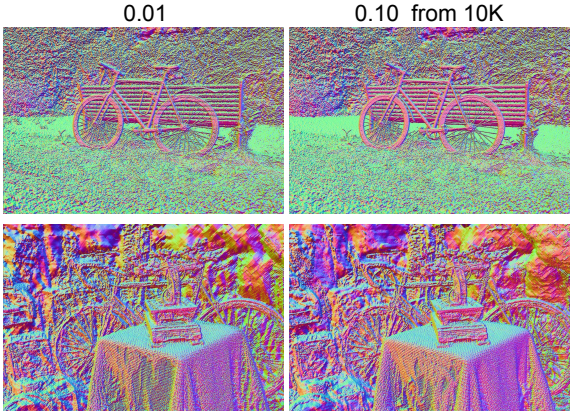



Table 15. **Distortion loss.** Distortion loss is proposed by MipNeRF-360 [2] and employed by many NeRF-based rendering approaches to encourage concentration of the blending weight distribution on a ray. We find distortion loss is also helpful in our method, especially for the PSNR. We also find that employing a larger distortion loss weight after the total variation loss is turned off lead to a cleaner geometry as shown in the above depth-derived normal visualization.

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| $\lambda_{\mathrm{T}}$=1e−0 | **109** | 13.4m | 0.192 | 28.23 | 0.847 |
| $\lambda_{\mathrm{T}}$=1e−3 | **109** | 13.5m | 0.191 | 28.25 | 0.847 |
| $\lambda_{\mathrm{T}}$=1e−2* | 107 | 13.5m | **0.190** | **28.32** | **0.848** |
| $\lambda_{\mathrm{T}}$=1e−1 | **109** | **12.8m** | 0.192 | 28.13 | 0.845 |

Table 16. **Transmittance concentration loss.** The effect of encouraging final ray transmittance to be either zero or one is marginal in the unbounded scenes. We find this loss is more important for the object-centric scenes with foreground region only and known background colors (*e.g.*, Synthetic-NeRF dataset [31]).

| Setup | FPS↑ | Tr. time↓ | LPIPS↓ | PSNR↑ | SSIM↑ |
|---|---|---|---|---|---|
| neither* | 107 | **13.5m** | **0.190** | **28.32** | 0.848 |
| n-dmed | **114** | **13.5m** | 0.191 | 28.10 | 0.849 |
| n-dmean | 97 | 14.0m | **0.190** | 28.14 | **0.850** |
| both | 103 | 14.4m | 0.191 | 27.99 | 0.849 |

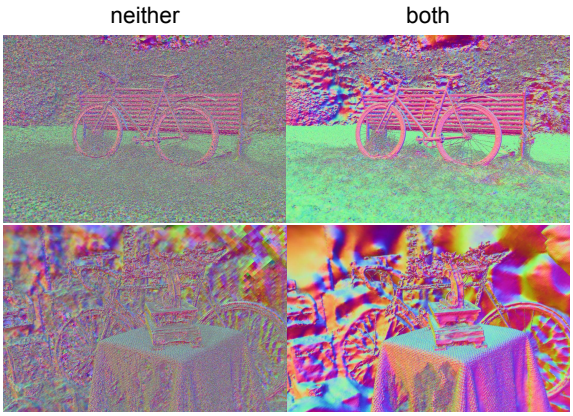



Table 17. **Mesh regularization losses for novel-view synthesis.** We also try the normal-depth self-consistency losses for novel-view synthesis task. Despite of loss a little in PSNR, the regularization can make the rendered normals much smoother as shown in the visualization.

| | | | TnT dataset | | DTU dataset | |
|---|---|---|---|---|---|---|
| $\mathcal{L}_{\text{n-dmed}}$ | $\mathcal{L}_{\text{n-dmean}}$ | $K$ | F-score↑ | Tr. time↓ | Cf.↓ | Tr. time↓ |
| | | 3 | 0.56 | **10.1m** | 0.94 | **5.5m** |
| ✓ | | 3 | 0.59 | **10.1m** | 0.68 | **5.5m** |
| | ✓ | 3 | 0.61 | 10.6m | 0.68 | 5.7m |
| ✓ | ✓ | 1 | 0.61 | 10.7m | 0.66 | 5.8m |
| ✓ | ✓ | 2 | 0.61 | 10.8m | **0.65** | 5.9m |
| ✓ | ✓ | 3 | **0.62** | 10.9m | **0.65** | 6.0m |

Table 18. **Mesh regularization losses.** We show the results of the mesh regularization losses and the number of sample points when rendering a voxel on a subset of Tanks&Temples [21] and DTU [17] datasets.

| Method | 3DGS variants | | | | Ours | |
|---|---|---|---|---|---|---|
| | 3DGS [20]† | StopThePop [38]† | 3DGRT [32] | EVER [28] | fast-rend | base |
| No popping | | △ | △ | ✓ | ✓ | ✓ |
| FPS↑ | 131 | 94 | 43‡ | 20‡ | **258** | 121 |
| LPIPS↓ | 0.257 | 0.251 | 0.248 | 0.233 | 0.249 | **0.219** |
| PSNR↑ | 27.45 | 27.35 | 27.20 | **27.51** | 26.87 | 27.33 |
| SSIM↑ | 0.815 | 0.816 | 0.818 | **0.825** | 0.804 | 0.822 |

† Re-evaluated on our machine using the public code.

‡ We scale the FPS to align their reported 3DGS FPS to our reproduced 3DGS.

Table 19. **Comparison with 3DGS variants tackling popping artifact on Mip-NeRF360 dataset [2].** The LPIPS values here are evaluated with the correct intensity scale between $[-1, 1]$ following EVER [28]. 3DGRT and EVER use ray tracing approach instead of rasterization. EVER solves the Gaussians ordering and overlapping issues but sacrificing more FPS.

## E. More Results

**Comparison with popping-resistant 3DGS variants.** More comparisons with recent 3DGS variants are in Tab. 19. 3DGS [20] has popping artifacts due to the ordering and overlapping issues. StopThePop [38] uses running sort and 3DGRT [32] uses ray tracing for accurate ordering but drops FPS by 28% and 67% respectively. EVER [28] further handles the Gaussian overlapping cases but with even less FPS. Our method ensures correct ordering (Sec. 3.1.2) with FPS and quality comparable to the original 3DGS.

**Results breakdown for novel-view synthesis.** In Tab. 20, we show details per-scene comparison with 3DGS [20] using our base setup. Our method uses much more primitives (*i.e.*, voxels or Gaussians) compared to 3DGS on all the scenes. However, our average rendering FPS is still comparable to 3DGS. We find the FPS is scene-dependent, where we achieve much faster FPS on some of the scenes while slower on the others. Our method generally uses short training time. Regarding the quality metrics, our results are typically $-0.2$db PSNR and $-0.01$ SSIM behind 3DGS, while our LPIPS is better on average.

As discussed in the main paper, not only the scene representation itself affects the results, but the optimization and adaptive procedure are also an important factor. The strategy of adding more Gaussians progressively is not applicable to ours. We also have not explored to use of the coarse geometry estimated from SfM, while 3DGS uses SfM sparse points for initialization. As the first attempt of marrying rasterizer with fully explicit sparse voxels for scene reconstruction, there is still future potential for improvement from different aspects.

**Results breakdown for mesh reconstruction.** The F-score and chamfer distance of each scene from the Tanks&Temples and DTU [17] datasets are provided in Tab. 22. We only list the two representative NeRF-based methods and two GS-based methods in the result breakdown comparison. More methods with the average scores are in the main paper.

**Synthetic dataset.** The results on the Synthetic-NeRF [31] dataset is provided in the last section of Tab. 21. We achieve good quality, high FPS, and fast training on this dataset. However, our quality is slightly worse than 3DGS [20] with slower FPS. Our development mainly focuses on real-world datasets. Future work may need more exploration to continue development on this dataset.

**More qualitative results** We show qualitative comparison with 3DGS [20] on indoor and outdoor scenes in Fig. 11 and Fig. 12, respectively. Our visual quality is on par with 3DGS. We provide the visualization of the raw reconstructed meshes in Fig. 13. For quantitative evaluation, we follow previous works to apply mesh cleaning with the provided bounding box or masks. Despite the good quantitative results for meshes, some apparent artifacts can be observed from the visualization. In particular, our method focuses more on the geometric details and sometimes over-explains the texture on a flat surface with complex geometry. Future work may want to model a signed distance field instead of our current density field and introduce surface smoothness regularizers.

| Scene | FPS↑ | | Tr. time↓ (mins) | | LPIPS↓ | | PSNR↑ | | SSIM↑ | | # prim.↓ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours |
| MipNeRF-360 [2] indoor scenes | | | | | | | | | | | | |
| bonsai | **215** | 128 | 18.3 | **15.3** | 0.204 | **0.171** | **31.89** | 31.51 | 0.942 | **0.944** | **1.2M** | 6.6M |
| counter | **160** | 85 | 20.7 | **18.7** | 0.199 | **0.176** | **29.03** | 28.72 | **0.909** | 0.905 | **1.2M** | 8.4M |
| kitchen | **128** | 78 | 25.0 | **17.8** | 0.126 | **0.112** | **31.47** | 31.29 | 0.927 | **0.934** | **1.8M** | 9.2M |
| room | **153** | 131 | 21.1 | **17.1** | 0.218 | **0.185** | **31.44** | 31.10 | 0.919 | **0.924** | **1.5M** | 8.9M |
| MipNeRF-360 [2] outdoor scenes | | | | | | | | | | | | |
| bicycle | 72 | **147** | 31.9 | **13.5** | 0.211 | **0.190** | 25.18 | **25.29** | 0.765 | **0.773** | **6.1M** | 9.2M |
| garden | 81 | **118** | 33.1 | **12.4** | 0.107 | **0.106** | **27.39** | 27.31 | **0.867** | 0.865 | **5.9M** | 9.6M |
| stump | 110 | **129** | 25.5 | **13.0** | 0.216 | **0.206** | **26.61** | 26.38 | **0.772** | 0.769 | **4.9M** | 9.2M |
| treehill | 123 | **157** | 22.4 | **13.7** | 0.327 | **0.262** | 22.47 | **22.74** | 0.632 | **0.646** | **3.7M** | 9.4M |
| flowers | **137** | 120 | 22.0 | **14.4** | 0.335 | **0.268** | 21.57 | **21.72** | 0.606 | **0.637** | **3.6M** | 9.4M |
| DeepBlending [15] indoor scenes | | | | | | | | | | | | |
| drjohnson | 116 | **297** | 25.0 | **8.7** | 0.244 | **0.242** | 29.11 | **29.22** | **0.901** | 0.892 | **3.3M** | 6.8M |
| playroom | 163 | **308** | 19.7 | **7.4** | 0.244 | **0.211** | 30.08 | **30.53** | **0.907** | 0.900 | **2.3M** | 6.3M |
| Tanks&Temples [21] outdoor scenes | | | | | | | | | | | | |
| train | **206** | 127 | **11.3** | 11.4 | 0.206 | **0.186** | **22.11** | 21.26 | **0.816** | 0.813 | **1.1M** | 8.3M |
| truck | **154** | 129 | 16.3 | **11.0** | 0.147 | **0.100** | **25.40** | 25.00 | 0.882 | **0.888** | **2.6M** | 9.0M |

Table 20. **Real-world datasets for per-scene side-by-side comparison with 3DGS [20].** Our result here is the base setup. We show average results of our 2x faster rendering and 3x faster training variants in the main paper.

| Scene | FPS↑ | | Tr. time↓ (mins) | | LPIPS↓ | | PSNR↑ | | SSIM↑ | | # prim.↓ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours | 3DGS | ours |
| Synthetic-NeRF [31] object scenes | | | | | | | | | | | | |
| chair | **418** | 197 | 5.4 | **4.6** | **0.012** | 0.013 | 35.89 | **35.91** | **0.987** | 0.986 | **0.3M** | 3.0M |
| drums | **406** | 241 | 5.8 | **4.1** | **0.037** | 0.043 | **26.16** | 26.09 | **0.955** | 0.947 | **0.3M** | 2.3M |
| ficus | **476** | 360 | 5.0 | **3.1** | **0.012** | 0.014 | **34.85** | 34.37 | **0.987** | 0.984 | **0.3M** | 1.3M |
| hotdog | **596** | 218 | 5.2 | **4.8** | 0.020 | **0.019** | **37.67** | 37.42 | **0.985** | 0.984 | **0.1M** | 2.8M |
| lego | **415** | 156 | 5.9 | **5.8** | **0.015** | 0.016 | **35.77** | 35.54 | **0.983** | 0.981 | **0.3M** | 4.3M |
| materials | **575** | 213 | 5.3 | **4.5** | **0.034** | 0.037 | **30.01** | 30.00 | **0.960** | 0.954 | **0.3M** | 2.7M |
| mic | **344** | 328 | 5.6 | **3.0** | **0.006** | 0.007 | 35.38 | **36.00** | 0.991 | **0.992** | **0.3M** | 1.1M |
| ship | **254** | 111 | **7.9** | 8.6 | 0.107 | **0.106** | **30.92** | 30.38 | **0.907** | 0.886 | **0.3M** | 5.7M |

Table 21. **Synthetic object-centric dataset for per-scene side-by-side comparison with 3DGS [20].** Our overall quality is slightly worse than 3DGS on this dataset while the synthetic object-centric scenario is off our main focus.

| Method | Tanks&Temples F-score↑ | | | | | | DTU Chamfer Cistance↓ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Barn | Caterpillar | Courthouse | Ignatius | Meetingroom | Truck | 24 | 37 | 40 | 55 | 63 | 65 | 69 | 83 | 97 | 105 | 106 | 110 | 114 | 118 | 122 |
| NeuS | 0.29 | 0.29 | 0.17 | 0.83 | 0.24 | 0.45 | 1.00 | 1.37 | 0.93 | 0.43 | 1.10 | 0.65 | 0.57 | 1.48 | 1.09 | 0.83 | 0.52 | 1.20 | 0.35 | 0.49 | 0.54 |
| Neuralangelo | **0.70** | **0.36** | 0.28 | **0.89** | **0.32** | 0.48 | **0.37** | **0.72** | **0.35** | **0.35** | **0.87** | **0.54** | **0.53** | **1.29** | **0.97** | 0.73 | **0.47** | **0.74** | **0.32** | **0.41** | **0.43** |
| 3DGS | 0.13 | 0.08 | 0.09 | 0.04 | 0.01 | 0.19 | 2.14 | 1.53 | 2.08 | 1.68 | 3.49 | 2.21 | 1.43 | 2.07 | 2.22 | 1.75 | 1.79 | 2.55 | 1.53 | 1.52 | 1.50 |
| 2DGS | 0.41 | 0.23 | 0.16 | 0.51 | 0.17 | 0.45 | 0.48 | 0.91 | 0.39 | 0.39 | 1.01 | 0.83 | 0.81 | 1.36 | 1.27 | 0.76 | 0.70 | 1.40 | 0.40 | 0.76 | 0.52 |
| ours | 0.35 | 0.33 | **0.29** | 0.69 | 0.19 | **0.54** | 0.61 | 0.74 | 0.41 | 0.36 | 0.93 | 0.75 | 0.94 | 1.33 | 1.40 | **0.61** | 0.63 | 1.19 | 0.43 | 0.57 | 0.44 |

Table 22. **Result breakdown on Tanks&Temples [21] and DTU [17] datasets.**

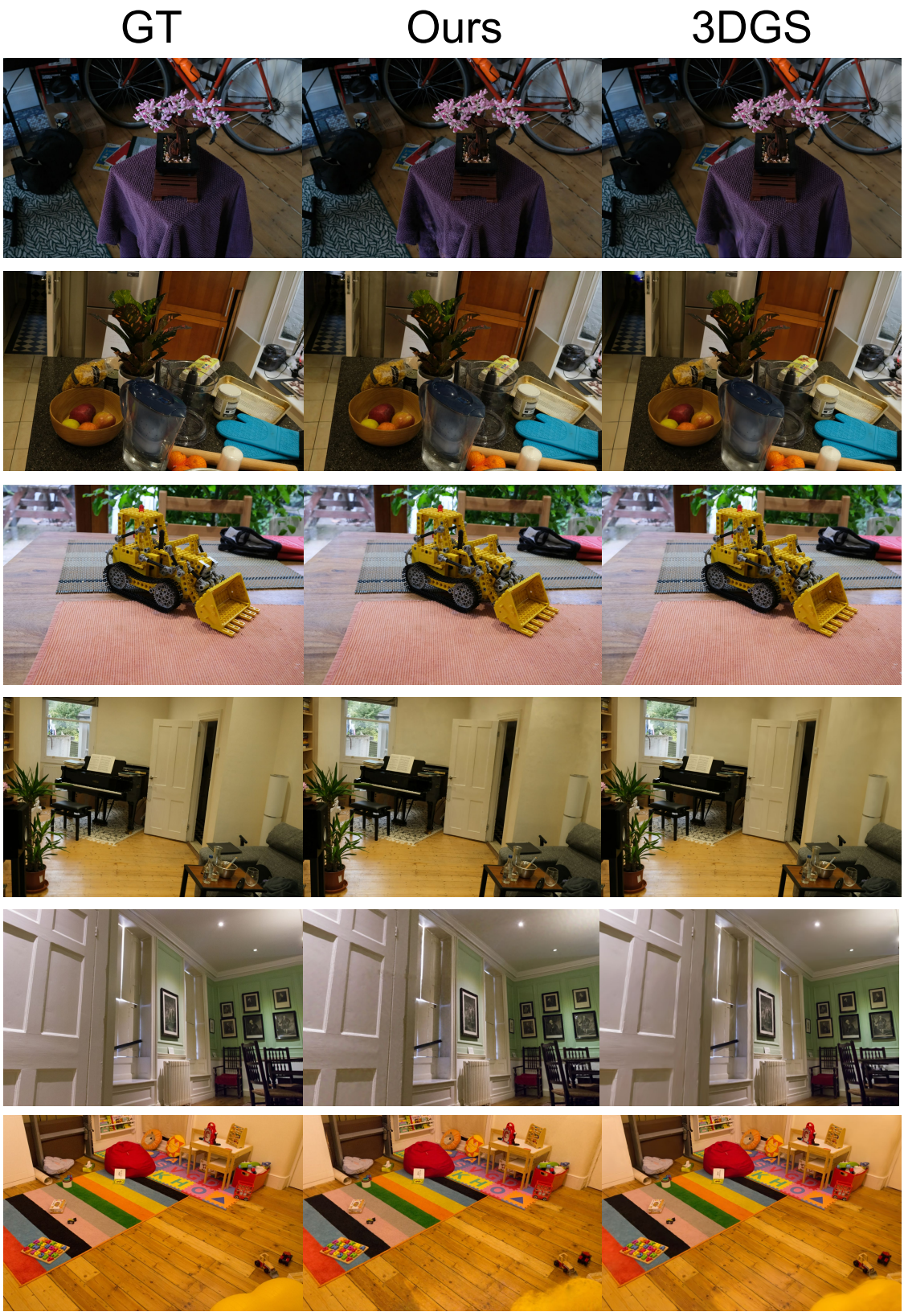



Figure 11. **Qualitative novel-view rendering results on-par with 3DGS.**


Figure 12. **Qualitative novel-view rendering results on-par with 3DGS.**

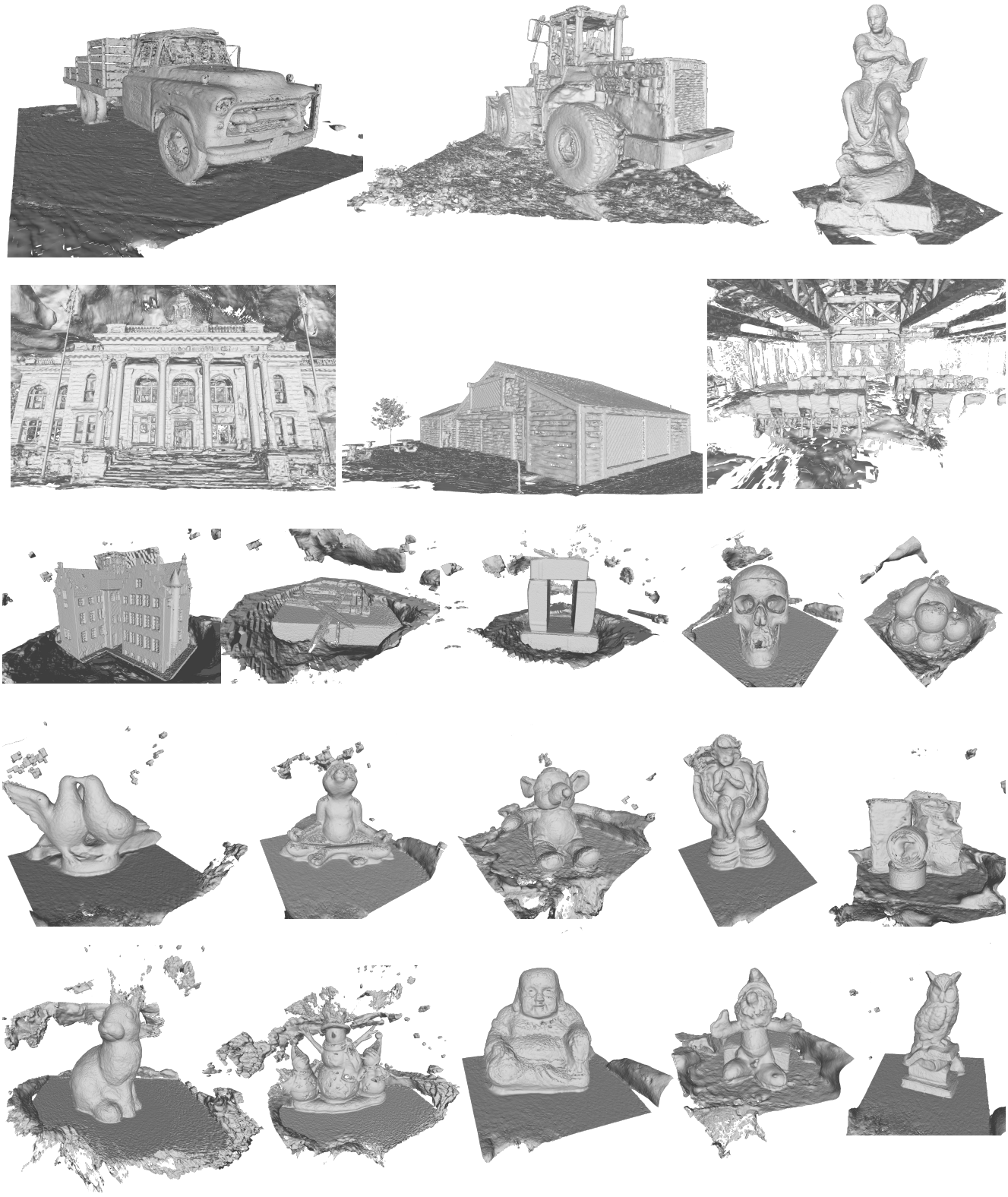


Figure 13. **Qualitative results of the reconstructed mesh.**