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:

$$\mathbf{v}_\text{s} = \mathbf{w}_\text{s} \cdot 2^{-l}\,, \qquad \mathbf{v}_\text{c} = \mathbf{w}_\text{c} - 0.5 \cdot \mathbf{w}_\text{s} + \mathbf{v}_\text{s} \cdot v\,. \tag{1}$$

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. 1. 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 [7, 19].

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:

$$t_k = a + \frac{k-0.5}{K} \cdot (b-a) \tag{2a}$$

$$\mathbf{p}_k = \mathbf{r}_\mathrm{o} + t_k \cdot \mathbf{r}_\mathrm{d} \tag{2b}$$

$$\mathbf{q}_k = (\mathbf{p}_k - (\mathbf{v}_\mathrm{c} - 0.5 \cdot \mathbf{v}_\mathrm{s})) \cdot \frac{1}{\mathbf{v}_\mathrm{s}} , \tag{2c}$$

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:

$$\mathrm{interp}(\mathbf{V}, \mathbf{q}) = \begin{bmatrix} (1-\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (1-\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(1-\mathbf{q}_y)\cdot(\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(1-\mathbf{q}_z) \\ (\mathbf{q}_x)\cdot(\mathbf{q}_y)\cdot(\mathbf{q}_z) \end{bmatrix}^\mathsf{T} \begin{bmatrix} \mathbf{V}_{000} \\ \mathbf{V}_{001} \\ \mathbf{V}_{010} \\ \mathbf{V}_{011} \\ \mathbf{V}_{100} \\ \mathbf{V}_{101} \\ \mathbf{V}_{110} \\ \mathbf{V}_{111} \end{bmatrix} , \tag{3}$$

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 [12, 14], we use quadrature to compute the integrated volume density for alpha value:

$$\alpha = 1 - \exp\left(-\frac{l}{K} \sum_{k=1}^{K} \mathrm{explin}\,(v_k) \right) \tag{4a}$$

$$v_k = \mathrm{interp}\,(\mathbf{V}, \mathbf{q}_k) \tag{4b}$$

$$l = (b-a) \cdot \|\mathbf{r}_\mathrm{d}\| , \tag{4c}$$

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

$$\nabla_\mathbf{V}\, \alpha = (1-\alpha) \cdot \frac{l}{K} \cdot \sum_{k=1}^{K} \left(\frac{\mathrm{d}}{\mathrm{d}v_k} \mathrm{explin}(v_k) \cdot \nabla_\mathbf{V} v_k \right) . \tag{5}$$

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:

$$\nabla_\mathbf{q} \mathrm{interp}(\mathbf{V}, \mathbf{q}^{(\mathrm{c})}) = 0.25 \cdot \begin{bmatrix} (\mathbf{V}_{100}+\mathbf{V}_{101}+\mathbf{V}_{110}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{001}+\mathbf{V}_{010}+\mathbf{V}_{011}) \\ (\mathbf{V}_{010}+\mathbf{V}_{011}+\mathbf{V}_{110}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{001}+\mathbf{V}_{100}+\mathbf{V}_{101}) \\ (\mathbf{V}_{001}+\mathbf{V}_{011}+\mathbf{V}_{101}+\mathbf{V}_{111}) - (\mathbf{V}_{000}+\mathbf{V}_{010}+\mathbf{V}_{100}+\mathbf{V}_{110}) \end{bmatrix} \tag{6}$$

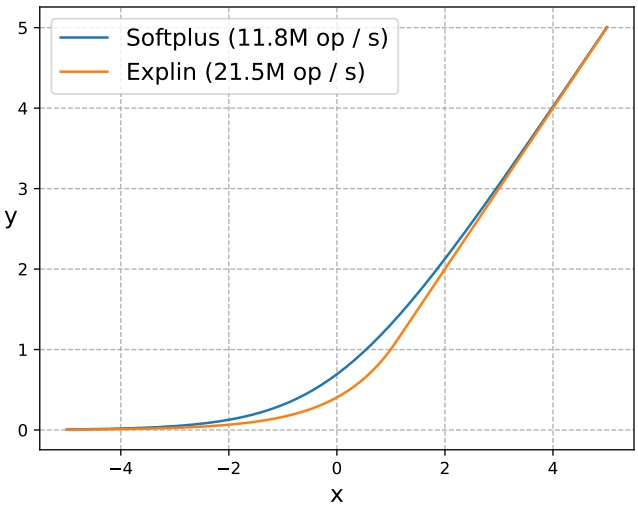

Figure 1. **Activation functions.** We use exponential-linear activation to softly map raw density to non-negative volume density. Exp-lin activation is about two times faster to compute in CUDA, which is 21.5M operations per second in a CUDA thread comparing to 11.8M of Softplus. See Sec. A.2 for more details.

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

$$\nabla_{\mathbf{V}_{ijk}} \nabla_\mathbf{q} \mathrm{interp}(\mathbf{V}, \mathbf{q}^{(\mathrm{c})}) = 0.25 \cdot \begin{bmatrix} 2i-1 \\ 2j-1 \\ 2k-1 \end{bmatrix} . \tag{7}$$

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. (2a) into the local alpha composition in Eq. (4). Let

$$\alpha_k = 1 - \exp\left(-\frac{l}{K} \cdot \mathrm{explin}\,(\mathrm{interp}(\mathbf{V}, \mathbf{q}_k)) \right) \tag{8}$$

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

$$d = \sum_{k=1}^{K} \left(\prod_{j=1}^{k-1} (1-\alpha_j) \right) \cdot \alpha_k \cdot t_k . \tag{9}$$

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:

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_1} = t_1 - \alpha_2 t_2, \quad \frac{\mathrm{d}d}{\mathrm{d}\alpha_2} = t_2 - \alpha_1 t_2 \,. \tag{10}$$

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

$$d = \alpha_1 t_1 + (1-\alpha_1)\alpha_2 t_2 + (1-\alpha_1)(1-\alpha_2)\alpha_3 t_3 \,. \tag{11}$$

The backward equations are:

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_1} = t_1 + \alpha_2\alpha_3 t_3 - \alpha_2 t_2 - \alpha_3 t_3 \tag{12a}$$

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_2} = t_2 + \alpha_1\alpha_3 t_3 - \alpha_1 t_2 - \alpha_3 t_3 \tag{12b}$$

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha_3} = t_3 + \alpha_1\alpha_2 t_3 - \alpha_1 t_3 - \alpha_2 t_3 \,. \tag{12c}$$

B. More Details of Voxel Rendering Order

Our sorting-based rasterizer is based on the efficient CUDA implementation done by 3DGS [8]. 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 [8].

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:

$$\text{key} = |\underbrace{\text{tile id}}_{\text{32 bits}}|\underbrace{\text{Gaussian z-depth}}_{\text{32 bits}}| \tag{13a}$$

$$\text{value} = |\underbrace{\text{Gaussian id}}_{\text{32 bits}}| \tag{13b}$$

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:

$$\text{key} = |\underbrace{\text{tile id}}_{\text{16 bits}}|\underbrace{\text{Morton order}}_{\text{48 (=3}L\text{) bits}}| \tag{14a}$$

$$\text{value} = |\underbrace{\text{ray sign bits}}_{\text{3 bits}}|\underbrace{\text{voxel id}}_{\text{29 bits}}| \tag{14b}$$

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 [13] 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):

$$\ldots\, b_x b_y b_z\; a_x a_y a_z \mapsto \ldots\, f^{(\mathrm{k})}(b_x b_y b_z)\; f^{(\mathrm{k})}(a_x a_y a_z)\,, \tag{15}$$

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. 2.

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

> "For all sparse voxel sets in V^ℓ, 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 ℓ=1, there is only one Octree level. The direction-dependent Morton order of the eight voxels for $(+,+,+)$ ray directions is illustrated in Fig. 2. 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 ℓ.

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

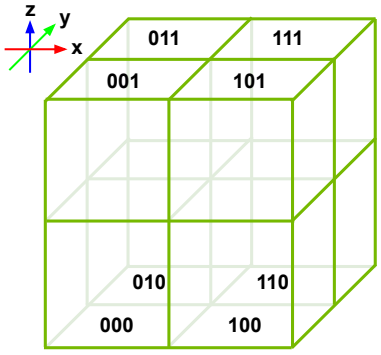

Figure 2. **Base case.** Direction-dependent Morton order for $(+,+,+)$ ray direction signs under the base case with 1 Octree level. The three bits from left to right is for the x, y, and z directions respectively. The rendering order is correct for all rays going toward $(+,+,+)$ direction. See Sec. 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 3ℓ bits as the source voxel, while the least significant 3 bits follow the same direction-dependent Morton order as in the base case Fig. 2. 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 ℓ.

C. Additional Implementation Details

We start the optimization from empty space with raw density set to h_{geo}=-10. We use spherical harmonic (SH) with N_{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 $1\text{e}{-15}$. The initial Octree level is h_{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_{out}=5 background shell levels with h_{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 1. **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_{T}=1e−4 and the supersampling scale is set to h_{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_{every}=$1K$ iterations until $15K$ iterations, where the voxels with top h_{percent}=5 percent priority are subdivided each time. We set h_{rate}=1 and skip subdividing voxels with a maximum sampling rate below $2h_{\mathrm{rate}}$. The voxels are pruned every h_{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_{prune}=0.05 at the last pruning.

The loss weights are set to λ_{ssim}=0.02, λ_{T}=0.01, λ_{dist}=0.1 after $10K$ iterations, λ_{R}=0.01, λ_{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 [1] 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. 1. 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. 1 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. 2 to 12. 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 [9] dataset and three scans with id **24, 69, 122** from DTU [6] dataset for ablation studies. The results are shown in Tab. 13. While the normal-depth self-consistency losses do not improve novel-view synthesis quality in Tab. 12, 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_{ss}=1.01	108	**13.5m**	0.193	28.24	0.845
h_{ss}=1.10*	107	**13.5m**	0.190	28.32	0.848
h_{ss}=1.20	99	13.6m	0.188	28.36	0.849
h_{ss}=1.30	100	13.7m	0.187	28.39	0.850
h_{ss}=1.50	92	13.8m	0.186	28.42	0.851
h_{ss}=2.00	75	14.2m	**0.185**	**28.46**	**0.853**

Table 2. **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_{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_{shd}=1	**118**	**11.2m**	0.201	27.43	0.840
N_{shd}=2	114	12.1m	0.193	27.94	0.847
N_{shd}=3*	107	13.5m	**0.190**	**28.32**	**0.848**

Table 3. **Degree of Spherical Harmonic (SH).** The rendering time with higher SH degree is similar but the quality is much better. We use N_{shd}=3 as our final setup. However, about 80% of the parameters and the disk space is occupied by the SH coefficient with N_{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**

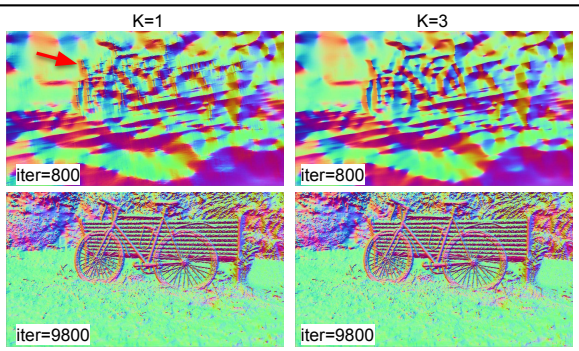

Table 4. **Number of sample points in a voxel when rendering.** The effect of sampling more point inside a voxel is marginal as the voxels are typically subdivided into fine level with small size. It mainly affects the depth rendering for larger voxels. The figure shows the normal derived from the rendered 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_{prune}=0.01	96	16.7m	0.191	**28.41**	0.847
h_{prune}=0.03	107	13.5m	**0.190**	28.32	**0.848**
h_{prune}=0.05*	119	11.6m	0.192	28.20	0.846
h_{prune}=0.10	158	9.1m	0.199	27.95	0.840
h_{prune}=0.15	188	7.7m	0.212	27.72	0.831
h_{prune}=0.20	213	6.8m	0.224	27.53	0.823
h_{prune}=0.30	**241**	**5.9m**	0.248	27.22	0.806

Table 5. **Pruning threshold.** We prune voxels with maximum blending weights below h_{prune}. Higher FPS and faster processing time can be achieved by pruning more voxels but with loss in quality. We finally use h_{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 6. **Subdivision scale.** We subdivide h_{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_{ratio}=1.0	**120**	**11.6m**	0.195	28.17	0.844
h_{ratio}=2.0*	107	13.5m	0.190	28.32	0.848
h_{ratio}=3.0	103	14.7m	**0.189**	28.35	**0.849**
h_{ratio}=4.0	103	15.5m	**0.189**	**28.38**	**0.849**

Table 7. **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_{ratio} to the foreground region. The overall result quality are similar for different h_{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↑
λ_{tv}=0	102	14.0m	0.202	27.77	0.832
λ_{tv}=1e−11	106	13.6m	0.196	27.97	0.840
λ_{tv}=1e−10*	**107**	**13.5m**	**0.190**	**28.32**	**0.848**
λ_{tv}=1e−9	99	15.5m	0.213	27.85	0.822

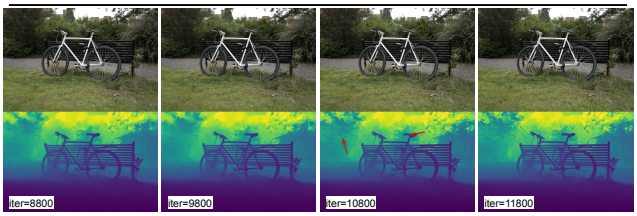

Table 8. **Total Variation (TV) loss.** Similar to previous grid-based approaches [3, 4, 19], TV loss is also important in our method. We apply TV loss on density grid only for the first half 10,000 iterations as applying TV for all iterations leads to blurrier rendering. TV with proper loss weighting 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↑
λ_{R}=0	111	14.6m	0.200	28.11	0.843
λ_{R}=1e−4	110	14.6m	0.199	28.11	0.843
λ_{R}=1e−3	107	14.5m	0.196	28.20	0.845
λ_{R}=1e−2*	107	13.5m	**0.190**	**28.32**	**0.848**
λ_{R}=1e−1	**118**	**10.9m**	0.205	27.77	0.830

Table 9. **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 [19], which slightly improve training time and result quality.

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

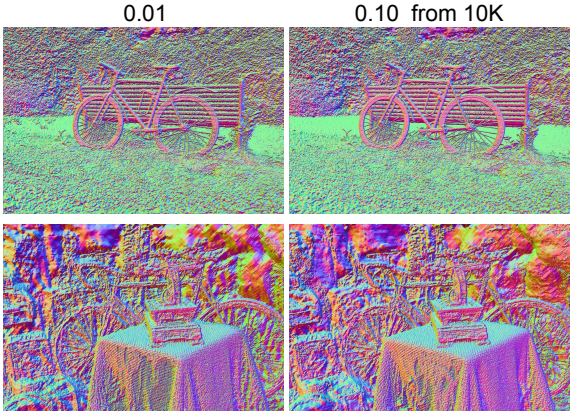

Table 10. **Distortion loss.** Distortion loss is proposed by MipNeRF-360 [1] and employed by many NeRF-based rendering approaches to encourage concentration of the blending weight distribution on a ray. We find distortion loss is also helpful in our method, especially for the PSNR. We also find that 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↑
λ_{T}=1e−0	**109**	13.4m	0.192	28.23	0.847
λ_{T}=1e−3	**109**	13.5m	0.191	28.25	0.847
λ_{T}=1e−2*	107	13.5m	**0.190**	**28.32**	**0.848**
λ_{T}=1e−1	**109**	**12.8m**	0.192	28.13	0.845

Table 11. **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 [14]).

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

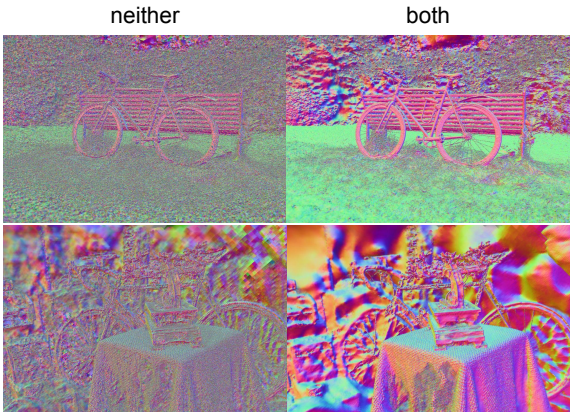

Table 12. **Mesh regularization losses for novel-view synthesis.** We also try the normal-depth self-consistency losses for novel-view synthesis task. Despite of loss a little in PSNR, the regularization can make the rendered normals much smoother as shown in the visualization.

			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 13. **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 [9] and DTU [6] datasets.

Method	3DGS variants				Ours	
	3DGS [8]†	StopThePop [17]†	3DGRT [15]	EVER [10]	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 14. **Comparison with 3DGS variants tackling popping artifact on Mip-NeRF360 dataset [1].** The LPIPS values here are evaluated with the correct intensity scale between $[-1, 1]$ following EVER [10]. 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. 14. 3DGS [8] has popping artifacts due to the ordering and overlapping issues. StopThePop [17] uses running sort and 3DGRT [15] uses ray tracing for accurate ordering but drops FPS by 28% and 67% respectively. EVER [10] further handles the Gaussian overlapping cases but with even less FPS. Our method ensures correct ordering (**??**) with FPS and quality comparable to the original 3DGS.

Results on Scannet++. Scannet++ [21] is a large-scale dataset covering various types of indoor scenes. To reconstruct the bounded indoor environments, we heuristically set the scene center as the camera centroid and the scene radius as twice the maximum camera distance from the centroid. The voxel grid starts at 64^3 without a background region. Additionally, we implement ray density ascending regularization and a spherical harmonic reset trick, which we find improves results on the public validation set. The result on the held-out test-set is shown in Tab. 15. Our method achieves good results on all metrics. Some indoor fly-through videos are provided in the released code.

Results breakdown for novel-view synthesis. In Tab. 16, we show details per-scene comparison with 3DGS [8] 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 -0.2db 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.

Method	LPIPS↓	PSNR↑	SSIM↑
Small set (12 scenes)			
Plenoxels [22]	0.399	22.177	0.841
TensoRF [3]	0.404	23.524	0.857
INGP [16]	0.363	23.695	0.871
Zip-NeRF [2]	0.320	**24.630**	**0.887**
Nerfacto [20]	0.340	23.498	0.868
3DGS [8]	0.312	23.389	0.876
FeatSplat [11]	0.303	24.177	0.880
RPBG [23]	**0.271**	24.005	0.882
Ours	0.300	24.365	0.886
Full set (50 scenes)			
TensoRF [3]	0.406	24.022	0.850
Zip-NeRF [2]	0.325	**25.041**	**0.880**
3DGS [8]	0.319	23.893	0.871
Ours	**0.313**	24.709	0.874

Table 15. **Scannet++ [21] indoor dataset.** The test-set images are not released to prevent overfitting. We submit our rendering results to the scannet++ official website for a 3rd-party evaluation. The online benchmark is: https://kaldir.vc.in.tum.de/scannetpp/benchmark/nvs. In average, our training time is 12 minutes per scene; our rendering FPS is 197 at 1752×1168 resolutions. Voxel size statistic is: 13.61% <3mm, 19.25% 3-5mm, 32.43% 5mm-1cm, 23.31% 1-2cm, 6.66% 2-3cm, 4.73% >3cm. We do not use the sparse points prior from COLMAP [18] in this submission.

Results breakdown for mesh reconstruction. The F-score and chamfer distance of each scene from the Tanks&Temples and DTU [6] datasets are provided in Tab. 18. 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 [14] dataset is provided in the last section of Tab. 17. We achieve good quality, high FPS, and fast training on this dataset. However, our quality is slightly worse than 3DGS [8] 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 [8] on indoor and outdoor scenes in Fig. 3 and Fig. 4, respectively. Our visual quality is on par with 3DGS. We provide the visualization of the raw reconstructed meshes in Fig. 5. For quantitative evaluation, we follow previous works to apply mesh cleaning with the provided bounding box or masks. Despite the good quantitative

	FPS↑		Tr. time↓ (mins)		LPIPS↓		PSNR↑		SSIM↑		# prim.↓	
Scene	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours
MipNeRF-360 [1] 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 [1] 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 [5] 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 [9] 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 16. **Real-world datasets for per-scene side-by-side comparison with 3DGS [8].** 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.

	FPS↑		Tr. time↓ (mins)		LPIPS↓		PSNR↑		SSIM↑		# prim.↓	
Scene	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours	3DGS	ours
Synthetic-NeRF [14] 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 17. **Synthetic object-centric dataset for per-scene side-by-side comparison with 3DGS [8].** Our overall quality is slightly worse than 3DGS on this dataset while the synthetic object-centric scenario is off our main focus.

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.

References

[1] 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. 5, 7, 8, 9

[2] 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. 8

[3] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In ECCV, 2022. 6, 8

[4] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In CVPR, 2022. 6

[5] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel J. Brostow. Deep blending for

	Tanks&Temples F-score↑						DTU Chamfer Cistance↓														
Method	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 18. **Result breakdown on Tanks&Temples [9] and DTU [6] datasets.**

free-viewpoint image-based rendering. ACM TOG, 2018. 9

[6] Rasmus Ramsbøl Jensen, Anders Lindbjerg Dahl, George Vogiatzis, Engin Tola, and Henrik Aanæs. Large scale multi-view stereopsis evaluation. In CVPR, 2014. 5, 7, 8, 10

[7] Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy J. Mitra. Relu fields: The little non-linearity that could. In ACM TOG, 2022. 1

[8] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. ACM TOG, 2023. 3, 8, 9

[9] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: benchmarking large-scale scene reconstruction. ACM TOG, 2017. 5, 7, 9, 10

[10] 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. 8

[11] T. Berriel Martins and Javier Civera. Feature splatting for better novel view synthesis with low overlap. In BMVC, 2024. 8

[12] Nelson L. Max. Optical models for direct volume rendering. IEEE TVCG, 1995. 2

[13] Duane Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In ACM PACT, 2010. 3

[14] 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, 7, 8, 9

[15] 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. 8

[16] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. ACM TOG, 2022. 8

[17] 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. 8

[18] Johannes L. Schönberger, Enliang Zheng, Jan-Michael Frahm, and Marc Pollefeys. Pixelwise view selection for unstructured multi-view stereo. In ECCV, 2016. 8

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

[20] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, et al. Nerfstudio: A modular framework for neural radiance field development. In SIGGRAPH, 2023. 8

[21] Chandan Yeshwanth, Yueh-Cheng Liu, Matthias Nießner, and Angela Dai. Scannet++: A high-fidelity dataset of 3d indoor scenes. In ICCV, 2023. 8

[22] 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. 8

[23] Qingtian Zhu, Zizhuang Wei, Zhongtian Zheng, Yifan Zhan, Zhuyu Yao, Jiawang Zhang, Kejian Wu, and Yinqiang Zheng. Rpgb: Towards robust neural point-based graphics in the wild. In ECCV, 2024. 8

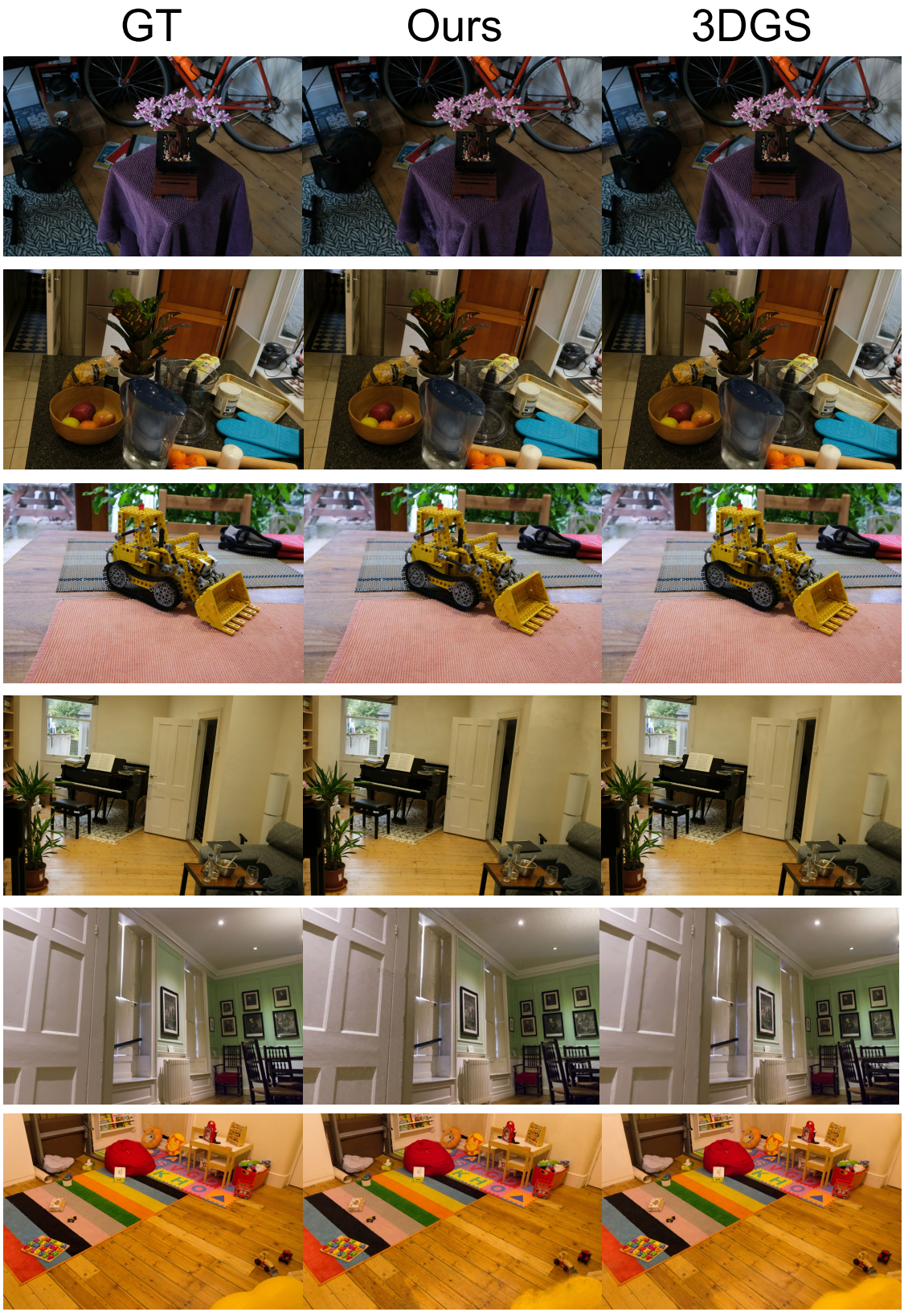

Figure 3. **Qualitative novel-view rendering results on-par with 3DGS.**

Figure 4. **Qualitative novel-view rendering results on-par with 3DGS.**

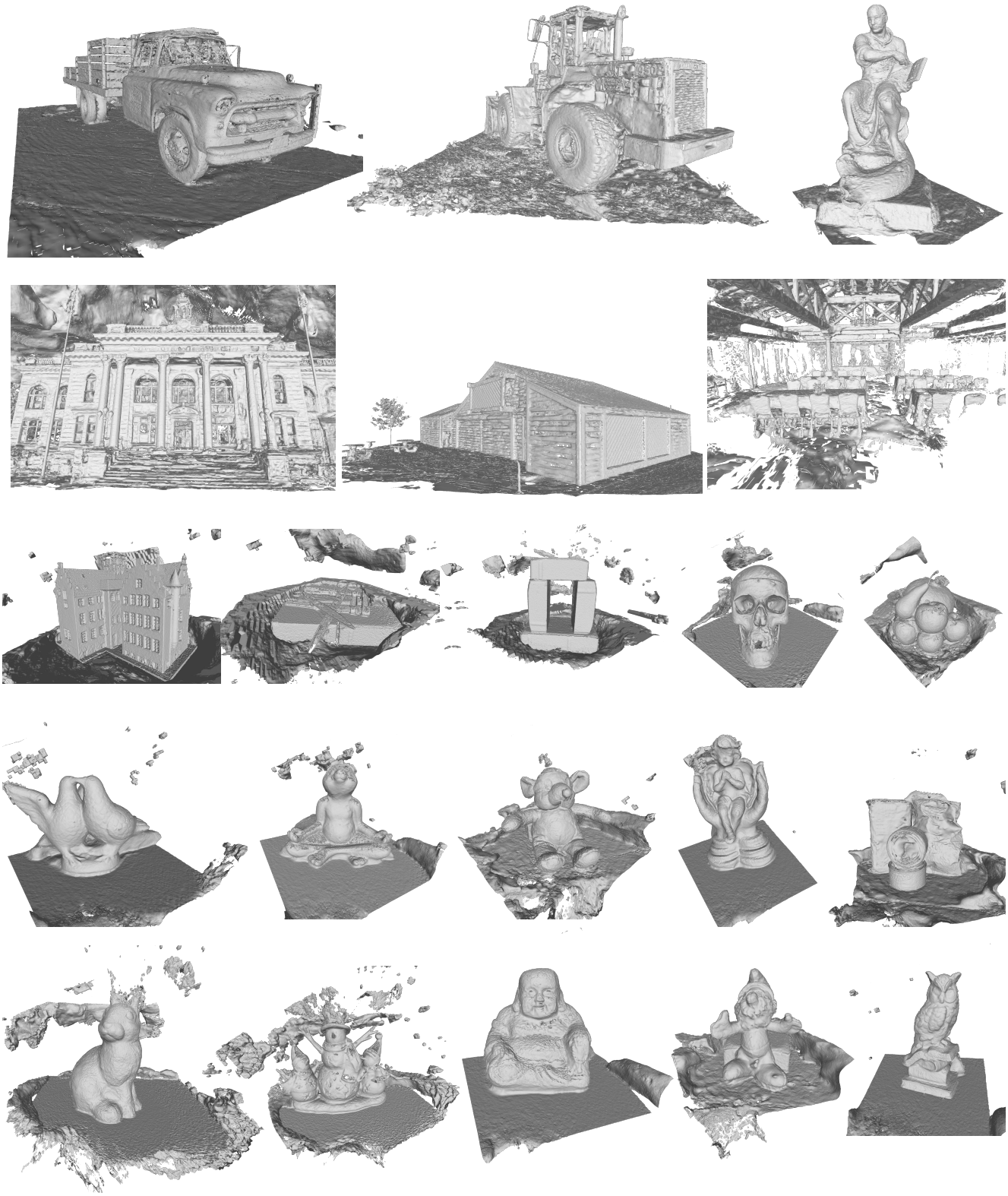

Figure 5. **Qualitative results of the reconstructed mesh.**