MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures

Zhiqin Chen[1,2,4] Thomas Funkhouser[1] Peter Hedman[1] Andrea Tagliasacchi[1,2,3,4]
Google Research[1] Simon Fraser University[2] University of Toronto[3]

Abstract

Neural Radiance Fields (NeRFs) have demonstrated amazing ability to synthesize images of 3D scenes from novel views. However, they rely upon specialized volumetric rendering algorithms based on ray marching that are mismatched to the capabilities of widely deployed graphics hardware. This paper introduces a new NeRF representation based on textured polygons that can synthesize novel images efficiently with standard rendering pipelines. The NeRF is represented as a set of polygons with textures representing binary opacities and feature vectors. Traditional rendering of the polygons with a z-buffer yields an image with features at every pixel, which are interpreted by a small, view-dependent MLP running in a fragment shader to produce a final pixel color. This approach enables NeRFs to be rendered with the traditional polygon rasterization pipeline, which provides massive pixel-level parallelism, achieving interactive frame rates on a wide range of compute platforms, including mobile phones.

Project page: https://mobile-nerf.github.io

1. Introduction

Neural Radiance Fields (NeRF) [33] have become a popular representation for novel view synthesis of 3D scenes. They represent a scene using a multilayer perceptron (MLP) that evaluates a 5D implicit function estimating the density and radiance emanating from any position in any direction, which can be used in a volumetric rendering framework to produce novel images. NeRF representations optimized to minimize multi-view color consistency losses for a set of posed photographs have demonstrated remarkable ability to reproduce fine image details for novel views.

One of the main impediments to wide-spread adoption of NeRF is that it requires specialized rendering algorithms that are poor match for commonly available hardware. Traditional NeRF implementations use a volumetric rendering algorithm that evaluates a large MLP at hundreds of sample positions along the ray for each pixel in order to estimate and integrate density and radiance. This rendering process is far too slow for interactive visualization.

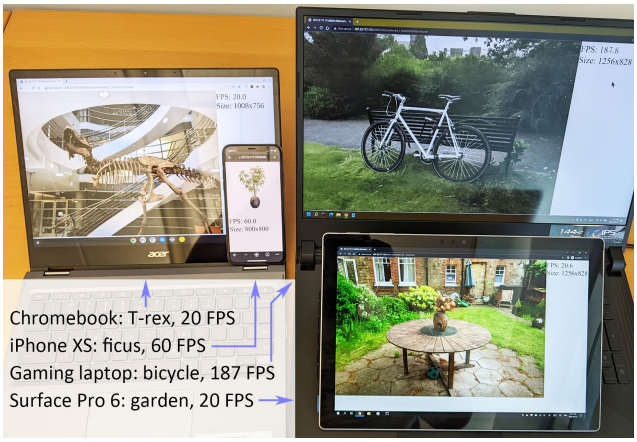

Figure 1. **Teaser –** We present a NeRF that can run on a variety of common devices at interactive frame rates.

Recent work has addressed this issue by "baking" NeRFs into a sparse 3D voxel grid [21, 51]. For example, Hedman et al. introduced Sparse Neural Radiance Grids (SNeRG) [21], where each active voxel contains an opacity, diffuse color, and learned feature vector. Rendering an image from SNeRG is split into two phases: the first uses ray marching to accumulate the precomputed diffuse colors and feature vectors along each ray, and the second uses a light-weight MLP operating on the accumulated feature vector to produce a view-dependent residual that is added to the accumulated diffuse color. This precomputation and deferred rendering approach increase the rendering speed of NeRF by three orders of magnitude. However, it still relies upon ray marching through a sparse voxel grid to produce the features for each pixel, and thus it cannot fully utilize the parallelism available in commodity graphics processing units (GPUs). In addition, SNeRG requires a significant amount of GPU memory to store the volumetric textures, which prohibits it from running on common mobile devices.

In this paper, we introduce MobileNeRF, a NeRF that

[4]Work done while at Google.

can run on a variety of common mobile devices at interactive frame rates. The NeRF is represented by a set of textured polygons, where the polygons roughly follow the surface of the scene, and the texture atlas stores opacity and feature vectors. To render an image, we utilize the classic polygon rasterization pipeline with Z-buffering to produce a feature vector for each pixel and pass it to a lightweight MLP running in a GLSL fragment shader to produce the output color. This rendering pipeline *does not* sample rays or sort polygons in depth order, and thus can model only binary opacities. However, it takes full advantage of the parallelism provided by z-buffers and fragment shaders in modern graphics hardware, and thus is $10\times$ faster than SNeRG with the same output quality on standard test scenes. Moreover, it requires only a standard polygon rendering pipeline, which is implemented and accelerated on virtually every computing platform, and thus it runs on mobile phones and other devices previously unable to support NeRF visualization at interactive rates.

Contributions. In summary, MobileNeRF:

- Is $10\times$ *faster* than the state-of-the-art (SNeRG), with the same output quality;
- Consumes less memory by storing *surface* textures instead of volumetric textures, enabling our method to run on integrated GPUs with limited memory and power;
- Runs on a web browser and is *compatible* with all devices we have tested, as our viewer is an HTML webpage;
- Allows real-time *manipulation* of the reconstructed objects/scenes, as they are simple triangle meshes.

2. Related work

Our work lies within the field of view-synthesis, which encompasses many areas of research: light fields, image-based rendering and neural rendering. To narrow the scope, we focus on methods that render output views in *real-time*.

Light fields [27] and Lumigraphs [19] store a dense grid of images, enabling real-time rendering of high quality scenes, albeit with limited camera freedom and significant storage overhead. Storage can be reduced by interpolating intermediate images with optical flow [5], representing the light field as a neural network [1], or by reconstructing a Multi-Plane Image (MPI) representation of the scene [15, 32, 37, 47, 54]. Multi-sphere images enable larger fields of view [2, 6], but these representations still only support limited output camera motion

Other approaches leverage explicit 3D geometry to enable more camera freedom. While early methods applied view-dependent texturing to a 3D mesh [7, 12, 13], later methods incorporated convolutional neural networks as a post-processing step to improve quality [20, 31, 44]. Alternatively, the input geometry can be simplified into a collection of textured planes with alpha [28]. Point-based representations further increase quality by jointly refining the scene geometry while training the post-processing network [24, 25, 40]. However, as this convolutional post-processing runs independently per output frame it often results in a lack of 3D consistency. Furthermore, unlike our work, they require powerful desktop GPUs and have not been demonstrated to run on a mobile device. Finally, unlike the vast majority of the methods above, our method does not need reconstructed 3D geometry as input.

It is also possible to extract explicit triangle meshes via differentiable inverse-rendering [11, 16, 35]. DefTet [16] differentiably renders a tetrahedral grid with occupancy and color at each vertex, and then compositing the interpolated values at all intersected faces along a ray. NVDiffRec [35] combines differentiable marching tetrahedra [42] with differentiable rasterization to perform full inverse rendering and extract triangle meshes, materials, and lighting from images. This representation enables elaborate editing and scene relighting. However, it incurs a significant loss in view-synthesis quality. Furthermore, while real-time rendering is possible with simple lighting, global illumination (GI) is computationally infeasible on mobile hardware. In contrast, our method simply caches the outgoing radiance, which does not need expensive compute to model GI effects, and also results in higher view-synthesis quality.

NeRF [33] represents the scene as a continuous field of opacity and view-dependent color, and produces images with volume rendering. This representation is 3D consistent and reaches high quality results [3, 45]. However, rendering a NeRF involves evaluating a large neural network at multiple 3D locations per pixel, preventing real-time rendering.

Recent works have improved the training speed of NeRF. For example, by modeling the opacity and color of entire ray segments instead of just points [29] or by subdividing the scene and modeling each sub-region with a smaller neural network [38]. Recently, significant speed-ups have been achieved by decoding features fetched from a 3D embedding with a small neural network. This embedding can either be a dense voxel grid [23, 43], a sparse voxel grid [41], a low-rank decomposition of a voxel grid [9], a point-based representation [50], or a multi-resolution hash map [34]. These 3D embeddings can also be used without a trained decoder, for example by directly storing diffuse colors [30] or by encoding view-dependent colors as spherical harmonics [41]. While these approaches drastically speed up training, they still require a large consumer GPU for rendering.

Rendering performance can further be increased by *post-processing* a trained NeRF. For example, by reducing the network queries per pixel with learned sampling [36], by evaluating the network for larger ray segments [48], or by subdividing the scene into smaller networks [38, 39, 49].

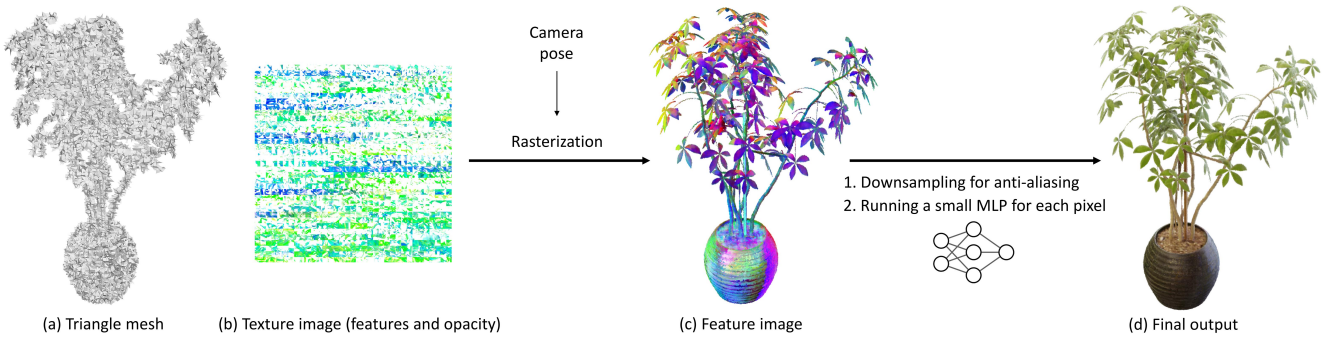

Figure 2. **Overview (rendering) –** We represent the scene as a triangle mesh textured by deep features. We first rasterize the mesh to a deferred rendering buffer. For each visible fragment, we execute a neural deferred shader that converts the feature and view direction to the corresponding output pixel color.

Alternatively, pre-computation can speed up rendering, by storing both scene opacity and a latent representation for view-dependent colors in a grid. FastNeRF [17] uses a dense voxel grid and represents view-dependence with a global spherical basis function. PlenOctrees [51] uses an octree representation, where each leaf node stores both opacity and spherical harmonics for colors. SNeRG [21] uses a sparse grid representation, and evaluates view-dependence as a post-process with a small neural network. Among these real-time methods, only SNeRG has been shown to work on lower-powered devices without access to CUDA. As our method directly targets rendering on low-powered hardware, we primarily compare with SNeRG in our experiments.

3. Method

Given a collection of (calibrated) images, we seek to optimize a representation for *efficient* novel-view synthesis. Our representation consists of a polygonal mesh (Figure 2a) whose texture maps (Figure 2b) store features and opacity. At rendering time, given a camera pose, we adopt a two-stage *deferred rendering* process:

- **Rendering Stage 1** – we rasterize the mesh to screen space and construct a *feature image* (Figure 2c), i.e. we create a deferred rendering buffer in GPU memory;
- **Rendering Stage 2** – we convert these features into a color image via a (neural) deferred renderer running in a fragment shader, i.e. a small MLP, which receives a feature vector and view direction and outputs a pixel color (Figure 2d).

Our representation is built in three *training* stages, gradually moving from a classical NeRF-like continuous representation towards a discrete one:

- **Training Stage 1 (Section 3.1)** – We train a NeRF-like model with *continuous* opacity, where volume rendering quadrature points are derived from the polygonal mesh;
- **Training Stage 2 (Section 3.2)** – We *binarize* the opacities, as while classical rasterization can easily discard fragments, they cannot elegantly deal with semi-transparent fragments.
- **Training Stage 3 (Section 3.3)** – We *extract* a sparse polygonal mesh, bake opacities and features into texture maps, and store the weights of the neural deferred shader.

The mesh is stored as an OBJ file, the texture maps in PNGs, and the deferred shader weights in a (small) JSON file. As we employ the standard GPU rasterization pipeline, our real-time renderer is simply an HTML webpage.

As representing continuous signals with discrete representations can introduce aliasing, we also detail a simple, yet computationally efficient, anti-aliasing solution based on super-sampling (Section 3.4).

3.1. Continuous training (Training Stage 1)

As Figure 3 shows, our *training setup* consists of a polygonal mesh $\mathcal{M}{=}(\mathcal{T},\mathcal{V})$ and three MLPs. The mesh topology $\mathcal{T}$ is fixed, but the vertex locations $\mathcal{V}$ and MLPs are optimized, similarly to NeRF, in an auto-decoding fashion by minimizing the mean squared error between predicted colors and ground truth colors of the pixels in the training images[1]:

$$\mathcal{L}_{\mathbf{C}} = \mathbb{E}_{\mathbf{r}} \|\mathbf{C}(\mathbf{r}) - \mathbf{C}_{\text{gt}}(\mathbf{r})\|_2^2. \tag{1}$$

where the predicted color $\mathbf{C}(.)$ is obtained by alpha-compositing the radiance $\mathbf{c}_k$ along a ray $\mathbf{r}(t){=}\mathbf{o}+t\mathbf{d}$, at the (depth sorted) quadrature points $\mathcal{K}{=}\{t_k\}_{k=1}^{K}$:

$$\mathbf{C}(\mathbf{r}) = \sum_{k=1}^{K} T_k \alpha_k \mathbf{c}_k, \quad T_k = \prod_{l=1}^{k-1}(1-\alpha_l) \tag{2}$$

[1]For real-world scenes, we further incorporate the distortion loss $\mathcal{L}_{dist}$ introduced by [3, Eq. 15] to suppress *floaters* and *background collapse*.

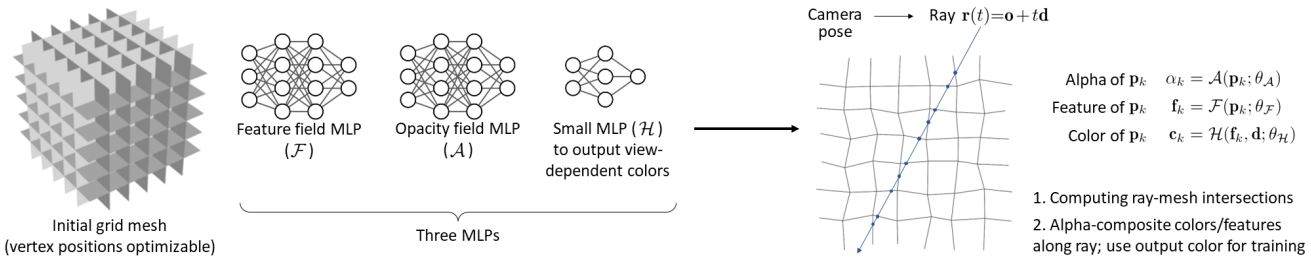

Figure 3. **Overview (train) –** We initialize the mesh as a regular grid, and use MLPs to represent features and opacity for any point on the mesh. For each ray, we compute its intersection points on the mesh, and alpha-composite the colors of those points to obtain the output color. In a later training stage, we enforce *binary* opacity, and perform super-sampling on features for anti-aliasing.

where *opacity* α_k and the view-dependent *radiance* $\mathbf{c}_k$ are given by evaluating the MLPs at position $\mathbf{p}_k{=}\mathbf{r}(t_k)$:

$$\alpha_k = \mathcal{A}(\mathbf{p}_k;\theta_\mathcal{A}) \qquad \mathcal{A}:\mathbb{R}^3 \rightarrow [0,1] \tag{3}$$

$$\mathbf{f}_k = \mathcal{F}(\mathbf{p}_k;\theta_\mathcal{F}) \qquad \mathcal{F}:\mathbb{R}^3 \rightarrow [0,1]^8 \tag{4}$$

$$\mathbf{c}_k = \mathcal{H}(\mathbf{f}_k,\mathbf{d};\theta_\mathcal{H}) \qquad \mathcal{H}:[0,1]^8\times[-1,1]^3 \rightarrow [0,1]^3 \tag{5}$$

The small network $\mathcal{H}$ is our *deferred neural shader*, which outputs the color of each fragment given the fragment feature and viewing direction. Finally, note that (2) does not perform compositing with volumetric density [33], but rather with opacity [1, Eq.8].

Polygonal mesh. Without loss of generality, we describe the polygonal mesh used in *Synthetic* 360° scenes, and provide the configurations for *Forward-Facing* and *Unbounded* 360° scenes in supplementary (Section H). 2D illustrations can be found in Figure 4. We first define a *regular* grid $\mathcal{G}$ of size $P{\times}P{\times}P$ in the unit cube centered at the origin; see Figure 4a. We instantiate $\mathcal{V}$ by creating one vertex per voxel, and $\mathcal{T}$ by creating one quadrangle (two triangles) per grid edge connecting the vertices of the four adjacent voxels, akin to Dual Contouring [10, 22]. We locally parameterize vertex locations with respect to the voxel centers (and sizes), resulting in $\mathcal{V}{\in}[-.5,+.5]^{P\times P\times P\times 3}$ free variables. During optimization, we initialize the vertex locations to $\mathcal{V}{=}\mathbf{0}$, which corresponds to a regular Euclidean lattice, and we regularize them to prevent vertices from exiting their voxels, and to promote their return to their neutral position whenever the optimization problem is under-constrained:

$$\mathcal{L}_\mathcal{V} = \sum_{\mathbf{v}\in\mathcal{V}} (10^3\,\mathcal{I}(\mathbf{v}) + 10^{-2})\cdot ||\mathbf{v}||_1, \tag{6}$$

where the indicator function $\mathcal{I}(\mathbf{v}){\equiv}1$ whenever $\mathbf{v}$ is outside its corresponding voxel.

Quadrature. As evaluating the MLPs of our representation is computationally expensive, we rely on an acceleration grid to limit the cardinality $|\mathcal{K}|$ of quadrature points. First of all, quadrature points are only generated for the set of voxels that intersect the ray; see Figure 5a: Then, like InstantNGP [34], we employ an acceleration grid $\mathcal{G}$ to prune voxels that are unlikely to contain geometry; see Figure 5b. Finally, we compute intersections between the ray and the faces of $\mathcal{M}$ that are incident to the voxel's vertex to obtain the final set of quadrature points; see Figure 5c. We use the barycentric interpolation to back-propagate the gradients from the intersection point to the three vertices in the intersected triangle. For further technical details on the computation of intersections, we refer the reader to supplementary (Section G). In summary, for each input ray $\mathbf{r}$:

$$\tilde{\mathcal{B}} = \text{intersect}(\mathbf{r},\mathcal{G}) \tag{7}$$

$$\mathcal{B} = \{b\in\tilde{\mathcal{B}} \mid \mathcal{G}[b] > \tau_\mathcal{G}\} \tag{8}$$

$$\mathcal{K} = \text{intersect}(\mathbf{r},\{\mathbf{t}\in\mathcal{T} \mid \mathbf{t}\cap\mathcal{B}\}) \tag{9}$$

where $(a\cap b){=}\textit{true}$ if a intersects b, and the acceleration grid is supervised so to upper-bound[2] the alpha-compositing visibility $T_k\alpha_k$ *across* viewpoints during training.

$$\mathcal{L}_\mathcal{G}^{\text{bnd}} = \sum_k \max(\not\nabla[T_k\alpha_k] - \mathcal{G}[\mathbf{p}_k], 0) \tag{10}$$

where $\not\nabla[.]$ is the stop-gradient operator that prevents the acceleration grid from (negatively) affecting the image reconstruction quality. This can be interpreted as a way to compute the so-called "surface field" *during* NeRF training, as opposed to *after* training as in nerf2nerf [18]. Similarly to Plenoxels [41], we additionally regularize the content of the grid by promoting its pointwise sparsity (i.e. lasso), and its spatial smoothness:

$$\mathcal{L}_\mathcal{G}^{\text{sparse}} = \|\mathcal{G}\|_1^1 \qquad \mathcal{L}_\mathcal{G}^{\text{smooth}} = \|\nabla\mathcal{G}\|_2^2 \tag{11}$$

3.2. Binarized training (Training Stage 2)

Rendering pipelines implemented in typical hardware *do not* natively support semi-transparent meshes. Rendering semi-transparent meshes requires cumbersome (per-frame) sorting so to execute rendering in back-to-front order to guarantee correct alpha-compositing. We overcome this issue by converting the smooth opacity $\alpha_k{\in}[0,1]$ from (3) to

[2] This loss performs a *stochastic upper-bound*, as we initialize $\mathcal{G}[*]{=}\mathbf{0}$, and $\mathcal{G}[\mathbf{p}_k]$ receives gradients whenever $T_k\alpha_k{>}\mathcal{G}[\mathbf{p}_k]$.

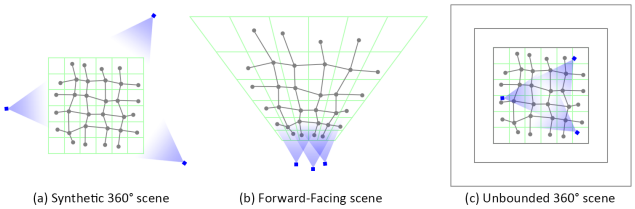

Figure 4. **Configurations of polygonal meshes –** The meshes employed for the different types of scenes. We sketch the distribution of camera poses in training views.

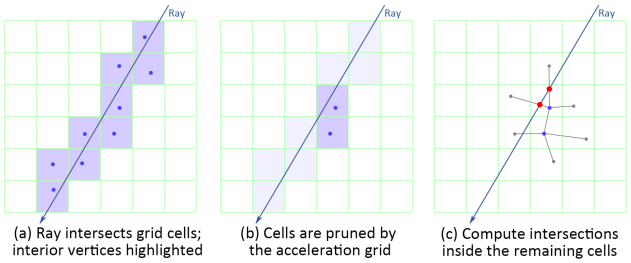

Figure 5. **Quadrature points –** are obtained by (a) identifying cells that intersect the ray; (b) pruning cells that do not contain geometry; and, (c) computing explicit intersections with the mesh.

a discrete/categorical opacity $\hat{\alpha}_k \in \{0,1\}$. To optimize for discrete opacities via photometric supervision we employ a *straight-through estimator* [4]:

$$\hat{\alpha}_k = \alpha_k + \not{\nabla}[\mathbb{1}(\alpha_k > 0.5) - \alpha_k] \tag{12}$$

Please note that the gradients are transparently passed through the discretization operation (i.e. $\nabla\hat{\alpha} \equiv \nabla\alpha$), regardless of the values of α_k and the resulting $\hat{\alpha}_k \in \{0,1\}$. To stabilize training, we then co-train the continuous and discrete models:

$$\mathcal{L}_{\mathbf{C}}^{\text{bin}} = \mathbb{E}_{\mathbf{r}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}_{\text{gt}}(\mathbf{r})\|_2^2 \tag{13}$$

$$\mathcal{L}_{\mathbf{C}}^{\text{stage2}} = \tfrac{1}{2}\mathcal{L}_{\mathbf{C}}^{\text{bin}} + \tfrac{1}{2}\mathcal{L}_{\mathbf{C}} \tag{14}$$

where $\hat{\mathbf{C}}(\mathbf{r})$ is the output radiance corresponding to the discrete opacity model $\hat{\alpha}$:

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{k=1}^{K} \hat{T}_k \hat{\alpha}_k \mathbf{c}_k, \quad \hat{T}_k = \prod_{l=1}^{k-1}(1-\hat{\alpha}_l) \tag{15}$$

Once (14) has converged, we will apply a fine-tuning step to the weights in $\mathcal{F}$ and $\mathcal{H}$ by minimizing $\mathcal{L}_{\mathbf{C}}^{\text{bin}}$, while fixing the weights of others.

3.3. Discretization (Training Stage 3)

After binarization and fine-tuning, we convert the representation into an explicit polygonal mesh (in OBJ format). We only store quads if they are at least partially visible in the training camera poses (i.e. non-visible quads are discarded). We then create a texture image whose size is proportional to the number of visible quads, and for each quad we allocate a $K \times K$ patch in the texture, similarly to Disney's Ptex [8]. We use $K=17$ in our experiments, so that the quad has a 16×16 texture with half-a-pixel boundary padding. We then iterate over the pixels of the texture, convert the pixel coordinate to 3D coordinates, and *bake* the values of the discrete opacity (i.e. (3) and (12)) and features (i.e. (4)) into the texture map. We quantize the $[0,1]$ ranges to 8-bit integers, and store the texture into (losslessly compressed) PNG images. Our experiments show that quantizing the $[0,1]$ range with 8-bit precision, which is not accounted for during back-propagation, does not significantly affect rendering quality.

3.4. Anti-aliasing

In classic rasterization pipelines, aliasing is an issue that ought to be considered to obtain high-quality rendering. While classical NeRF hallucinates smooth edges via semi-transparent volumes, as previously discussed, semi-transparency would require per-frame polygon sorting. We overcome this issue by employing anti-aliasing by super-sampling. While we could simply execute (5) four times/pixel and average the resulting color, the execution of the deferred neural shader $\mathcal{H}$ is the computational bottleneck of our technique. We can overcome this issue by simply averaging the features, that is, *averaging the input* of the deferred neural shader, rather than averaging its output. We first rasterize features (at $2\times$ resolution):

$$\mathbf{F}(\mathbf{r}) = \sum_k T_k \alpha_k \mathbf{f}_k, \tag{16}$$

and then average sub-pixel features to produce the anti-aliased representation we feed to our neural deferred shader:

$$\mathbf{C}(\mathbf{r}) = \mathcal{H}\left(\mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}[\mathbf{F}(\mathbf{r}_\delta)],\ \mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}[\mathbf{d}_\delta]\right) \tag{17}$$

where $\mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}$ computes the average between the sub-pixels (i.e. four in our implementation), and $\mathbf{d}_\delta$ is the direction of ray $\mathbf{r}_\delta$. Note how with this change we only query $\mathcal{H}$ *once* per output pixel. Finally, this process is analogously applied to (15) for discrete occupancies $\hat{\alpha}$. These changes for anti-aliasing are applied in training stage 2 (14).

3.5. Rendering

The result of the optimization process is a textured polygonal mesh (where texture maps store features rather than colors) and a small MLP (which converts view direction and features to colors). Rendering this representation is done in two passes using a deferred rendering pipeline:

1. we rasterize all faces of the textured mesh with a z-buffer to produce a $2M \times 2N$ feature image with 12 channels per pixel, comprising 8 channels of learned features, a binary opacity, and a 3D view direction;

2. we synthesize an $M \times N$ output RGB image by rendering a textured rectangle that uses the feature image as its texture, with linear filtering to average the features for antialiasing. We apply the small MLP for pixels with non-zero alphas to convert features into RGB colors. The small MLP is implemented as a GLSL fragment shader.

These rendering steps are implemented within the classic rasterization pipeline. Since z-buffering with binary transparency is order-independent, polygons *do not* need to be sorted into depth-order for each new view, and thus can be loaded into a buffer in the GPU once at the start of execution. Since the MLP for converting features to colors is very small, it can be implemented in a GLSL fragment shader [21], which is run in parallel for all pixels. These classical rendering steps are highly-optimized on GPUs, and thus our rendering system can run at interactive frame rates on a wide variety of devices; see Table 2. It is also easy to implement, since it requires only standard polygon rendering with a fragment shader. Our interactive viewer is an HTML webpage with Javascript, rendered by WebGL via the `threejs` library.

4. Experiments

We run a series of experiments to test how well MobileNeRF performs on a wide variety of scenes and devices. We test on three datasets: the 8 synthetic 360° scenes from NeRF [33], the 8 forward-facing scenes from LLFF [32], and 5 unbounded 360° outdoor scenes from Mip-NeRF 360 [3]. We compare with SNeRG [21], since, to our knowledge, it is the only NeRF model that can run in real-time on common devices. We also include extensive ablation studies to investigate the impact of different design choices.

4.1. Comparisons

To show the superior performance and compatibility of our method, we test our method and SNeRG on a variety of devices, as shown in Table 1. We report the rendering speed in Table 2. The rendering resolution is the same as the training images: 800×800 for synthetic, 1008×756 for forward-facing, and 1256×828 for unbounded. We test all methods on a chrome browser and rotate/pan the camera in a full circle to render 360 frames. Note that SNeRG is unable to represent unbounded 360° scenes due to its regular grid representation, and it does not run on phone or tablet due to compatibility or out-of-memory issues. We also report the GPU memory consumption and storage cost in Table 3. MobileNeRF requires 5x less GPU memory than SNeRG.

Rendering quality. We report the rendering quality in Table 4, while comparing with other methods using the common PSNR, SSIM [46], and LPIPS [53] metrics. Our method has roughly the same image quality as SNeRG, and better than NeRF. Visual results are shown in Figure 6 (a-c). Our method achieves image quality similar to SNeRG when the camera is at an appropriate distance. When the camera is zoomed in, SNeRG tends to render over-smoothed images.

Polygon count. Table 5 shows the average number of vertices and triangles produced by our method, and the per-

Device	Type	OS	GPU	Power
iPhone XS	Phone	iOS 15	Integrated GPU	6W
Pixel 3	Phone	Android 12	Integrated GPU	9W
Surface Pro 6	Tablet	Windows 10	Integrated GPU	15W
Chromebook	Laptop	Chrome OS	Integrated GPU	15W
Gaming laptop	Laptop	Windows 11	NVIDIA RTX 2070	115W
Desktop	PC	Ubuntu 16.04	NVIDIA RTX 2080 Ti	250W

Table 1. **Hardware specs –** of the devices used in our rendering experiments. The power is the max GPU power for discrete NVIDIA cards, and the combined max CPU and GPU power for integrated GPUs.

Dataset	Synthetic 360°		Forward-facing		Unbounded 360°
Method	Ours	SNeRG	Ours	SNeRG	Ours
iPhone XS	**55.89**	0.0$\frac{8}{8}$	**27.19**$\frac{2}{8}$	0.0$\frac{8}{8}$	22.20$\frac{4}{5}$
Pixel 3	**37.14**	0.0$\frac{8}{8}$	**12.40**	0.0$\frac{8}{8}$	9.24
Surface Pro 6	**77.40**	Unsupported	**21.51**	Unsupported	19.44
Chromebook	**53.67**	22.62$\frac{2}{8}$	**19.44**	7.85$\frac{3}{8}$	15.28
Gaming laptop	**178.26**	8.30$\frac{1}{8}$	**57.72**	3.63	55.32
Gaming laptop 🔌	**606.73**	43.87$\frac{1}{8}$	**250.17**	26.01	192.59
Desktop 🔌	**744.91**	207.26	**349.34**	50.71	279.70

Table 2. **Rendering speed –** on various devices in frames per second (FPS). The devices are on battery, except for the gaming laptop and the desktop which are plugged in, indicated with a 🔌. The mobile devices (first four rows) have almost identical rendering speed when plugged in. With the notation $\frac{M}{N}$ we indicate that M out of N testing scenes failed to run due to out-of-memory errors.

Dataset	Synthetic 360°		Forward-facing		Unbounded 360°
Method	Ours	SNeRG	Ours	SNeRG	Ours
GPU memory	**538.38**	2707.25	**759.25**	4312.13	1162.20
Disk storage	125.75	**86.75**	**201.50**	337.25	344.60

Table 3. **Resources –** memory and disk storage (MB).

	Synthetic 360°			Forward-facing			Unbounded 360°		
	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓
NeRF	31.00	0.947	0.081	26.50	0.811	0.250	-	-	-
JAXNeRF	31.65	0.952	0.051	26.92	0.831	0.173	21.46	0.458	0.515
NeRF++	-	-	-	-	-	-	22.76	0.548	0.427
SNeRG	30.38	**0.950**	**0.050**	25.63	0.818	**0.183**	-	-	-
Ours	**30.90**	0.947	0.062	**25.91**	**0.825**	**0.183**	21.95	0.470	0.470

Table 4. **Quantitative Analysis –** For NeRF [33] and NeRF++ [52], we dash entries where the original papers did not report quantitative performance. For SNeRG, while one could extend the method to include the unbounded design from [3], implementing this is far from trivial. Our method can be easily adapted to work across all modalities.

	Synthetic 360°		Forward-facing		Unbounded 360°	
	V	T	V	T	V	T
Number	494,289	224,341	830,076	338,535	1,436,033	608,785
Percentage	1.964%	1.783%	3.298%	2.690%	4.891%	4.147%

Table 5. **Polygon count –** Average number of vertices and triangles produced, and their percentage compared to all available vertices/triangles in the initial mesh.

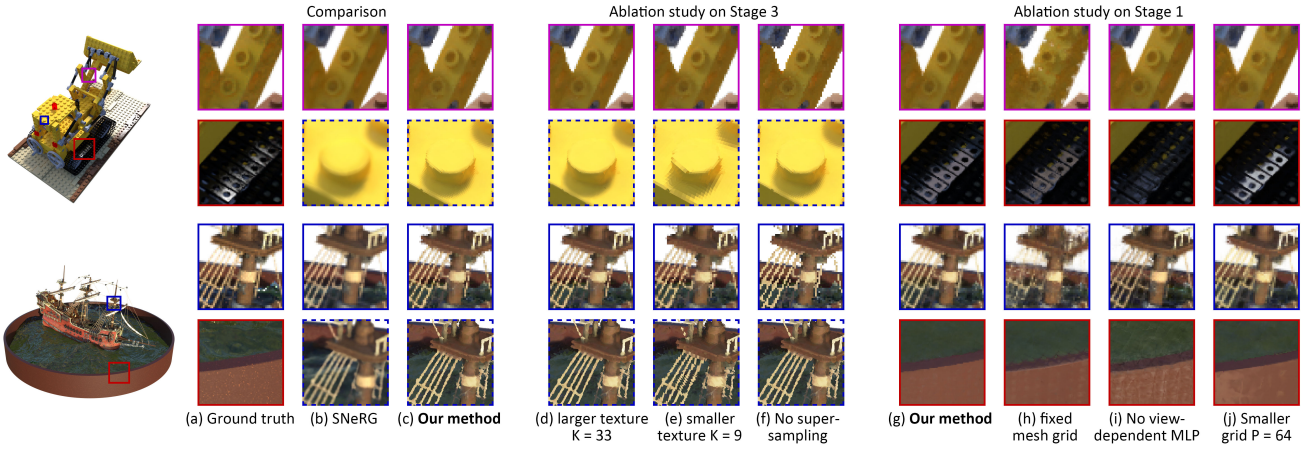

Figure 6. **Qualitative Results –** Comparisons to the state-of-the-art and ablation studies. With a *solid* line we denote zoom-ins of the rendered (800×800) image, while with a *dashed* line we move the camera to zoom-in onto the same detail.

	Synthetic 360°		Forward-facing	
	PSNR↑	SSIM↑	PSNR↑	SSIM↑
Stage 1, **our method**	**32.13**	**0.955**	26.57	**0.839**
Stage 1, fixed mesh grid	29.87	0.938	25.43	0.797
Stage 1, no view-dependent MLP	29.91	0.935	25.91	0.824
Stage 1, smaller grid P=128 → 64	31.58	0.952	26.39	0.831
Stage 1, no acceleration grid	31.77	0.953	**26.61**	0.835
Stage 2, **our method**	**31.01**	**0.948**	**26.32**	**0.833**
Stage 2, no fine-tuning	30.80	0.946	26.25	0.832
Stage 2, only pseudo-gradients	29.70	0.935	26.01	0.820
Stage 2, binary loss	30.89	0.947	**26.32**	0.832
Stage 3, **our method**	30.90	0.947	25.91	0.825
Stage 3, larger texture K=17 → 33	**30.99**	**0.948**	**26.14**	**0.830**
Stage 3, smaller texture K=17 → 9	30.49	0.945	24.85	0.796
Stage 3, no supersampling	29.26	0.937	24.88	0.799

Table 6. **Ablation –** rendering quality.

	Speed in FPS			Space in MB	
Synthetic 360° scenes	Pixel 3	Surface Pro 6	Gaming laptop	GPU memory	Disk storage
our method	37.14	77.40	606.73	538.38	125.75
Larger texture $K = 33$	32.48	59.15	589.20	1290.88	283.50
Smaller texture $K = 9$	37.74	94.62	617.74	**336.63**	**67.00**
No supersampling	51.81	**113.41**	**649.86**	440.25	125.75
No view-dependent MLP	**52.16**	96.76	638.30	538.38	125.75
Forward-facing scenes	Pixel 3	Surface Pro 6	Gaming laptop	GPU memory	Disk storage
our method	12.40	21.51	250.17	759.25	201.50
Larger texture $K = 33$	12.88	18.79	241.52	2024.13	462.75
Smaller texture $K = 9$	12.70	23.61	257.64	**394.13**	**105.75**
No supersampling	16.97	**42.11**	**413.02**	645.00	201.50
No view-dependent MLP	**23.72**	28.06	385.65	759.25	201.50

Table 7. **Ablation –** rendering speed/memory.

centage compared to all available vertices/triangles in the initial mesh. As we only retain visible triangles, most vertices/triangles are removed in the final mesh.

Shading mesh. In Figure 2a and Figure 7, we show the extracted triangle meshes without the textures. Most triangle faces do not align with the actual object surface. This is perhaps due to the ambiguity that good rendering quality can be achieved despite how the triangles are aligned. For example, the results of our method after Stage 1 in Table 6 is similar to other methods in Table 4. Therefore, better regularization losses or training objectives need to be devised if one wishes to have better surface quality. However, optimizing vertices does improve the rendering quality, as shown in Figure 6h.

4.2. Ablation studies

In Table 6, we show the rendering quality of our method at each stage, and report our ablation studies. The rendering quality gradually drops after each stage, because each stage adds more constraints to the model. In Stage 1, the performance drops significantly if we use a fixed regular grid mesh instead of having optimizable mesh vertices, or

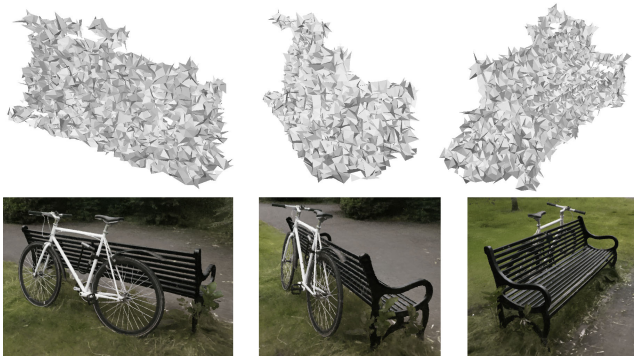

Figure 7. **Shading mesh** – not textured. The mesh corresponds to the bicycle (see Figure 1). We manually removed the background mesh to better show the geometry of the object. Zoom-in to see more details. In the bottom, we also show the rendered images of our method. Note how the coarse mesh is able to represent detailed structures such as the spokes of the wheels and the wires around the handles, thanks to high-resolution textures with transparencies.

if we forgo view-dependent effects by directly predicting the color and alpha of each point. The performance drops slightly if the grid is smaller (P=64 vs. 128). If we remove the acceleration grid, we are not able to quadruple the batch

Figure 8. **Limitations –** (a) the monitor/table are hollow, because the reflections are modelled as real objects behind the monitor and below the table. (b) our method generates scattered small fragments in the semi-transparent parts. (c) the camera is too close to the scene and details in the grass cannot be represented at the chosen texture resolution.

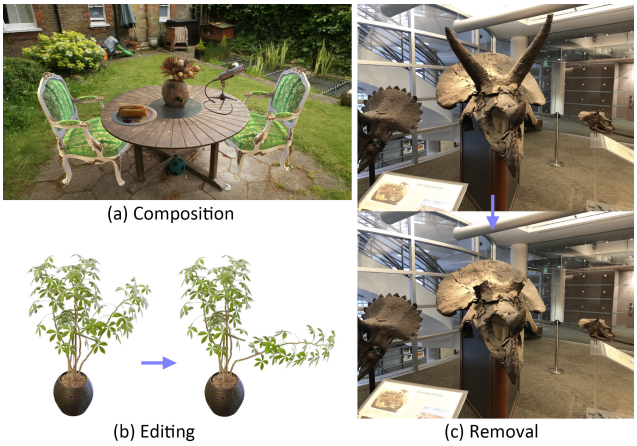

Figure 9. **Scene editing –** (a) four objects learned from the synthetic scenes are added into an unbounded scene. (b) a branch of the ficus is bent. (c) the horns are removed.

size during training; the performance drops if we train this model the same number of iterations as our method. Note that the PSNR of this model is higher on forward-facing scenes. This is because the acceleration grid will remove cells that are not visible in the training images, thus cannot "inpaint" the objects and may leave holes. In Stage 2, if we do not perform the fine-tuning step that only optimizes $\mathcal{F}$ and $\mathcal{H}$ and fix the weights of others, the performance drops. If we only use the binary opacity with pseudo-gradients by applying $\mathcal{L}_{\mathbf{C}}^{\text{stage2}}=\mathcal{L}_{\mathbf{C}}^{\text{bin}}$ instead of Eq. 14, the performance drops. If we use a binary loss on the predicted opacity, e.g., $\mathcal{L}_{binary}=-\sum|\alpha_k-0.5|$, instead of using the pseudo-gradients with $\hat{\mathbf{C}}(\mathbf{r})$, the performance drops slightly. In stage 3, when we use a larger texture size K=33 instead of 17, the performance improves, but the texture size will be quadrupled; the performance drops when we use a smaller texture size K=9. If we remove the super-sampling step, the performance drops significantly. Visual results are shown in Figure 6. We omit some models because they do not have significant visual differences compared to our method. Notice the squared pixels of the texture images are clearly visible in the dashed-line boxes in (e) and almost invisible in (d). The aliasing artifacts are conspicuous in the solid-line boxes in (f). In Stage 1, if the grid vertices cannot be optimized, the results will be significantly worse, as shown in (h). Without the small MLP, the model cannot handle reflections, as shown in (i). Changing to a smaller grid size introduces some minor artifacts in (j). In Table 7. we show the rendering speed and space cost if we use a larger or smaller texture size, or if we remove the super-sampling step, or if we only perform the rasterization without using the small MLP to predict the view-dependent colors. One can find that the super-sampling step and the small MLP have the most significant impact.

5. Conclusions

We introduce MobileNeRF, an architecture that takes advantage of the classical rasterization pipeline (i.e. z-buffers and fragment shaders) to perform efficient rendering of surface-based neural fields on a wide range of compute platforms. It achieves frame rates an order of magnitude faster than the previous state-of-the-art (SNeRG) while producing images of equivalent quality.

Limitations. Our estimated *surface* may be incorrect, especially for scenes with specular surfaces and/or sparse views (Figure 8a); it uses *binary* opacities to avoid sorting polygons, and thus cannot handle scenes with semi-transparencies (Figure 8b); it uses fixed mesh and texture resolutions, which may be too coarse for close-up novel-view synthesis (Figure 8c); it models a radiance field without explicitly decomposing illumination and reflectance, and thus does not handle glossy surfaces as well as recent methods [45]. Extending the polygon rendering pipeline with efficient partial sorting, levels-of-detail, mipmaps, and surface shading should address some of these issues. Also, the current training speed of MobileNeRF is slow due to NeRF's MLP backbone. The extension of MobileNeRF to fast-training architectures (e.g., Instant NGP [34]) constitutes an exciting avenue for future works.

The explicit mesh representation provided by MobileNeRF gives us direct editing control over the NeRF model without any complex architectural change (e.g. ControlNerf [26]), but in this paper we only superficially investigated these possibilities; see Figure 9 and the videos in the supplementary (Section C).

Acknowledgements. We thank the reviewers as well as Simon Kornblinth, Ting Chen, Daniel Rebain, Kevin Swersky, and David Fleet for their valuable feedback.

References

[1] Benjamin Attal, Jia-Bin Huang, Michael Zollhöfer, Johannes Kopf, and Changil Kim. Learning neural light fields with ray-space embedding networks. *CVPR*, 2022. 2, 4

[2] Benjamin Attal, Selena Ling, Aaron Gokaslan, Christian Richardt, and James Tompkin. MatryODShka: Real-time 6DoF video view synthesis using multi-sphere images. *ECCV*, 2020. 2

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022. 2, 3, 6

[4] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 5

[5] Tobias Bertel, Mingze Yuan, Reuben Lindroos, and Christian Richardt. OmniPhotos: Casual 360° VR photography. *ACM Transactions on Graphics*, 2020. 2

[6] Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew DuVall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. Immersive light field video with a layered mesh representation. *ACM Transactions on Graphics*, 2020. 2

[7] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *Proceedings of Computer graphics and interactive techniques*, 2001. 2

[8] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Computer Graphics Forum*, 2008. 5

[9] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. *ECCV*, 2022. 2

[10] Zhiqin Chen, Andrea Tagliasacchi, Thomas Funkhouser, and Hao Zhang. Neural dual contouring. *ACM Transactions on Graphics*, 2022. 4

[11] Forrester Cole, Kyle Genova, Avneesh Sud, Daniel Vlasic, and Zhoutong Zhang. Differentiable surface rendering via non-differentiable sampling. In *ICCV*, 2021. 2

[12] Abe Davis, Marc Levoy, and Fredo Durand. Unstructured light fields. *Computer Graphics Forum*, 2012. 2

[13] Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Workshop on Rendering Techniques*, 1998. 2

[14] Boyang Deng, Jonathan T. Barron, and Pratul P. Srinivasan. JaxNeRF: an efficient JAX implementation of NeRF, 2020. 4, 5, 6

[15] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. DeepView: View synthesis with learned gradient descent. *CVPR*, 2019. 2

[16] Jun Gao, Wenzheng Chen, Tommy Xiang, Clement Fuji Tsang, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Learning deformable tetrahedral meshes for 3d reconstruction. In *NeurIPS*, 2020. 2

[17] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien P. C. Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *ICCV*, 2021. 3

[18] Lily Goli, Daniel Rebain, Sara Sabour, Animesh Garg, and Andrea Tagliasacchi. nerf2nerf: Pairwise registration of neural radiance fields. *ICRA*, 2023. 4

[19] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *SIGGRAPH*, 1996. 2

[20] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics*, 2018. 2

[21] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. *ICCV*, 2021. 1, 3, 6, 4, 5

[22] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of Hermite data. *ACM Transactions on graphics*, 2002. 4

[23] Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy J. Mitra. Relu fields: The little non-linearity that could. *ACM Transactions on Graphics*, 2022. 2

[24] Georgios Kopanas, Julien Philip, Thomas Leimkühler, and George Drettakis. Point-based neural rendering with per-view optimization. In *Computer Graphics Forum*, 2021. 2

[25] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. *CVPR*, 2021. 2

[26] Verica Lazova, Vladimir Guzov, Kyle Olszewski, Sergey Tulyakov, and Gerard Pons-Moll. Control-nerf: Editable feature volumes for scene rendering and manipulation. *arXiv preprint arXiv:2204.10850*, 2022. 8

[27] Marc Levoy and Pat Hanrahan. Light field rendering. *SIGGRAPH*, 1996. 2

[28] Zhi-Hao Lin, Wei-Chiu Ma, Hao-Yu Hsu, Yu-Chiang Frank Wang, and Shenlong Wang. Neurmips: Neural mixture of planar experts for view synthesis. *CVPR*, 2022. 2

[29] David B. Lindell, Julien N.P. Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural rendering. *CVPR*, 2021. 2

[30] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *SIGGRAPH*, 2019. 2

[31] Ricardo Martin-Brualla, Rohit Pandey, Shuoran Yang, Pavel Pidlypenskyi, Jonathan Taylor, Julien Valentin, Sameh Khamis, Philip Davidson, Anastasia Tkach, Peter Lincoln, Adarsh Kowdle, Christoph Rhemann, Dan B Goldman, Cem Keskin, Steve Seitz, Shahram Izadi, and Sean Fanello. Lookingood: Enhancing performance capture with real-time neural re-rendering. *ACM Transactions on Graphics*, 2018. 2

[32] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics*, 2019. 2, 6

[33] 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. 1, 2, 4, 6, 5

[34] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics*, 2022. 2, 4, 8

[35] Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. Extracting triangular 3d models, materials, and lighting from images. In *CVPR*, 2022. 2

[36] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H Mueller, Chakravarty R Alla Chaitanya, Anton Kaplanyan, and Markus Steinberger. Donerf: Towards real-time rendering of compact neural radiance fields using depth oracle networks. In *Computer Graphics Forum*, 2021. 2

[37] Eric Penner and Li Zhang. Soft 3D reconstruction for view synthesis. *ACM Transactions on Graphics*, 2017. 2

[38] Daniel Rebain, Wei Jiang, Soroosh Yazdani, Ke Li, Kwang Moo Yi, and Andrea Tagliasacchi. DeRF: Decomposed radiance fields. *CVPR*, 2021. 2

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

[40] Darius Rückert, Linus Franke, and Marc Stamminger. Adop: Approximate differentiable one-pixel point rendering. *arXiv preprint arXiv:2110.06635*, 2021. 2

[41] Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *CVPR*, 2022. 2, 4

[42] Tianchang Shen, Jun Gao, Kangxue Yin, Ming-Yu Liu, and Sanja Fidler. Deep marching tetrahedra: a hybrid representation for high-resolution 3d shape synthesis. In *NeurIPS*, 2021. 2

[43] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *CVPR*, 2022. 2

[44] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics*, 2019. 2

[45] Dor Verbin, Peter Hedman, Ben Mildenhall, Todd Zickler, Jonathan T. Barron, and Pratul P. Srinivasan. Ref-NeRF: Structured view-dependent appearance for neural radiance fields. *CVPR*, 2022. 2, 8

[46] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004. 6

[47] Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. Nex: Real-time view synthesis with neural basis expansion. *CVPR*, 2021. 2

[48] Liwen Wu, Jae Yong Lee, Anand Bhattad, Yuxiong Wang, and David Forsyth. Diver: Real-time and accurate neural radiance fields with deterministic integration for volume rendering. *CVPR*, 2022. 2

[49] Xiuchao Wu, Jiamin Xu, Zihan Zhu, Hujun Bao, Qixing Huang, James Tompkin, and Weiwei Xu. Scalable neural indoor scene rendering. *ACM Transactions on Graphics*, 2022. 2

[50] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. *CVPR*, 2022. 2

[51] 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. 1, 3

[52] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun. Nerf++: Analyzing and improving neural radiance fields. *arXiv preprint arXiv:2010.07492*, 2020. 6

[53] 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. 6

[54] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *ACM Transactions on Graphics*, 2018. 2

MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures

(Supplementary Material)

A. More results

Check out our project page: https://mobile-nerf.github.io

The models used in the online demos of our project page are the same as the ones used in our paper. The rendered images of our method are nearly identical whether they are rendered in Python (for computing quantitative metrics) or web browsers, see Figure 10. If one overlays the difference image and the rendered image, one can find that the few very different pixels are all on the boundary of a part, which indicates that they are likely caused by precision errors in rasterization.

B. [Post-submission] Shader code optimization

With the optimizations suggested by Noeri Huisman, we have greatly improved the rendering speed of the fragment shader containing the small MLP $\mathcal{H}$. specifically, we

- Inject network weights directly into the shader source code, instead of using weight textures and texel fetches;
- Use mat4 and vec3 multiplications in all operations, instead of multiplying and adding float numbers.

When tested on the 5 real unbounded scenes with a Samsung Galaxy S22 Ultra mobile phone, the average FPS is 35 using the optimized implementation, which is 35% faster than our original implementation (26 FPS).

Note that our default rendering setting is **deferred rendering**, that is, meshes and textures are rasterized into a feature image with pixels containing features, and then we treat this feature image as the texture image of a rectangle polygon mesh and rasterize it into the final output image with pixels containing RGB colors using our MLP fragment shader. This design was to ensure that the MLP shader is executed once per output pixel. in contrast, **forward rendering**, where the meshes and textures are directly rasterized into pixels containing RGB colors using the MLP fragment shader, will need to execute the MLP shader once per fragment. Since the number of fragments are usually much larger than the number of output pixels, the deferred rendering setting has a speed advantage over forward rendering when the speed of the MLP shader is a bottleneck in rendering.

Now that the optimized MLP fragment shader is so fast, we have observed that forward rendering is much faster than deferred rendering on some mobile devices. When tested on the 5 real unbounded scenes with a Samsung Galaxy S22 Ultra mobile phone, the average FPS is 84 using the optimized forward rendering implementation, which is 223% faster than our original deferred rendering implementation (26 FPS) and 140% faster than the optimized deferred rendering implementation (35 FPS).

Note that forward rendering is still slower than deferred rendering on some devices due to excessive overdraw. A depth pre-pass can help resolve this issue, but it may not be available on certain devices.

We provide demos for both forward and deferred rendering settings on our project page: https://mobile-nerf.github.io. However, the models in these demos were trained with deferred rendering, where super-sampling is done on pixel features, while the super-sampling in forward rendering should be done on pixel colors. Therefore the rendered results may be slightly different.

C. Scene editing

Our representation is a textured mesh with baked lighting, and thus can be used in any application that combines, renders, or manipulates such meshes. Figure 9 (a) shows a simple example where meshes learned from four different sets of photos are composited into a single scene. The scene, rendered in 1920×1080 resolution without super-sampling, runs at 150 FPS on the gaming laptop, and consumes 1.5 GB of GPU memory. Similarly Figure 9 (b)(c) show scenes where some parts or objects are edited or removed by manipulating the triangle meshes of the scenes in a 3D modeling software. The resulting renders do not account for differences in illumination between the captured photos or indirect illumination between different meshes. However, it suggests an easy way to create "photorealistic-looking" scenes from a library of objects captured using photos rather than painstaking 3D modeling.

https://youtu.be/kVy2W6afuyk shows three examples where we manipulate the learned NeRF objects interactively in real-time. We also highlight how easy it is to implement these operations with our mesh representation. In contrast, implementing those with classic NeRF is non-trivial.

In the first example, we render all 8 objects learned from the synthetic scenes at the same time, and we move the objects by using mouse to drag the objects. This is implemented by a single line of code with the *DragControls* class provided in the *threejs* library. *DragControls* is designed for manipulating meshes, which suits our needs exactly since our objects are meshes. We also cast real-time shadow of the objects by applying shadow mapping. This is imple-

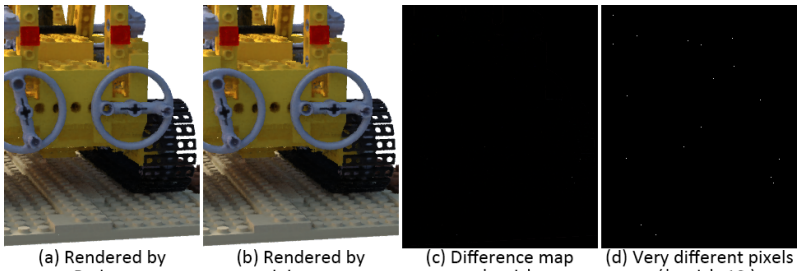

Figure 10. Comparison between images rendered in Python and in a web browser. Image pixel value range is 0-255. Zoom in for details.

mented by having a directional light, an ambient light, and a plane below the objects to receive shadows. The drag control and the real-time shadow are also used in the following examples.

In the second example, we interactively deform the learned chair object to create new variations of chairs. To implement the deformation, we only need to deform the vertex positions of the meshes, and this is achieved by adding vertex deformation code in the vertex shader. Specifically, we implemented three operations: moving the chair up/down will lengthen or shorten its legs, moving the chair left/right will adjust its width, and moving the chair forward/backward will adjust the skew of its back.

In the third example, we render 9 ficus objects, which are considered "NeRF" objects, and a blue ball, which is a classic object with standard material used in classic rendering. We again change the vertex shader to make the leaves of the plants to be repelled by the blue ball.

D. Training

Our training stages are formalized as follows. In the first training stage, we optimize

$$\underset{\mathcal{V},\theta_{\mathcal{A}},\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}} + w_d \mathcal{L}_{\text{dist}} + \mathcal{L}_{\text{v}} \tag{18}$$

and

$$\underset{\mathcal{G}}{\arg\min} \quad \mathcal{L}_{\mathcal{G}}^{\text{bnd}} + w_{g1} \mathcal{L}_{\mathcal{G}}^{\text{sparse}} + w_{g2} \mathcal{L}_{\mathcal{G}}^{\text{smooth}}, \tag{19}$$

where $w_{g1} = w_{g2} = 10^{-5}$. w_d is set to 0.0 for synthetic 360° scenes, 0.01 for forward-facing scenes, and 0.001 for unbounded 360° scenes. In the second training stage, we optimize

$$\underset{\mathcal{V},\theta_{\mathcal{A}},\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}}^{\text{stage2}} + w_d \mathcal{L}_{\text{dist}} + \mathcal{L}_{\text{v}} \tag{20}$$

and Eq. 19. When the loss converges, we fix the weights of $\mathcal{V}$, $\theta_{\mathcal{A}}$, and $\mathcal{G}$ and optimize

$$\underset{\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}}^{\text{bin}}. \tag{21}$$

E. Network architectures

We adopt the MLP designed in NeRF as the network for both $\mathcal{A}$ and $\mathcal{F}$. We increase the hidden layer sizes from 256 to 384, since $\mathcal{A}$ and $\mathcal{F}$ are not used during inference, so we can afford more time on training. The small MLP $\mathcal{H}$ is the same as the small MLP used in SNeRG, with two hidden layers, each consisting 16 neurons.

F. More details about texture images

Since the features to be stored are 8-dimensional, we use two PNG images to store them. Each PNG image has 4 channels, therefore two PNG images have a total of 8 channels to store 8-d features. To avoid having an extra image to store the binary alpha (opacity) channel, we squeeze the alpha channel into the first feature channel, so that the alpha is one when the first feature channel is non-zero, and zero when the channel is zero. Since phones have a hardware constraint that the texture size must be a power of 2 and at most 4096×4096, we split the large texture images into multiple 4096×4096 texture images.

G. Quadrature details

The regular-grid mesh $\mathcal{M}$ provides an efficient way for computing intersections between a ray and the mesh of size $P \times P \times P$ in $O(P)$ complexity, as shown in Figure 5.

First, we compute the set of voxels that are intersected by the ray. This involves solving $3P$ ray-plane intersections and using those intersection points to obtain at most $3P$ intersected voxels. This step is shown in Figure 5a and Eq. 7.

Then, we use the acceleration grid $\mathcal{G} \in \mathbb{R}^{P \times P \times P}$ to prune voxels that are unlikely to contain geometry, with respect to a threshold $\tau_{\mathcal{G}} = 0.1$. This step is shown in Figure 5b and Eq. 8.

Finally, we compute intersections between the ray and the faces of $\mathcal{M}$ that are incident to the voxel's vertex to obtain the final set of quadrature points. This step is shown in Figure 5c and Eq. 9.

During the first quarter of the training iterations, $\mathcal{G}$ may not be accurate, therefore we will keep all $3P$ intersected

voxels regardless of $\tau_{\mathcal{G}}$, and keep $3P$ intersection points (Recall that if the mesh grid is a regular grid, there are at most $3P$ intersections). Then in the next quarter, we will use $\mathcal{G}$ to remove empty voxels and keep at most $3P/2$ non-empty voxels and $3P/2$ intersection points that are closest to the camera. In the rest of the training, we will keep $3P/4$. We also double the training batch size each time we halve the number of intersections.

For the concentric boxes in unbounded 360° scenes, we will compute their intersections and keep all of them.

H. Initial meshes

In this section we detail the polygonal meshes used for synthetic 360°, forward-facing, and unbounded 360° scenes, see Fig. 4 for 2D illustrations.

We will call the coordinate system of a regular mesh grid in a unit cube centered at the origin as the normalized coordinates, and we can apply transformations to obtain the grids in the world coordinates for different types of scenes. In the following, we will denote points in the normalized coordinates as $\mathbf{p} \in [-0.5, 0.5]$ and points in the world coordinates as $\mathbf{p}'$.

For synthetic 360° scenes, we apply scaling to the grid to put the object inside the grid.

$$\mathbf{p}' = w\mathbf{p}, \tag{22}$$

where $w = 2.4$ or 3, depending on the size of the object. We use a grid size of $P = 128$.

In forward-facing scenes, we apply transformation to concentrate more voxels close to the camera, as shown in Fig. 4 (b).

$$\begin{cases} \mathbf{p}'_z &= \exp(w(\mathbf{p}_z + 0.5)), \\ \mathbf{p}'_x &= u\mathbf{p}_x\mathbf{p}'_z, \\ \mathbf{p}'_y &= v\mathbf{p}_y\mathbf{p}'_z, \end{cases} \tag{23}$$

where w is set to a value so that $\mathbf{p}'_z = 25$ when $\mathbf{p}_z = 0.5$; $u = v = 1.75$. We use a grid size of $P = 128$.

In unbounded 360° scenes, we assume the cameras are inside the unit cube in the normalized coordinates, therefore we do not apply transformations. However, to model the surrounding environments, we add a set of $L + 1$ concentric boxes around the regular grid. The boxes have fixed positions and geometry, and their distances to the center are given by

$$d_i = (\exp(\frac{wi}{L}) + w - 1)/2w, \tag{24}$$

where i ranges from 0 to L. w is set to a value so that $d_L = 8$, therefore $d_i \in [0.5, 8]$. We use a grid size of $P = 128$, and $L = 64$.

I. Per-Scene metrics

We provide per-scene breakdown for the quality metrics in Table 8 9 10 12 13 14 16 17 18. We provide per-scene breakdown for rendering speed and storage cost in Table 11 15 19, where OOM (out-of-memory) indicates the device cannot run a testing scene due to GPU memory issues, and ICP (incompatible) indicates the device cannot run the method due to compatibility issues. The GPU memory and disk storage were tested on the Desktop.

For Surface Pro 6, Gaming laptop, and Desktop, we disable frame-rate limiting from vertical synchronization by starting the Chrome browser with the following arguments:

```
--disable-frame-rate-limit
--disable-gpu-vsync
```

However, for phones and Chromebook, we did not find a way to easily disable vertical synchronization, therefore the FPS is capped at 60.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [33]	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65	31.00
JAXNeRF [14]	33.88	25.08	30.51	36.91	33.24	30.03	34.52	29.07	31.65
SNeRG [21]	33.24	24.57	29.32	34.33	33.82	27.21	32.60	27.97	30.38
Ours	34.09	25.02	30.20	35.46	34.18	26.72	32.48	29.06	30.90

Table 8. **PSNR**↑ on **Synthetic** 360° **scenes**.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [33]	0.967	0.925	0.964	0.974	0.961	0.949	0.980	0.856	0.947
JAXNeRF [14]	0.974	0.927	0.967	0.979	0.968	0.952	0.987	0.865	0.952
SNeRG [21]	0.975	0.929	0.967	0.971	0.973	0.938	0.982	0.865	0.950
Ours	0.978	0.927	0.965	0.973	0.975	0.913	0.979	0.867	0.947

Table 9. **SSIM**↑ on **Synthetic** 360° **scenes**.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [33]	0.046	0.091	0.044	0.121	0.050	0.063	0.028	0.206	0.081
JAXNeRF [14]	0.027	0.070	0.033	0.030	0.030	0.048	0.013	0.156	0.051
SNeRG [21]	0.025	0.061	0.028	0.043	0.022	0.052	0.016	0.156	0.050
Ours	0.025	0.077	0.048	0.050	0.025	0.092	0.032	0.145	0.062

Table 10. **LPIPS**↓ on **Synthetic** 360° **scenes**.

	SNeRG [21]								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
iPhone XS	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Pixel 3	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Surface Pro 6	ICP	ICP	ICP	ICP	ICP	ICP	ICP	ICP	-
Chromebook	28.06	OOM	OOM	26.11	27.08	16.48	26.99	11.01	22.62
Gaming laptop	4.94	10.27	OOM	8.10	9.41	2.05	21.65	1.69	8.30
Gaming laptop 🔌	37.66	51.06	OOM	45.52	60.20	13.81	87.67	11.17	43.87
Desktop 🔌	120.70	147.72	81.88	436.05	232.03	92.45	507.54	39.73	207.26
GPU memory	1254.00	4729.00	8243.00	1253.00	1253.00	1253.00	1251.00	2422.00	2707.25
Disk storage	141.00	44.00	43.00	67.00	114.00	134.00	22.00	129.00	86.75

	Ours								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
iPhone XS	60.00	60.00	60.00	60.00	50.10	54.65	60.00	42.37	55.89
Pixel 3	41.68	38.71	43.09	35.59	29.56	32.35	52.65	23.52	37.14
Surface Pro 6	83.40	83.15	99.34	64.01	57.11	58.80	130.62	42.76	77.40
Chromebook	60.00	60.00	60.00	53.24	47.51	51.04	60.00	37.56	53.67
Gaming laptop	186.03	183.04	231.01	156.08	118.27	129.80	332.10	89.74	178.26
Gaming laptop 🔌	657.77	656.22	643.32	649.58	566.39	618.98	648.88	412.70	606.73
Desktop 🔌	810.99	789.30	882.23	707.27	629.70	659.95	970.35	509.48	744.91
GPU memory	451.00	590.00	450.00	456.00	723.00	721.00	322.00	594.00	538.38
Disk storage	107.00	120.00	80.00	88.00	199.00	191.00	50.00	171.00	125.75

Table 11. **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Synthetic** 360° **scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [33]	32.70	25.17	20.92	31.16	20.36	27.40	26.80	27.45	26.50
JAXNeRF [14]	33.30	24.92	21.24	31.78	20.32	28.09	27.43	28.29	26.92
SNeRG [21]	30.04	24.85	20.01	30.91	19.73	27.00	25.80	26.71	25.63
Ours	31.28	24.59	20.54	30.82	19.66	27.05	26.26	27.09	25.91

Table 12. **PSNR**↑ on **Forward-facing scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [33]	0.948	0.792	0.690	0.881	0.641	0.827	0.880	0.828	0.811
JAXNeRF [14]	0.958	0.806	0.717	0.897	0.657	0.850	0.902	0.863	0.831
SNeRG [21]	0.936	0.802	0.696	0.889	0.655	0.835	0.882	0.852	0.818
Ours	0.943	0.808	0.711	0.891	0.647	0.839	0.900	0.864	0.825

Table 13. **SSIM**↑ on **Forward-facing scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [33]	0.178	0.280	0.316	0.171	0.321	0.219	0.249	0.268	0.250
JAXNeRF [14]	0.086	0.207	0.247	0.108	0.266	0.156	0.143	0.173	0.173
SNeRG [21]	0.133	0.198	0.252	0.125	0.255	0.167	0.157	0.176	0.183
Ours	0.143	0.202	0.245	0.115	0.277	0.163	0.147	0.169	0.183

Table 14. **LPIPS**↓ on **Forward-facing scenes**.

	SNeRG [21]								
	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
iPhone XS	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Pixel 3	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Surface Pro 6	ICP	ICP	ICP	ICP	ICP	ICP	ICP	ICP	-
Chromebook	9.75	6.02	OOM	9.68	OOM	5.12	8.68	OOM	7.85
Gaming laptop	7.77	1.28	0.80	8.46	1.14	0.67	4.72	4.18	3.63
Gaming laptop 🔌	52.40	14.45	6.15	54.47	12.43	8.77	32.87	26.51	26.01
Desktop 🔌	110.36	28.18	13.54	122.91	17.59	15.96	62.65	34.46	50.71
GPU memory	3594.00	3585.00	4729.00	3595.00	5903.00	3593.00	3595.00	5903.00	4312.13
Disk storage	149.00	288.00	408.00	162.00	704.00	321.00	251.00	415.00	337.25

	Ours								
	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
iPhone XS	29.82	25.10	OOM	30.02	OOM	26.28	26.30	25.59	27.19
Pixel 3	13.57	12.74	8.66	14.69	10.77	12.98	13.07	12.71	12.40
Surface Pro 6	22.92	20.32	13.84	29.13	17.10	22.30	22.53	23.92	21.51
Chromebook	20.70	18.95	14.65	23.16	16.79	20.06	20.08	21.12	19.44
Gaming laptop	64.27	55.88	37.11	76.29	48.72	60.60	59.65	59.26	57.72
Gaming laptop 🔌	281.01	252.70	170.66	303.77	222.54	260.44	258.45	251.82	250.17
Desktop 🔌	377.87	352.01	254.51	397.00	323.54	367.68	359.68	362.46	349.34
GPU memory	610.00	610.00	1143.00	473.00	1276.00	611.00	604.00	747.00	759.25
Disk storage	127.00	147.00	353.00	89.00	372.00	151.00	162.00	211.00	201.50

Table 15. **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Forward-facing scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [14]	21.76	19.40	23.11	21.73	21.28	21.46
NeRF++ [52]	22.64	20.31	24.32	24.34	22.20	22.76
Ours	21.70	18.86	23.54	23.95	21.72	21.95

Table 16. **PSNR**↑ on **Unbounded** 360° **scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [14]	0.455	0.376	0.546	0.453	0.459	0.458
NeRF++ [52]	0.526	0.453	0.635	0.594	0.530	0.548
Ours	0.426	0.321	0.599	0.556	0.450	0.470

Table 17. **SSIM**↑ on **Unbounded** 360° **scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [14]	0.536	0.529	0.415	0.551	0.546	0.515
NeRF++ [52]	0.455	0.466	0.331	0.416	0.466	0.427
Ours	0.513	0.526	0.358	0.430	0.522	0.470

Table 18. **LPIPS**↓ on **Unbounded** 360° **scenes**.

	Ours					
	Bicycle	Flower	Garden	Stump	Treehill	Mean
iPhone XS	OOM	OOM	22.20	OOM	OOM	22.20
Pixel 3	9.44	8.61	10.49	8.54	9.12	9.24
Surface Pro 6	20.24	19.12	21.67	18.21	17.97	19.44
Chromebook	15.89	14.72	16.56	14.23	15.02	15.28
Gaming laptop	55.62	59.18	58.19	51.73	51.89	55.32
Gaming laptop 🔌	195.63	194.66	204.31	178.89	189.46	192.59
Desktop 🔌	280.24	282.02	295.74	265.90	274.58	279.70
GPU memory	1350.00	1081.00	808.00	1082.00	1490.00	1162.20
Disk storage	400.00	294.00	239.00	337.00	453.00	344.60

Table 19. **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Unbounded** 360° **scenes**.