# Baking Neural Radiance Fields for Real-Time View Synthesis

Peter Hedman Pratul P. Srinivasan Ben Mildenhall Jonathan T. Barron Paul Debevec

Google Research

## Abstract

*Neural volumetric representations such as Neural Radiance Fields (NeRF) have emerged as a compelling technique for learning to represent 3D scenes from images with the goal of rendering photorealistic images of the scene from unobserved viewpoints. However, NeRF's computational requirements are prohibitive for real-time applications: rendering views from a trained NeRF requires querying a multilayer perceptron (MLP) hundreds of times* per ray. *We present a method to train a NeRF, then precompute and store (i.e.F "bake") it as a novel representation called a Sparse Neural Radiance Grid (SNeRG) that enables real-time rendering on commodity hardware. To achieve this, we introduce 1) a reformulation of NeRF's architecture, and 2) a sparse voxel grid representation with learned feature vectors. The resulting scene representation retains NeRF's ability to render fine geometric details and view-dependent appearance, is compact (averaging less than 90 MB per scene), and can be rendered in real-time (higher than 30 frames per second on a laptop GPU). Actual screen captures are shown in our video.*

## 1. Introduction

The task of view synthesis — using observed images to recover a 3D scene representation that can render the scene from novel unobserved viewpoints — has recently seen dramatic progress as a result of using neural volumetric representations. In particular, Neural Radiance Fields (NeRF) [29] are able to render photorealistic novel views with fine geometric details and realistic view-dependent appearance by representing a scene as a continuous volumetric function, parameterized by a multilayer perceptron (MLP) that maps from a continuous 3D position to the volume density and view-dependent emitted radiance at that location. Unfortunately, NeRF's rendering procedure is quite slow: rendering a ray requires querying an MLP hundreds of times, such that rendering a frame at $800 \times 800$ resolution takes roughly a minute on a modern GPU. This prevents NeRF from being used for interactive view synthesis applications such as virtual and augmented reality, or even simply inspecting a recovered 3D model in a web browser.

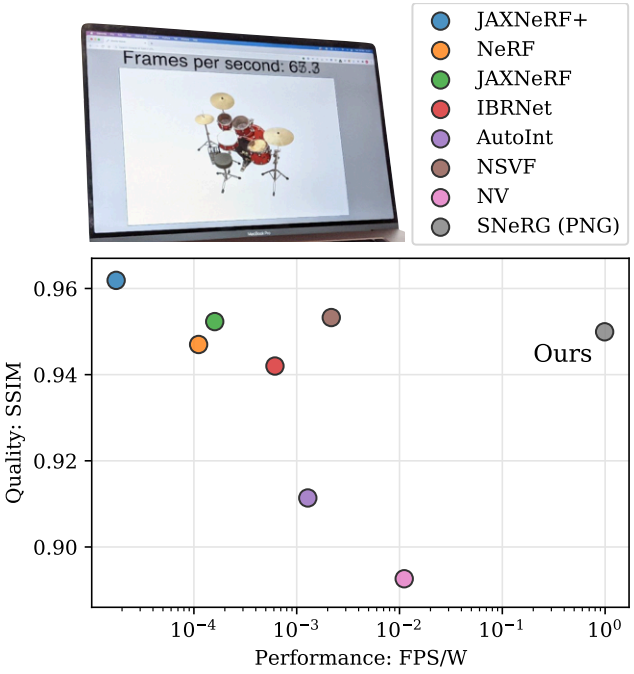



Figure 1: Our method "bakes" NeRF's continuous neural volumetric scene representation into a discrete Sparse Neural Radiance Grid (SNeRG) for real-time rendering on commodity hardware ($\sim$ 65 frames per second on a MacBook Pro in the example shown here and in our supplemental video). Our method is more than two orders of magnitude faster than prior work for accelerating NeRF's rendering procedure and more than an order of magnitude faster than the next-fastest alternative (Neural Volumes) while producing substantially higher-quality renderings.

In this paper, we address the problem of rendering a trained NeRF in real-time, see Figure 1. Our approach accelerates NeRF's rendering procedure by three orders of magnitude, resulting in a rendering time of 22 milliseconds per frame on a single GPU. We precompute and store (i.e. "bake") a trained NeRF into a sparse 3D voxel grid data

structure, which we call a Sparse Neural Radiance Grid (SNeRG). Each active voxel in a SNeRG contains opacity, diffuse color, and a learned feature vector that encodes view-dependent effects. To render this representation, we first accumulate the diffuse colors and feature vectors along each ray. Next, we pass the accumulated feature vector through a lightweight MLP to produce a view-dependent residual that is added to the accumulated diffuse color.

We introduce two key modifications to NeRF that enable it to be effectively baked into this sparse voxel representation: 1) we design a "deferred" NeRF architecture that represents view-dependent effects with an MLP that only runs once per *pixel* (instead of once per 3D sample as in the original NeRF architecture), and 2) we regularize NeRF's predicted opacity field during training to encourage sparsity, which improves both the storage cost and rendering time for the resulting SNeRG.

We demonstrate that our approach is able to increase the rendering speed of NeRF so that frames can be rendered in real-time, while retaining NeRF's ability to represent fine geometric details and convincing view-dependent effects. Furthermore, our representation is compact, and requires less than 90 MB on average to represent a scene.

## 2. Related work

Our work draws upon ideas from computer graphics to enable the real-time rendering of NeRFs. In this section, we review scene representations used for view synthesis with a specific focus on their ability to support real-time rendering, and discuss prior work in efficient representation and rendering of volumetric representations within the field of computer graphics.

**Scene Representations for View Synthesis** The task of view synthesis, using observed images of an object or scene to render photorealistic images from novel unobserved viewpoints, has a rich history within the fields of graphics and computer vision. The majority of prior work in this space has used traditional 3D representations from computer graphics which are naturally amenable to efficient rendering. For scenarios where the scene is captured by densely-sampled images, light field rendering techniques [8, 14, 21] can be used to efficiently render novel views by interpolating between sampled rays. Unfortunately, the sampling and storage requirements of light field interpolation techniques are typically intractable for settings with significant viewpoint motion. Methods that aim to support free-viewpoint rendering from sparsely-sampled images typically reconstruct an explicit 3D representation of the scene [9]. One popular class of view synthesis methods uses mesh-based representations, with either diffuse [27] or view-dependent [5, 9, 46] appearance. Recent methods have trained deep networks to increase the quality of mesh renderings, improving robustness to errors in the reconstructed mesh geometry [15, 42]. Mesh-based approaches are naturally amenable to real-time rendering with highly-optimized rasterization pipelines. However, gradient-based optimization of a rendering loss with a mesh representation is difficult, so these methods have difficulties reconstructing fine structures and detailed scene geometry.

Another popular class of view synthesis methods uses discretized volumetric representations such as voxel grids [25, 36, 37, 39] or multiplane images [12, 33, 40, 48]. While volumetric approaches are better suited to gradient-based optimization, discretized voxel representations are fundamentally limited by their cubic scaling. This restricts their usage to representing scenes at relatively low resolutions in the case of voxel grids, or rendering from a limited range of viewpoints in the case of multiplane images.

NeRF [29] proposes replacing these discretized volumetric representations with an MLP that represents a scene as a *continuous neural volumetric function* by mapping from a 3D coordinate to the volume density and view-dependent emitted radiance at that position. The NeRF representation has been remarkably successful for view synthesis, and follow-on works have extended NeRF for generative modeling [6, 35], dynamic scenes [22, 31], non-rigidly deforming objects [13, 32], and relighting [2, 38]. NeRF is able to represent detailed geometry and realistic appearance extremely efficiently (NeRF uses approximately 5 MB of network weights to represent each scene), but this comes at the cost of slow rendering. NeRF needs to query its MLP hundreds of times per ray, and requires roughly a minute to render a single frame. We specifically address this issue, and present a method that enables a trained NeRF to be rendered in real-time.

Recent works have explored a few strategies for improving the efficiency of NeRF's neural volumetric rendering. AutoInt [23] designs a network architecture that automatically computes integrals along rays, which enables a piecewise ray-marching procedure that requires far fewer MLP evaluations. Neural Sparse Voxel Fields [24] store a 3D voxel grid of latent codes, and sparsifies this grid during training to enable NeRF to skip free space during rendering. Decomposed Radiance Fields [34] represents a scene using a set of smaller MLPs instead of a single large MLP. However, these methods only achieve moderate speedups of around $10\times$ at best, and are therefore not suited for real-time rendering. In contrast to these methods, we specifically focus on accelerating the rendering of a NeRF after it has been trained, which allows us to leverage precomputation strategies that are difficult to incorporate during training.

**Efficient Volume Rendering** Discretized volumetric representations have been used extensively in computer graphics to make rendering more efficient both in terms of storage and rendering speed. Our representation is inspired by this long line of prior work on efficient volume rendering, and we extend these approaches with a deferred neural render-

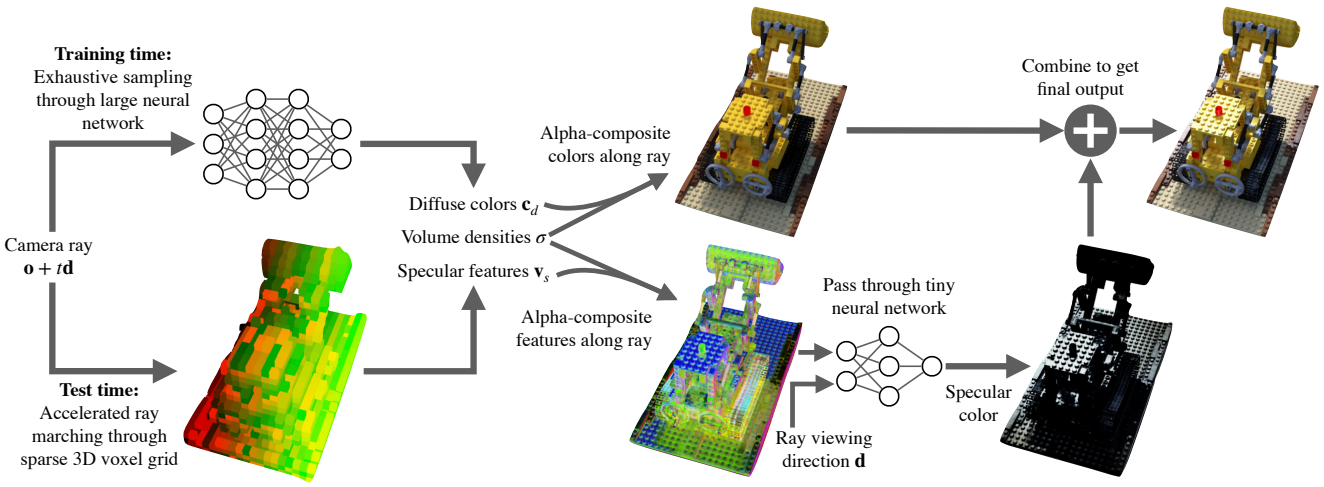



Figure 2: **Rendering pipeline overview.** We jointly design a "deferred" NeRF as well as a procedure to precompute and store the outputs of a trained deferred NeRF in a sparse 3D voxel grid data structure for real-time rendering. At training time, we query the deferred NeRF's MLP for the diffuse color, volume density, and feature-vector at any 3D location along a ray. At test time, we instead precompute and store these values in a sparse 3D voxel grid data structure. Next, we alpha-composite the diffuse colors and feature vectors along the ray. Once the ray has terminated, we use another MLP to predict a view-dependent specular color from the accumulated feature vector, diffuse color, and the ray's viewing direction. Note that this MLP is only run once per pixel.

ing technique to model view-dependent effects.

Early works in volume rendering [17, 20, 45] primarily focused on fast rendering of dense voxel grids. However, as shown by Laine and Karras [18], sparse voxel grids can be an effective and efficient representation for opaque surfaces. In situations where large regions of space share the same data value or a prefiltered representation is required to combat aliasing, hierarchical representations such as sparse voxel octrees [7] are a popular choice of data structure to represent this sparse volumetric content. However, for scenes with detailed geometry and appearance and scenarios where a variable level-of-detail is not required during rendering, octrees' intermediate non-leaf nodes and the tree traversals required to query them can incur a significant memory and time overhead.

Alternatively, sparse voxel grids can be efficiently represented with hash tables [19, 30]. However, hashing each voxel independently can lead to incoherent memory fetches when traversing the representation during rendering. We make a deliberate trade-off to use a *block-sparse* representation, which improves memory coherence but slightly increases the size of our representation.

Our work aims to combine the reconstruction quality and view-dependence of NeRF with the speed of these efficient volume rendering techniques. We achieve this by extending deferred neural rendering [42] to volumetric scene representations. This allows us to visualize trained NeRF models in real-time on commodity hardware, with minimal quality degradation.

## 3. Method Overview

Our overall goal is to design a practical representation that enables the serving and real-time rendering of scenes reconstructed by NeRF. This implies three requirements: 1) Rendering a $800 \times 800$ resolution frame (the resolution used by NeRF) should require less than 30 milliseconds on commodity hardware. 2) The representation should be compressible to 100 MB or less. 3) The uncompressed representation should fit within GPU memory (approximately 4 GB) and should not require streaming.

Rendering a standard NeRF in real-time is completely intractable on current hardware. NeRF requires about 100 teraflops to render a single $800 \times 800$ frame, which results in a best-case rendering time of 10 seconds per frame on an NVIDIA RTX 2080 GPU with full GPU utilization. To enable real-time rendering, we must therefore exchange some of this computation for storage. However, we do not want to precompute and store the entire 5D view-dependent representation [14, 21], as that would require a prohibitive amount of GPU memory.

We propose a hybrid approach that precomputes and stores some content in a sparse 3D data structure but defers the computation of view-dependent effects to rendering time. We jointly design a reformulation of NeRF (Section 4) as well as a procedure to bake this modified NeRF into a discrete volumetric representation that is suited for real-time rendering (Section 5).

## 4. Modifying NeRF for Real-time Rendering

We reformulate NeRF in three ways: 1) we limit the computation of view-dependent effects to a single network evaluation *per ray*, 2) we introduce a small bottleneck in the network architecture that can be efficiently stored as 8 bit integers, and 3) we introduce a sparsity loss during training, which concentrates the opacity field around surfaces in the scene. Here, we first review NeRF's architecture and rendering procedure before describing our modifications.

### 4.1. Review of NeRF

NeRF represents a scene as a continuous volumetric function parameterized by a MLP. Concretely, the 3D position $\mathbf{r}(t)$ and viewing direction $\mathbf{d}$ along a camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, are passed as inputs to an MLP with weights $\Theta$ to produce the volume density $\sigma$ of particles at that location as well as the RGB color $\mathbf{c}$ corresponding to the radiance emitted by particles at the input location along the input viewing direction:

$$\sigma(t), \mathbf{c}(t) = \mathrm{MLP}_\Theta(\mathbf{r}(t), \mathbf{d})\,. \tag{1}$$

A key design decision made in NeRF is to architect the MLP such that volume density is only predicted as a function of 3D position, while emitted radiance is predicted as a function of both 3D position and 2D viewing direction.

To render the color $\hat{\mathbf{C}}(\mathbf{r})$ of a pixel, NeRF queries the MLP at sampled positions $t_k$ along the corresponding ray and uses the estimated volume densities and colors to approximate a volume rendering integral using numerical quadrature, as discussed by Max [26]:

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_k T(t_k)\,\alpha(\sigma(t_k)\delta_k)\,\mathbf{c}(t_k)\,, \tag{2}$$

$$T(t_k) = \exp\left(-\sum_{k'=1}^{k-1} \sigma(t_{k'})\delta_{k'}\right), \quad \alpha(x) = 1 - \exp(-x)\,,$$

where $\delta_k = t_{k+1} - t_k$ is the distance between two adjacent points along the ray.

NeRF trains the MLP by minimizing the squared error between input pixels from a set of observed images (with known camera poses) and the pixel values predicted by rendering the scene as described above:

$$\mathcal{L}_r = \sum_i \left\|\mathbf{C}(\mathbf{r}_i) - \hat{\mathbf{C}}(\mathbf{r}_i)\right\|_2^2 \tag{3}$$

where $\mathbf{C}(\mathbf{r}_i)$ is the color of pixel $i$ in the input images.

By replacing a traditional discrete volumetric representation with an MLP, NeRF makes a strong space-time tradeoff: NeRF's MLP requires multiple orders of magnitude less space than a dense voxel grid, but accessing the properties of the volumetric scene representation at any location requires an MLP evaluation instead of a simple memory lookup. Rendering a single ray that passes through the volume requires hundreds of these MLP queries, resulting in extremely slow rendering times. This tradeoff is beneficial during training; since we do not know where the scene geometry lies during optimization, it is crucial to use a compact representation that can represent highly-detailed geometry at arbitrary locations. However, after a NeRF has been trained, we argue that it is prudent to rethink this space-time tradeoff and bake the NeRF representation into a data structure that stores pre-computed values from the MLP to enable real-time rendering.

### 4.2. Deferred NeRF Architecture

NeRF's MLP can be thought of as predicting a 256-dimensional feature vector for each input 3D location, which is then concatenated with the viewing direction and decoded into an RGB color. NeRF then accumulates these view-dependent colors into a single pixel color. However, evaluating an MLP at every sample along a ray to estimate the view-dependent color is prohibitively expensive for real-time rendering. Instead, we modify NeRF to use a strategy similar to deferred rendering [10, 42]. We restructure NeRF to output a diffuse RGB color $\mathbf{c}_d$ and a 4-dimensional feature vector $\mathbf{v}_s$ (which is constrained to $[0, 1]$ via a sigmoid so that it can be compressed, as discussed in Section 5.4) in addition to the volume density $\sigma$ at each input 3D location:

$$\sigma(t), \mathbf{c}_d(t), \mathbf{v}_s(t) = \mathrm{MLP}_\Theta(\mathbf{r}(t))\,. \tag{4}$$

To render a pixel, we accumulate the diffuse colors and feature vectors along each ray and pass the accumulated feature vector and color, concatenated to the ray's direction, to a very small MLP with parameters $\Phi$ (2 layers with 16 channels each) to produce a view-dependent residual that we add to the accumulated diffuse color:

$$\hat{\mathbf{C}}_d(\mathbf{r}) = \sum_k T(t_k)\,\alpha(\sigma(t_k)\delta_k)\,\mathbf{c}_d(t_k)\,, \tag{5}$$

$$\mathbf{V}_s(\mathbf{r}) = \sum_k T(t_k)\,\alpha(\sigma(t_k)\delta_k)\,\mathbf{v}_s(t_k)\,, \tag{6}$$

$$\hat{\mathbf{C}}(\mathbf{r}) = \hat{\mathbf{C}}_d(\mathbf{r}) + \mathrm{MLP}_\Phi(\mathbf{V}_s(\mathbf{r}), \mathbf{d}) \tag{7}$$

This modification enables us to precompute and store the diffuse colors and 4-dimensional feature vectors within our sparse voxel grid representation discussed below. Critically, we only need to evaluate the $\mathrm{MLP}_\Phi$ to produce view-dependent effects once per *pixel*, instead of once per sample in 3D space as in the standard NeRF model.

### 4.3. Opacity Regularization

Both the rendering time and required storage for a volumetric representation strongly depend on the sparsity of opacity within the scene. To encourage NeRF's opacity field to be sparse, we add a regularizer that penalizes pre-

dicted density using a Cauchy loss during training:

$$\mathcal{L}_s = \lambda_s \sum_{i,k} \log\left(1 + \frac{\sigma(\mathbf{r}_i(t_k))^2}{c}\right), \tag{8}$$

where $i$ indexes pixels in the input (training) images, $k$ indexes samples along the corresponding rays, and hyperparameters $\lambda_s$ and $c$ control the magnitude and scale of the regularizer respectively ($\lambda_s = 10^{-4}$ and $c = 1/2$ in all experiments). To ensure that this loss is not unevenly applied due to NeRF's hierarchical sampling procedure, we only compute it for the "coarse" samples that are distributed with uniform density along each ray.

## 5. Sparse Neural Radiance Grids

We now convert a trained deferred NeRF model, described above, into a representation suitable for real-time rendering. The core idea is to trade computation for storage, significantly reducing the time required to render frames. In other words, we are looking to replace the MLP evaluations in NeRF with fast lookups in a precomputed data structure. We achieve this by precomputing and storing, i.e. *baking*, the diffuse colors $\mathbf{c}_d$, volume densities $\sigma$, and 4-dimensional feature vectors $\mathbf{v}_s$ in a voxel grid data structure.

It is crucial for us to store this volumetric grid using a sparse representation, as a dense voxel grid can easily fill up all available memory on a modern high-end GPU. By exploiting sparsity and only storing voxels that are both occupied and visible, we end up with a much more compact representation.

### 5.1. SNeRG Data Structure

Our Sparse Neural Radiance Grid (SNeRG) data structure represents an $N^3$ voxel grid in a block-sparse format using two smaller dense arrays.

The first array is a 3D texture atlas containing densely-packed "macroblocks" of size $B^3$ each, corresponding to the content (diffuse color, feature vectors, and opacity) that actually exists in the sparse volume. Each voxel in the 3D atlas represents the scene at the full resolution of the dense $N^3$ grid, but the 3D texture atlas is much smaller than $N^3$ since it only contains the sparse "occupied" content. Compared to hashing-based data structures (where $B^3 = 1$), this approach helps keep spatially close content nearby in memory, which is beneficial for efficient rendering.

The second array is a low resolution $(N/B)^3$ indirection grid, which either stores a value indicating that the corresponding $B^3$ macroblock within the full voxel grid is empty, or stores an index that points to the high-resolution content of that macroblock within the 3D texture atlas. This structure crucially lets us skip blocks of empty space during rendering, as we describe below.

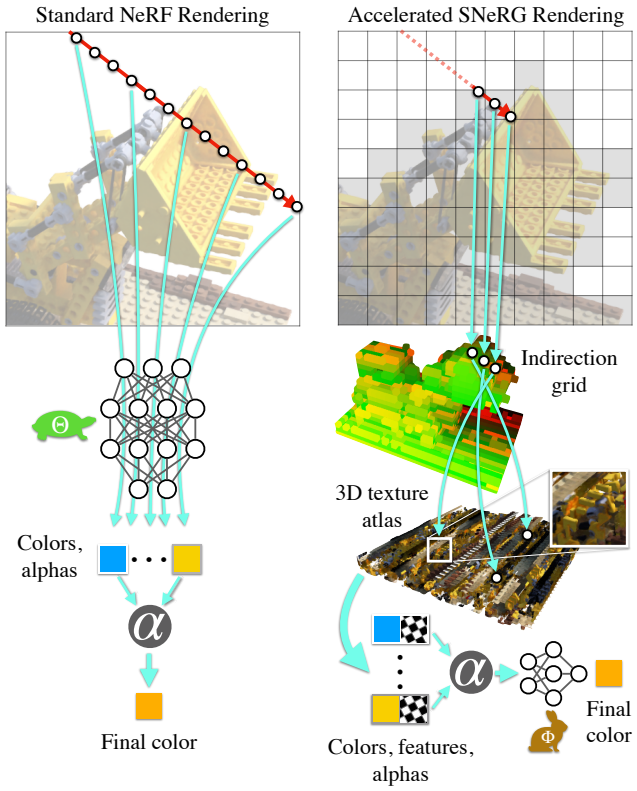
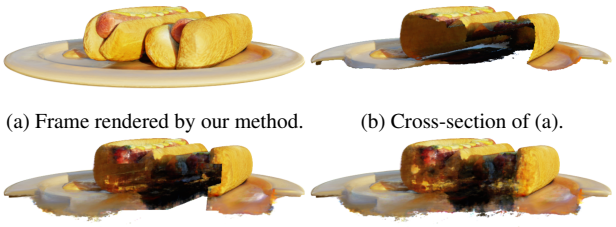



Figure 3: **Comparison of ray-marching procedures for NeRF and SNeRG.** Left: To render a ray with NeRF, we densely sample points along the ray and pass the coordinates to a large MLP to compute colors and opacities, which are alpha-composited into a pixel color. Right: SNeRG significantly accelerates rendering by replacing compute-intensive MLP evaluations with lookups into a precomputed sparse 3D grid representation. We use an indirection grid to map occupied voxel blocks to locations within a compact 3D texture atlas. During ray-marching, we skip unoccupied blocks, and alpha-composite the diffuse colors and feature vectors fetched along the ray. We terminate ray-marching once the accumulated opacity saturates, and pass the accumulated color and features to a small MLP that evaluates the view-dependent component for the ray.

### 5.2. Rendering

We render a SNeRG using a ray-marching procedure, as done in NeRF. The critical differences that enable real-time rendering are: 1) we precompute the diffuse colors and feature vectors at each 3D location, allowing us to look them up within our data structure instead of evaluating an MLP, and 2) we only evaluate an MLP to produce view-dependent effects once per pixel, as opposed to once per 3D location.

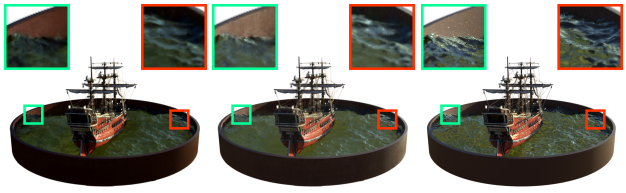
To estimate the color of each ray, we first march the ray through the indirection grid, skipping macroblocks that are marked as empty. For macroblocks that are occupied, we



(a) Frame rendered by our method. (b) Cross-section of (a).

(c) Trained without $\mathcal{L}_s$. (d) No $\mathcal{L}_s$, no visibility culling.

Figure 4: **Visualization of sparsity loss and visibility culling.** We render cross-sections of the *Hotdog* scene to inspect the effect of our sparsity loss $\mathcal{L}_s$ and visibility culling. Our full method (b) represents the scene by only allocating content around visible scene surfaces. Removing either the sparsity loss alone (c) or both the sparsity loss and visibility culling (d) results in a much less compact representation.



(a) No FT (23.30) (b) FT (28.35) (c) Ground Truth

Figure 5: **Impact of fine-tuning (FT) the view-dependent appearance network** (PSNR in parentheses). (a) Renderings from a SNeRG representation can be lower-quality than those from the deferred NeRF, primarily due the quantization from 32-bit floating point to 8-bit integer values (see Sections 5.3, 5.4). (b) We are able to regain most of the lost quality by fine-tuning the weights of the deferred shading network $\text{MLP}_\Phi$ (Section 5.5).

step at the voxel width $1/N$ through the corresponding block in the 3D texture atlas, and use trilinear interpolation to fetch values at each sample location. We further accelerate rendering and conserve memory bandwidth by only fetching features where the volume density is non-zero. We use standard alpha compositing to accumulate the diffuse color and features, terminating ray-marching once the opacity has saturated. Finally, we compute the view-dependent specular color for the ray by evaluating $\text{MLP}_\Phi$ with the accumulated color, feature vector and the ray's viewing direction. We then add the resulting residual color to the accumulated diffuse color, as described in Equation 7.

### 5.3. Baking

To minimize storage cost and rendering time, our baking procedure aims to only allocate storage for voxels in the scene that are both non-empty and visible in at least one of the training views. We start by densely evaluating the NeRF network for the full $N^3$ voxel grid. We convert NeRF's unbounded volume density values, $\sigma$, to traditional opacity values $\alpha = 1 - \exp(\sigma v)$, where $v = 1/N$ is the width of a voxel. Next, we sparsify this voxel grid by culling empty space, i.e. macroblocks where the maximum opacity is low (below $\tau_\alpha$), and culling macroblocks for which the voxel visibilities are low (maximum transmittance $T$ between the voxel and all training views is below $\tau_{vis}$). In all experiments, we set $\tau_\alpha = 0.005$ and $\tau_{vis} = 0.01$. Finally, we compute an anti-aliased estimate for the content in the remaining macroblocks by densely evaluating the trained NeRF at 16 Gaussian distributed locations within each voxel ($\sigma = v/\sqrt{12}$) and averaging the resulting diffuse colors, feature vectors, and volume densities.

### 5.4. Compression

We quantize all values in the baked SNeRG representation to 8 bits and separately compress the indirection grid and the 3D texture atlas. We compress each slice of the indirection grid as a lossless PNG, and we compress the 3D texture atlas as either a set of lossless PNGs, a set of JPEGs, or as a single video encoded with H264. The quality versus storage tradeoff of this choice is evaluated in Table 3. For synthetic scenes, compressing the texture atlas results in approximately $80\times$, $100\times$, and $230\times$ compression rates for PNG, JPEG, and H264, respectively. We specifically choose a macroblock size of $32^3$ voxels to align the 3D texture atlas macroblocks with the blocks used in image compression. This reduces the size of the compressed 3D texture atlas because additional coefficients are not needed to represent discontinuities between macroblocks.

### 5.5. Fine-tuning

While the compression and quantization procedure described above is crucial for making SNeRG compact and easy to distribute, the quality of images rendered from the baked SNeRG is lower than the quality of images rendered from the corresponding deferred NeRF. Figure 5 visualizes how quantization affects view-dependent effects by biasing renderings towards a darker, diffuse-only color $\hat{\mathbf{C}}_d(\mathbf{r})$.

Fortunately, we are able to recoup almost all of that lost accuracy by fine-tuning the weights of the deferred per-pixel shading $\text{MLP}_\Phi$ to improve the final rendering quality (Table 2). We optimize the parameters $\Phi$ to minimize the squared error between the observed input images used to train the deferred NeRF and the images rendered from our SNeRG. We use the Adam optimizer [16] with a learning rate of $3 \times 10^{-4}$ and optimize for 100 epochs.

## 6. Implementation Details

Our deferred NeRF model is based on JAXNeRF [11], an implementation of NeRF in JAX [3]. As in NeRF, we apply a positional encoding [41] to positions and view di-

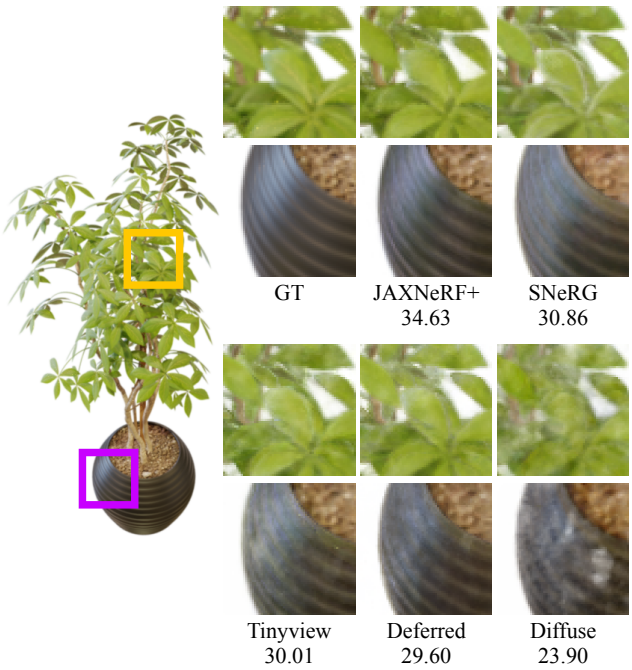
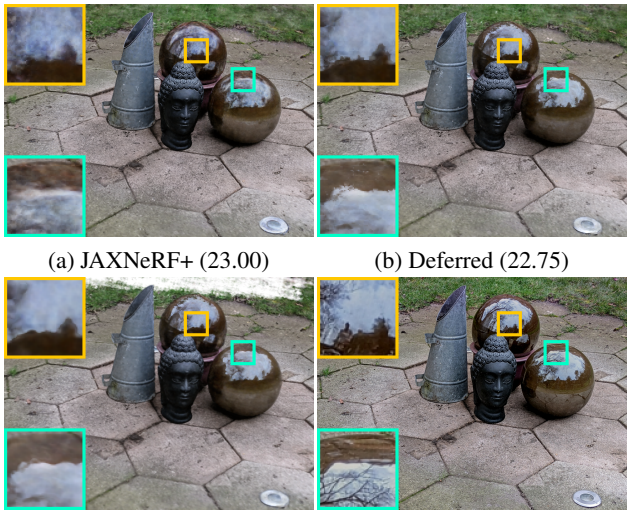



Figure 6: **Ablation study** showing visual examples from Table 2. Note the minimal difference in quality between the full JAXNeRF+ network and the successive approximations we make to speed up view-dependence (*Tinyview*, *Deferred NeRF* and *SNeRG*). For completeness, we show the floating alpha artifacts introduced by not modelling view-dependence at all (*Diffuse*).

rections. We train all networks for 250k iterations with a learning rate which decays log-linearly from $2 \times 10^{-3}$ to $2 \times 10^{-5}$. To improve stability we use JAXNeRF's "warm up" functionality to reduce the learning rate to $2 \times 10^{-4}$ for the first 2500 iterations, and we clip gradients by value (at $0.01$) and then by norm (also at $0.01$). We use a batch size of 8,192 for synthetic scenes and a batch size of 16,384 for real scenes.

As our rendering time is independent of $\mathrm{MLP}_\Theta$'s model size, we can afford to use a larger network for our experiments. To this end, we base our method on the JAXNeRF+ model, which was trained with 576 samples per ray (192 coarse, 384 fine) and uses 512 channels per layer in $\mathrm{MLP}_\Theta$.

During baking, we set the voxel grid resolution to be slightly larger than the size of the training images: $1000^3$ for the synthetic scenes, and $1300^3$ for the real datasets. We implement the SNeRG renderer in Javascript and WebGL using the THREE.js library. We load the indirection grid and 3D texture atlas into 8 bit 3D textures. The view-dependence $\mathrm{MLP}_\Phi$ is stored uncompressed and is implemented in a WebGL shader. In all experiments, we run this renderer on a 2019 MacBook Pro laptop equipped with a 85W AMD Radeon Pro 5500M GPU.



(a) JAXNeRF+ (23.00) (b) Deferred (22.75)

(c) SNeRG (21.45) (d) Ground Truth

Figure 7: **Real 360° scene** (PSNR in parentheses). Note how our real-time method is able to model the mirror-like reflective surface of the garden spheres.

## 7. Experiments

We validate our design decisions with an extensive set of ablation studies and comparisons to recent techniques for accelerating NeRF. Our experiments primarily focus on free-viewpoint rendering of 360° scenes (scenes captured by inwards-facing cameras on the upper hemisphere). Though acceleration techniques already exist for the special case in which all cameras face the same direction (see Broxton *et al*. [4]), 360° scenes represent a challenging and general use-case that has not yet been addressed. In Figure 7, we show an example of a real 360° scene and present more results in our supplement, including the forward-facing scenes from Local Light Field Fusion (LLFF) [28].

We evaluate all ablations and baseline methods according to three criteria: render-time performance (measured by frames per second as well as GPU memory consumption in gigabytes), storage cost (measured by megabytes required to store the compressed representation), and rendering quality (measured using the PSNR, SSIM [44], and LPIPS [47] image quality metrics). It is important to explicitly account for power consumption when evaluating performance — algorithms that are fast on a high performance GPU are not necessarily fast on a laptop. We therefore adopt the convention used by the high performance graphics community of measuring performance relative to power consumption, i.e. FPS per watt, or equivalently, frames per joule [1].

Please refer to our video for screen captures of our technique being used for real-time rendering on a laptop.

| | MLP | $\mathcal{L}_s$ | Defer | ms/frame ↓ | GPU GB ↓ |
|---|---|---|---|---|---|
| Ours | ✓ | ✓ | ✓ | 11.9± 4.5 | 1.73 ± 1.48 |
| 1) | | ✓ | | 9.2± 4.6 | 1.73 ± 1.48 |
| 2) | ✓ | | ✓ | 20.0*± 5.3 | 4.26 ± 1.56 |
| 3) | ✓ | ✓ | | 343.6± 247.5 | 1.73 ± 1.48 |

Table 1: Performance ablation study, including uncompressed GPU memory used during rendering, on the Synthetic 360° scenes. Ablation 2 ran out of memory on the *Ficus* scene, which biases the average runtime for that row.

### 7.1. Ablation Studies

In Table 1, we ablate combinations of three components of our method that primarily affect speed and GPU memory usage. Ablation 1 shows that removing the view-dependence MLP has a minimal effect on runtime performance. Ablation 2 shows that removing the sparsity loss $\mathcal{L}_s$ greatly increases (uncompressed) memory usage. Ablation 3 shows that switching from our "deferred" rendering back to NeRF's approach of querying an MLP at each sample along the ray results in prohibitively large render times.

Table 2 and Figure 6 show the impact on rendering quality of each of our design decisions in building a representation suitable for real-time rendering. Although our simplifications of using a deferred rendering scheme ("Deferred") and a smaller network architecture ("Tinyview") for view-dependent appearance do slightly reduce rendering quality, they are crucial for enabling real-time rendering, as discussed above. Note that the initial impact on quality from quantizing and compressing our representation is significant. However, after fine tuning ("FT"), the final rendering quality of our SNeRG model remains competitive with the neural model from which it was derived ("Deferred").

In Table 3 we explore the impact of various compression schemes on disk storage space requirements. Our sparse voxel grid benefits greatly from applying compression techniques such as JPEG or H264 to its 3D texture atlas, achieving a file size over 200× more compact than a naive 32 bit float array while sacrificing less than 1dB of PSNR. Because our sparsity loss $\mathcal{L}_s$ concentrates opaque voxels around surfaces (see Figure 4), ablating it significantly increases model size. Our compressed SNeRG representations are small enough to be quickly loaded in a web page.

The positive impact of training our models using the sparsity loss $\mathcal{L}_s$ is visible across these ablations — it more than doubles rendering speed, halves the storage requirements of both the compressed representation on disk and the uncompressed representation in GPU memory, and minimally impacts rendering quality.

### 7.2. Baseline Comparisons

As shown in Table 4, the quality of our method is comparable to all other methods, while our run-time performance

| | PSNR ↑ | SSIM ↑ | LPIPS ↓ |
|---|---|---|---|
| JAXNeRF+ | 33.00 | 0.962 | 0.038 |
| JAXNeRF+ Tinyview | 31.65 | 0.954 | 0.047 |
| JAXNeRF+ Deferred | 30.55 | 0.952 | 0.049 |
| SNeRG (PNG) | 30.38 | 0.950 | 0.050 |
| SNeRG (PNG, no $\mathcal{L}_s$) | 30.22 | 0.949 | 0.050 |
| SNeRG (PNG, no FT) | 26.68 | 0.930 | 0.053 |
| JAXNeRF+ Diffuse | 27.39 | 0.927 | 0.068 |

Table 2: Quality ablation study on Synthetic 360° scenes.

| | PSNR ↑ | SSIM ↑ | LPIPS ↓ | MB ↓ |
|---|---|---|---|---|
| SNeRG (Float) | 30.47 | 0.951 | 0.049 | 6883.6 |
| SNeRG (PNG, no $\mathcal{L}_s$) | 30.22 | 0.949 | 0.050 | 176.0 |
| SNeRG (PNG) | 30.38 | 0.950 | 0.050 | 86.7 |
| SNeRG (JPEG) | 29.71 | 0.939 | 0.062 | 70.9 |
| SNeRG (H264) | 29.86 | 0.938 | 0.065 | 30.2 |
| JAXNeRF+ | 33.00 | 0.962 | 0.038 | 18.0 |
| JAXNeRF | 31.65 | 0.952 | 0.051 | 4.8 |

Table 3: Storage ablation study on Synthetic 360° scenes.

| | PSNR ↑ | SSIM ↑ | LPIPS ↓ | W ↓ | FPS ↑ | FPS/W ↑ |
|---|---|---|---|---|---|---|
| JAXNeRF+ [11] | 33.00 | 0.962 | 0.038 | 300 | 0.01 | 0.00002 |
| NeRF [29] | 31.00 | 0.947 | 0.081 | 300 | 0.03 | 0.00011 |
| JAXNeRF [11] | 31.65 | 0.952 | 0.051 | 300 | 0.05 | 0.00016 |
| IBRNet [43] | 28.14 | 0.942 | 0.072 | 300 | 0.18 | 0.00061 |
| AutoInt [23] | 25.55 | 0.911 | 0.170 | 300 | 0.38 | 0.00128 |
| NSVF [24] | 31.74 | 0.953 | 0.047 | 300 | 0.65 | 0.00217 |
| NV [25] | 26.05 | 0.893 | 0.160 | 300 | 3.33 | 0.01111 |
| SNeRG (PNG) | 30.38 | 0.950 | 0.050 | 85 | 84.06 | 0.98897 |

Table 4: Baseline comparisons on Synthetic 360° scenes.

is an order of magnitude faster than the fastest competing approach (Neural Volumes [25]) and more than a thousand times faster than the slowest (NeRF). Note that we measure the run-time rendering performance of our method on a laptop with an 85W mobile GPU, while all other methods are run on servers or workstations equipped with much more powerful GPUs (over 3× the power draw).

## 8. Conclusion

We have presented a technique for rendering Neural Radiance Fields in real-time by precomputing and storing a Sparse Neural Radiance Grid. This *SNeRG* uses a sparse voxel grid representation to store the precomputed scene geometry, but keeps storage requirements reasonable by maintaining a neural representation for view-dependent appearance. Rendering is accelerated by evaluating the view-dependent shading network only on the visible parts of the scene, achieving over 30 frames per second on a laptop GPU for typical NeRF scenes. We hope this ability to render neural volumetric representations such as NeRF in real time on commodity graphics hardware will help increase the adoption of these neural scene representations in vision and graphics applications.

**Acknowledgements** We thank Keunhong Park and Michael Broxton for their generous help with debugging, and Ryan Overbeck for last-minute JavaScript help. Many thanks to John Flynn, Dominik Kaeser, Keunhong Park, Ricardo Martin-Brualla, Hugues Hoppe, Janne Kontkanen, Utkarsh Sinha, Per Karlsson, and Mark Matthews for fruitful discussions, brainstorming, and testing out our viewer.

## References

[1] Tomas Akenine-Möller and Björn Johnsson. Performance per what? *Journal of Computer Graphics Techniques*, 2012. 7

[2] Sai Bi, Zexiang Xu, Pratul P. Srinivasan, Ben Mildenhall, Kalyan Sunkavalli, Miloš Hašan, Yannick Hold-Geoffroy, David Kriegman, and Ravi Ramamoorthi. Neural reflectance fields for appearance acquisition. *arXiv cs.CV arXiv:2008.03824*, 2020. 2

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. http://github.com/google/jax. 6

[4] 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. 7

[5] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. *SIGGRAPH*, 2001. 2

[6] Eric R. Chan, Marco Monteiro, Petr Kellnhofer, Jiajun Wu, and Gordon Wetzstein. pi-GAN: Periodic implicit generative adversarial networks for 3D-aware image synthesis. *CVPR*, 2021. 2

[7] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. *Symposium on Interactive 3D Graphics and Games*, 2009. 3

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

[9] Paul Debevec, C. J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. *SIGGRAPH*, 1992. 2

[10] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *SIGGRAPH*, 1988. 4

[11] Boyang Deng, Jonathan T. Barron, and Pratul P. Srinivasan. JaxNeRF: an efficient JAX implementation of NeRF, 2020. http://github.com/google-research/google-research/tree/master/jaxnerf. 6, 8, 12

[12] 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, 12

[13] Guy Gafni, Justus Thies, Michael Zollhöfer, and Matthias Nießner. Dynamic neural radiance fields for monocular 4D facial avatar reconstruction. *CVPR*, 2021. 2

[14] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *SIGGRAPH*, 1996. 2, 3

[15] 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

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015. 6

[17] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH*, 1994. 3

[18] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *I3D*, 2010. 3

[19] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 2006. 3

[20] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 1980. 3

[21] Marc Levoy and Pat Hanrahan. Light field rendering. *SIGGRAPH*, 1996. 2, 3

[22] Zhengqi Li, Simon Niklaus, Noah Snavely, and Oliver Wang. Neural scene flow fields for space-time view synthesis of dynamic scenes. *CVPR*, 2021. 2

[23] David B. Lindell, Julien N.P. Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural rendering. *CVPR*, 2021. 2, 8, 12

[24] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *NeurIPS*, 2020. 2, 8, 12

[25] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *SIGGRAPH*, 2019. 2, 8, 12

[26] Nelson Max. Optical models for direct volume rendering. *IEEE TVCG*, 1995. 4

[27] Michael Goesele Michael Waechter, Nils Moehrle. Let there be color! Large-scale texturing of 3D reconstructions. *ECCV*, 2014. 2

[28] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima K. Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics*, 2019. 7, 12

[29] 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. *ECCV*, 2020. 1, 2, 8, 12

[30] Matthias Nießner, Michael Zollhofer, Shahram Izadi, and Marc Stamminger. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics*, 2013. 3

[31] Julian Ost, Fahim Mannan, Nils Thuerey, Julian Knodt, and Felix Heide. Neural scene graphs for dynamic scenes. *CVPR*, 2021. 2

[32] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. Deformable neural radiance fields. *arXiv cs.CV 2011.12948*, 2020. 2

[33] Eric Penner and Li Zhang. Soft 3D reconstruction for view synthesis. *ACM Transactions on Graphics*, 2017. 2, 12

[34] Daniel Rebain, Wei Jiang, Soroosh Yazdani, Ke Li, Kwang Moo Yi, and Andrea Tagliasacchi. DeRF: Decomposed radiance fields. *CVPR*, 2021. 2, 12

[35] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. GRAF: Generative radiance fields for 3D-aware image synthesis. *NeurIPS*, 2020. 2

[36] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. *IJCV*, 1999. 2

[37] Vincent Sitzmann, Michael Zollhoefer, and Gordon Wetzstein. Scene representation networks: Continuous 3D-structure-aware neural scene representations. *NeurIPS*, 2019. 2

[38] Pratul P. Srinivasan, Boyang Deng, Xiuming Zhang, Matthew Tancik, Ben Mildenhall, and Jonathan T. Barron. NeRV: Neural reflectance and visibility fields for relighting and view synthesis. *CVPR*, 2021. 2

[39] Pratul P. Srinivasan, Ben Mildenhall, Matthew Tancik, Jonathan T. Barron, Richard Tucker, and Noah Snavely. Lighthouse: Predicting lighting volumes for spatially-coherent illumination. *CVPR*, 2020. 2

[40] Pratul P. Srinivasan, Richard Tucker, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, and Noah Snavely. Pushing the boundaries of view extrapolation with multiplane images. *CVPR*, 2019. 2, 12

[41] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *NeurIPS*, 2020. 6

[42] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics*, 2019. 2, 3, 4

[43] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P. Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas Funkhouser. IBRNet: Learning multi-view image-based rendering. *CVPR*, 2021. 8, 12

[44] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE TIP*, 2004. 7

[45] Lee A Westover. Splatting: A parallel, feed-forward volume rendering algorithm. Technical report, University of North Carolina at Chapel Hill, USA, 1991. 3

[46] Daniel Wood, Daniel Azuma, Wyvern Aldinger, Brian Curless, Tom Duchamp, David Salesin, and Werner Stuetzle. Surface light fields for 3D photography. *SIGGRAPH*, 2000. 2

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

[48] 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, 12

## A. WebGL Implementation Details

Our web renderer is implemented in WebGL using the THREE.js library. To conserve memory bandwidth, we load the 3D texture atlas as three separate 8-bit 3D textures: one for alpha, one for RGB and one for features. We load the indirection grid as a low resolution 8-bit 3D texture.

During ray marching, we first query the intersection grid at the current location along the ray. If this value indicates that the macroblock is empty, we use a ray-box intersection test to skip ahead to the next macroblock along the ray.

For non-empty macroblocks, we first query the alpha texture using nearest neighbor interpolation. If alpha is zero, the current voxel within the macroblock contains empty space, and we do not fetch any additional information. If alpha is non-zero, we use trilinear interpolation to fetch the high-resolution alpha, colors and features at that voxel. This reduces the bandwidth requirement from 64 bytes per sample to 1 byte per sample for rays that are traversing empty space inside each occupied macroblock.

We implement the view-dependence MLP as simple nested for-loops in a GLSL shader. We load the network weights as 32-bit floating point textures and hard-code the network biases directly into the shader. Interestingly, we found that reducing precision lower than 32 bits did not improve the rendering performance noticeably. For added efficiency, we only evaluate the view-dependence MLP for pixels that have non-zero accumulated alpha.

## B. Performance Measurement

We measure performance using the Chrome browser running on a 2019 MacBook Pro Laptop equipped with an 85 watt AMD Radeon Pro 5500M GPU (8GB of GPU RAM).

For accurate performance measurements, we make sure the laptop is connected to the charger, close all other applications on the laptop, and restart our browser to disable frame-rate limiting from vertical synchronization:

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

In our results, we report the average frame time for rendering a 150-frame camera animation orbiting the scene (or rotating within the camera plane for forward-facing scenes). Our test-time renderings use the same image resolutions and camera intrinsics as the input training images:

- $39^\circ$ field-of-view at $800 \times 800$ for Synthetic $360^\circ$ scenes,
- $53^\circ$ field-of-view at $1006 \times 756$ for Real Forward-Facing, and
- $53^\circ$ at field-of-view $990 \times 773$ for Real $360^\circ$ scenes.

| | PSNR ↑ | SSIM ↑ | LPIPS ↓ | FPS ↑ | MB ↓ |
|---|---|---|---|---|---|
| $1000^3$ | 30.38 | 0.950 | 0.050 | 84.06 | 86.7 |
| $750^3$ | 29.94 | 0.947 | 0.054 | 89.82 | 44.7 |
| $500^3$ | 28.93 | 0.939 | 0.064 | 101.78 | 17.8 |

Table 5: Voxel grid resolution ablation using SNeRG (PNG) on Synthetic 360° Scenes.

# C. Additional Experiment Details

## C.1. Experiments with Changing 3D Resolution

Table 5 demonstrates that our method is able to achieve even higher rendering speeds and lower storage costs by baking the 3D grids at a lower resolution, at the expense of a slight decrease in rendering quality.

## C.2. Real 360° Scenes

We evaluate our method on the two real 360° scenes provided by the original NeRF paper (*Flowers* and *Pine Cone*) and two new scenes that we have captured ourselves (*Toy Car* and *Spheres*). All four datasets contain 100-200 images where the camera orbits around an object. Note that the *Spheres* scene contains glossy objects that are hard to model using diffuse geometry alone.

Tables 6, 7, and 8 demonstrate that our method is able to maintain rendering quality close to the trained NeRF models while rendering about 30 frames per second (Table 9). Table 12 studies the impact of using different image and video compression algorithms for these datasets, and shows that we are able to store these scenes using about 50 MB.

We train all NeRF models on this data by shifting and scaling the camera translations so that they approximately lie on a sphere around the origin, and sampling points linearly in disparity along each camera ray, as done by Mildenhall *et al*. After training, we manually set a bounding box to isolate the objects of interest in the scene and ignore the unbounded peripheral content that is not sampled well enough for NeRF to recover. During baking, we only evaluate the subset of the scene which is inside this bounding box. We change our quality measurements to reflect this, masking all of the images (our results, baseline results, and ground truth images) using the alpha mask generated by our method. Otherwise, the results would be significantly biased by the missing background geometry that was outside the scene bounding box. Interestingly, we find that a diffuse-only model without any view-dependent effects is surprisingly competitive for these scenes, potentially due to the low-frequency lighting conditions during capture. Additionally, the diffuse model is able to reasonably fake view-dependent effects in some cases by hiding mirrored versions of reflected content inside the objects' surfaces.

| | Mean | *Spheres* | *Toy Car* | *Flowers* | *Pine Cone* |
|---|---|---|---|---|---|
| JAXNeRF+ | 24.56 | 23.56 | 26.13 | 25.22 | 23.33 |
| JAXNeRF+ Deferred | 24.31 | 23.44 | 26.16 | 24.81 | 22.81 |
| SNeRG (PNG) | 23.97 | 23.16 | 26.17 | 24.43 | 22.14 |
| JAXNeRF+ Diffuse | 24.02 | 23.26 | 26.17 | 24.23 | 22.42 |

Table 6: PSNR ↑, Real 360° scenes.

| | Mean | *Spheres* | *Toy Car* | *Flowers* | *Pine Cone* |
|---|---|---|---|---|---|
| JAXNeRF+ | 0.703 | 0.573 | 0.792 | 0.765 | 0.681 |
| JAXNeRF+ Deferred | 0.693 | 0.563 | 0.787 | 0.754 | 0.666 |
| SNeRG (PNG) | 0.662 | 0.521 | 0.788 | 0.746 | 0.595 |
| JAXNeRF+ Diffuse | 0.681 | 0.565 | 0.788 | 0.728 | 0.645 |

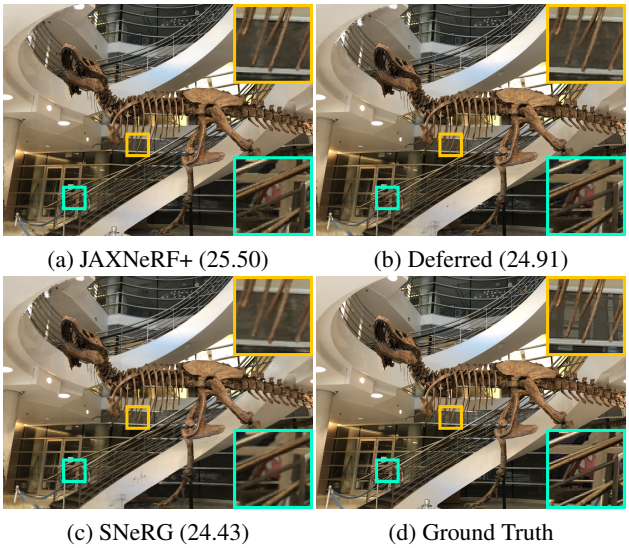
Table 7: SSIM ↑, Real 360° scenes.

| | Mean | *Spheres* | *Toy Car* | *Flowers* | *Pine Cone* |
|---|---|---|---|---|---|
| JAXNeRF+ | 0.248 | 0.311 | 0.195 | 0.227 | 0.257 |
| JAXNeRF+ Deferred | 0.261 | 0.320 | 0.209 | 0.244 | 0.271 |
| SNeRG (PNG) | 0.293 | 0.357 | 0.209 | 0.272 | 0.336 |
| JAXNeRF+ Diffuse | 0.260 | 0.306 | 0.200 | 0.257 | 0.277 |

Table 8: LPIPS ↓, Real 360° scenes.

| *Spheres* | *Toy Car* | *Flowers* | *Pine Cone* |
|---|---|---|---|
| 53.7 | 41.1 | 40.8 | 39.5 |

Table 9: Performance (FPS ↑), Real 360° Scenes.



(a) JAXNeRF+ (25.50) (b) Deferred (24.91)

(c) SNeRG (24.43) (d) Ground Truth

Figure 8: **Real forward-facing scene example results** (PSNR in parentheses).

| | PSNR ↑ | SSIM ↑ | LPIPS ↓ | W ↓ | FPS ↑ | FPS/W ↑ |
|---|---|---|---|---|---|---|
| JAXNeRF+ | 26.95 | 0.845 | 0.145 | 300 | 0.00 | 0.00001 |
| DeRF | 24.81 | 0.767 | 0.274 | 300 | 0.03 | 0.00009 |
| NeRF | 26.50 | 0.811 | 0.250 | 300 | 0.03 | 0.00011 |
| JAXNeRF | 26.92 | 0.831 | 0.173 | 300 | 0.04 | 0.00013 |
| IBRNet | 26.73 | 0.851 | 0.175 | 300 | 0.18 | 0.00061 |
| LLFF | 24.13 | 0.798 | 0.212 | 250 | 60.00 | 0.24000 |
| SNeRG (PNG) | 25.63 | 0.818 | 0.183 | 85 | 27.38 | 0.32210 |

Table 10: Quality and performance comparison for Real Forward-Facing scenes.

| *T-Rex* | *Leaves* | *Room* | *Orchids* | *Horns* | *Fortress* | *Fern* | *Flower* |
|---|---|---|---|---|---|---|---|
| 34.78 | 19.02 | 37.09 | 18.54 | 34.02 | 39.16 | 26.73 | 26.58 |

Table 11: Performance (FPS ↑), Real Forward-Facing Scenes.

### C.3. Real Forward-Facing Scenes

We also evaluate our approach on the real forward-facing scenes in the NeRF paper (Tables 10, 11, and 12). Since these scenes are only captured and viewed from a limited range of forward-facing viewpoints, layered representations such as multi-plane images [12, 28, 33, 40, 48] are a compelling option for real-time rendering. Note that the normalized device coordinate transformation used in NeRF for these forward-facing scenes can be interpreted as transforming NeRF into a continuous version of a multiplane image representation that supports larger viewpoint changes.

We found that our baking procedure sometimes reduces the total alpha mass in the scene, introducing small semi-transparent holes for these datasets. To overcome this, we partially un-premultiply alpha after ray marching. That is, after alpha compositing:

$$\text{rgbafeatures} \leftarrow \text{rgbafeatures} \times \frac{\min(1.0, 1.5\alpha)}{\alpha} \quad (9)$$

if $\alpha > 0$. This fully saturates alpha values above 0.66, while still allowing for soft edges and a smooth fall-off.

### C.4. Baselines

Here we provide additional details for the baseline methods we use in our experiments.

**NeRF [29]** We directly use the results reported in the original paper by Mildenhall *et al*. Run-time was measured on a single NVIDIA V100 GPU.

**JAXNeRF [11]** is a JAX implementation of NeRF, with default settings (64 + 128 samples per ray, MLP width of 256). Run-time was measured on an NVIDIA V100 GPU.

**JAXNeRF+** is a more compute-intensive version of JAXNeRF, trained with 192 + 384 samples per ray and an MLP width of 512 channels. Run-time was measured on a single NVIDIA V100 GPU. We use this architecture as a starting point for our modifications (deferred shading and baking), as using more samples per ray allows us to recover a sparser representation that better concentrates opacity near object surfaces.

**JAXNeRF+ Tinyview** This baseline measures the effects of using a smaller network architecture (same as $\text{MLP}_\Phi$) for the view-dependent effects. It uses the same architecture for the view-dependent effects as our "Deferred" model, but evaluates view-dependent effects for every 3D sample instead of once per pixel.

**JAXNeRF+ Diffuse** This baseline measures the effects of modeling view-dependent appearance. It uses the same architecture as JAXNeRF+, but replaces the view dependence network with a single layer that directly outputs a color without any knowledge of the viewing direction.

**AutoInt [23]** We use the N=8 setting reported by Lindell *et al*., which achieves their highest ratio of quality to run-time. The authors did not mention what hardware they ran on, but we are assuming that they also run on an NVIDIA V100 GPU since they directly compare to NeRF runtimes.

**Neural Volumes [25]** We copy the rendering quality results reported in the NeRF paper and copy the rendering run-time results reported in the AutoInt paper. We assume that the run-times reported in the AutoInt paper are measured on an NVIDIA V100 GPU since the AutoInt paper directly compares these results with NeRF run-times.

**NSVF [24]** We use the average run-time of 1.537 seconds per frame reported by the authors, using early stopping. Performance was measured on an NVIDIA V100 GPU.

**DeRF [34]** We use the DeRF model with 8 heads and 96 channels per head, which achieves the highest ratio of quality to run-time according to the results in their paper. Run-times were measured on an NVIDIA V100 GPU.

**IBRNet [43]** We use the highest quality results (per-scene fine-tuned) in their paper. Run-times were estimated by scaling the NVIDIA V100 GPU NeRF run-times according to the TFLOPs in Table 3 of their paper.

**LLFF [28]** Run-times were measured using the original CUDA implementation on a GTX 2080 Ti (250W).

### C.5. Experiments with Changing 3D Resolution

Table 5 demonstrates that our method is able to achieve even higher rendering speeds and lower storage costs by baking the 3D grids at a lower resolution, at the expense of a slight decrease in rendering quality.

### C.6. Per-Scene Quality and Performance Metrics

Tables 13-15, provide a per-scene breakdown for the quality metrics in the Synthetic 360° scenes. Similar breakdowns for the Real Forward Facing scene can be found in Tables 16-18. Table 19 shows the per-scene frame time and Table 20 shows the per-scene GPU memory consumption our performance ablations: 1) removing the view-dependence MLP, 2) removing the sparsity loss, and 3) switching from 'deferred" rendering back to querying an MLP at each sample along the ray.

| | Synthetic 360° | | | | Real Forward-Facing | | | | Real 360° | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | PSNR ↑ | SSIM ↑ | LPIPS ↓ | MB ↓ | PSNR ↑ | SSIM ↑ | LPIPS ↓ | MB ↓ | PSNR ↑ | SSIM ↑ | LPIPS ↓ | MB ↓ |
| SNeRG (Float) | 30.47 | 0.951 | 0.049 | 6919.9 | 25.74 | 0.823 | 0.180 | 13830.3 | 24.05 | 0.661 | 0.299 | 7238.3 |
| SNeRG (PNG) | 30.38 | 0.950 | 0.050 | 86.7 | 25.63 | 0.818 | 0.183 | 373.2 | 23.97 | 0.662 | 0.293 | 264.7 |
| SNeRG (JPG) | 29.71 | 0.939 | 0.062 | 70.9 | 25.27 | 0.781 | 0.232 | 183.3 | 23.67 | 0.638 | 0.306 | 129.2 |
| SNeRG (H264) | 29.86 | 0.938 | 0.065 | 30.2 | 25.13 | 0.761 | 0.257 | 66.9 | 23.60 | 0.629 | 0.316 | 51.3 |
| JAXNeRF+ | 33.00 | 0.962 | 0.038 | 18.0 | 26.95 | 0.845 | 0.145 | 18.0 | 24.56 | 0.703 | 0.248 | 18.0 |

Table 12: Storage ablation.

| | PSNR ↑ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Chair* | *Drums* | *Ficus* | *Hotdog* | *Lego* | *Materials* | *Mic* | *Ship* |
| AutoInt | 25.55 | 25.60 | 20.78 | 22.47 | 32.33 | 25.09 | 25.90 | 28.10 | 24.15 |
| NV | 26.05 | 28.33 | 22.58 | 24.79 | 30.71 | 26.08 | 24.22 | 27.78 | 23.93 |
| IBRNet | 28.14 | — | — | — | — | — | — | — | — |
| NeRF | 31.00 | 33.00 | 25.01 | 30.13 | 36.18 | 32.54 | 29.62 | 32.91 | 28.65 |
| JAXNeRF | 31.65 | 33.88 | 25.08 | 30.51 | 36.91 | 33.24 | 30.03 | 34.52 | 29.07 |
| NSVF | 31.74 | 33.19 | 25.18 | 31.23 | 37.14 | 32.29 | 32.68 | 34.27 | 27.93 |
| JAXNeRF+ | 33.00 | 35.35 | 25.65 | 32.77 | 37.58 | 35.35 | 30.29 | 36.52 | 30.48 |
| JAXNeRF+ Tinyview | 31.65 | 34.24 | 25.06 | 29.52 | 36.75 | 34.34 | 29.17 | 34.31 | 29.84 |
| JAXNeRF+ Deferred | 30.55 | 33.63 | 23.73 | 28.46 | 35.10 | 34.67 | 26.74 | 33.03 | 29.04 |
| SNeRG (PNG) | 30.38 | 33.24 | 24.57 | 29.32 | 34.33 | 33.82 | 27.21 | 32.60 | 27.97 |
| JAXNeRF+ Diffuse | 27.39 | 29.95 | 21.93 | 22.37 | 32.99 | 32.17 | 24.83 | 28.36 | 26.57 |

Table 13: PSNR, Synthetic 360° scenes.

| | SSIM ↑ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Chair* | *Drums* | *Ficus* | *Hotdog* | *Lego* | *Materials* | *Mic* | *Ship* |
| AutoInt | 0.911 | 0.928 | 0.861 | 0.898 | 0.974 | 0.900 | 0.930 | 0.948 | 0.852 |
| NV | 0.893 | 0.916 | 0.873 | 0.910 | 0.944 | 0.880 | 0.888 | 0.946 | 0.784 |
| IBRNet | 0.942 | — | — | — | — | — | — | — | — |
| NeRF | 0.947 | 0.967 | 0.925 | 0.964 | 0.974 | 0.961 | 0.949 | 0.980 | 0.856 |
| JAXNeRF | 0.952 | 0.974 | 0.927 | 0.967 | 0.979 | 0.968 | 0.952 | 0.987 | 0.865 |
| NSVF | 0.953 | 0.968 | 0.931 | 0.973 | 0.980 | 0.960 | 0.973 | 0.987 | 0.854 |
| JAXNeRF+ | 0.962 | 0.982 | 0.936 | 0.980 | 0.983 | 0.979 | 0.956 | 0.991 | 0.887 |
| JAXNeRF+ Tinyview | 0.954 | 0.978 | 0.925 | 0.966 | 0.979 | 0.975 | 0.946 | 0.986 | 0.880 |
| JAXNeRF+ Deferred | 0.952 | 0.976 | 0.922 | 0.964 | 0.976 | 0.976 | 0.939 | 0.984 | 0.874 |
| SNeRG (PNG) | 0.950 | 0.975 | 0.929 | 0.967 | 0.971 | 0.973 | 0.938 | 0.982 | 0.865 |
| JAXNeRF+ Diffuse | 0.927 | 0.951 | 0.888 | 0.916 | 0.966 | 0.968 | 0.911 | 0.967 | 0.850 |

Table 14: SSIM, Synthetic 360° scenes.

| | LPIPS ↓ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Chair* | *Drums* | *Ficus* | *Hotdog* | *Lego* | *Materials* | *Mic* | *Ship* |
| AutoInt | 0.170 | 0.141 | 0.224 | 0.148 | 0.080 | 0.175 | 0.136 | 0.131 | 0.323 |
| NV | 0.160 | 0.109 | 0.214 | 0.162 | 0.109 | 0.175 | 0.130 | 0.107 | 0.276 |
| IBRNet | 0.072 | — | — | — | — | — | — | — | — |
| NeRF | 0.081 | 0.046 | 0.091 | 0.044 | 0.121 | 0.050 | 0.063 | 0.028 | 0.206 |
| JAXNeRF | 0.051 | 0.027 | 0.070 | 0.033 | 0.030 | 0.030 | 0.048 | 0.013 | 0.156 |
| NSVF | 0.047 | 0.043 | 0.069 | 0.017 | 0.025 | 0.029 | 0.021 | 0.010 | 0.162 |
| JAXNeRF+ | 0.038 | 0.017 | 0.057 | 0.018 | 0.022 | 0.017 | 0.041 | 0.008 | 0.123 |
| JAXNeRF+ Tinyview | 0.047 | 0.020 | 0.079 | 0.030 | 0.028 | 0.020 | 0.051 | 0.016 | 0.130 |
| JAXNeRF+ Deferred | 0.049 | 0.022 | 0.069 | 0.041 | 0.033 | 0.019 | 0.052 | 0.016 | 0.138 |
| SNeRG (PNG) | 0.050 | 0.025 | 0.061 | 0.028 | 0.043 | 0.022 | 0.052 | 0.016 | 0.156 |
| JAXNeRF+ Diffuse | 0.068 | 0.048 | 0.101 | 0.074 | 0.044 | 0.024 | 0.074 | 0.031 | 0.152 |

Table 15: LPIPS, Synthetic 360° scenes.

| | PSNR ↑ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Room* | *Fern* | *Leaves* | *Fortress* | *Orchids* | *Flower* | *T-Rex* | *Horns* |
| LLFF | 24.13 | 28.42 | 22.85 | 19.52 | 29.40 | 18.52 | 25.46 | 24.15 | 24.70 |
| DeRF | 24.81 | 29.72 | 24.87 | 20.64 | 26.84 | 19.97 | 25.66 | 24.86 | 25.89 |
| NeRF | 26.50 | 32.70 | 25.17 | 20.92 | 31.16 | 20.36 | 27.40 | 26.80 | 27.45 |
| IBRNet | 26.73 | — | — | — | — | — | — | — | — |
| JAXNeRF | 26.92 | 33.30 | 24.92 | 21.24 | 31.78 | 20.32 | 28.09 | 27.43 | 28.29 |
| JAXNeRF+ | 26.95 | 33.79 | 24.38 | 20.82 | 31.14 | 20.09 | 28.34 | 27.94 | 29.08 |
| JAXNeRF+ Deferred | 26.61 | 32.63 | 24.88 | 20.67 | 31.28 | 19.72 | 27.40 | 27.72 | 28.56 |
| SNeRG (PNG) | 25.63 | 30.04 | 24.85 | 20.01 | 30.91 | 19.73 | 27.00 | 25.80 | 26.71 |
| JAXNeRF+ Diffuse | 26.31 | 31.44 | 24.98 | 20.64 | 30.46 | 19.89 | 26.95 | 28.06 | 28.03 |

Table 16: PSNR, Real Forward-Facing scenes.

| | SSIM ↑ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Room* | *Fern* | *Leaves* | *Fortress* | *Orchids* | *Flower* | *T-Rex* | *Horns* |
| LLFF | 0.798 | 0.932 | 0.753 | 0.697 | 0.872 | 0.588 | 0.844 | 0.857 | 0.840 |
| DeRF | 0.767 | 0.930 | 0.770 | 0.680 | 0.730 | 0.610 | 0.790 | 0.840 | 0.790 |
| NeRF | 0.811 | 0.948 | 0.792 | 0.690 | 0.881 | 0.641 | 0.827 | 0.880 | 0.828 |
| IBRNet | 0.851 | — | — | — | — | — | — | — | — |
| JAXNeRF | 0.831 | 0.958 | 0.806 | 0.717 | 0.897 | 0.657 | 0.850 | 0.902 | 0.863 |
| JAXNeRF+ | 0.845 | 0.966 | 0.813 | 0.724 | 0.900 | 0.669 | 0.868 | 0.921 | 0.898 |
| JAXNeRF+ Deferred | 0.837 | 0.957 | 0.816 | 0.720 | 0.901 | 0.657 | 0.844 | 0.913 | 0.886 |
| SNeRG (PNG) | 0.818 | 0.936 | 0.802 | 0.696 | 0.889 | 0.655 | 0.835 | 0.882 | 0.852 |
| JAXNeRF+ Diffuse | 0.832 | 0.947 | 0.821 | 0.715 | 0.891 | 0.656 | 0.827 | 0.916 | 0.883 |

Table 17: SSIM, Real Forward-Facing scenes.

| | LPIPS ↓ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Mean | *Room* | *Fern* | *Leaves* | *Fortress* | *Orchids* | *Flower* | *T-Rex* | *Horns* |
| LLFF | 0.212 | 0.155 | 0.247 | 0.216 | 0.173 | 0.313 | 0.174 | 0.222 | 0.193 |
| DeRF | 0.274 | 0.160 | 0.300 | 0.310 | 0.320 | 0.340 | 0.240 | 0.220 | 0.300 |
| NeRF | 0.250 | 0.178 | 0.280 | 0.316 | 0.171 | 0.321 | 0.219 | 0.249 | 0.268 |
| IBRNet | 0.175 | — | — | — | — | — | — | — | — |
| JAXNeRF | 0.173 | 0.086 | 0.207 | 0.247 | 0.108 | 0.266 | 0.156 | 0.143 | 0.173 |
| JAXNeRF+ | 0.145 | 0.066 | 0.176 | 0.233 | 0.097 | 0.238 | 0.124 | 0.113 | 0.119 |
| JAXNeRF+ Deferred | 0.160 | 0.090 | 0.186 | 0.240 | 0.097 | 0.256 | 0.149 | 0.125 | 0.134 |
| SNeRG (PNG) | 0.183 | 0.133 | 0.198 | 0.252 | 0.125 | 0.255 | 0.167 | 0.157 | 0.176 |
| JAXNeRF+ Diffuse | 0.164 | 0.115 | 0.182 | 0.241 | 0.105 | 0.251 | 0.166 | 0.116 | 0.133 |

Table 18: LPIPS, Real Forward-Facing scenes.

| | MLP | $\mathcal{L}_s$ | Defer | Mean | *Chair* | *Drums* | *Ficus* | *Hotdog* | *Lego* | *Materials* | *Mic* | *Ship* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ours | ✓ | ✓ | ✓ | 11.9 | 9.8 | 9.6 | 21.1 | 7.2 | 9.3 | 10.1 | 10.1 | 17.9 |
| 1) | | ✓ | | 9.2 | 6.6 | 6.6 | 19.4 | 5.3 | 6.0 | 7.8 | 7.6 | 14.0 |
| 2) | ✓ | | ✓ | — | 15.3 | 15.7 | — | 24.4 | 25.4 | 19.7 | 12.6 | 27.0 |
| 3) | ✓ | ✓ | | 343.6 | 269.8 | 268.9 | 987.9 | 155.8 | 261.4 | 262.2 | 222.6 | 320.4 |

Table 19: Performance ablation (milliseconds/frame ↓), Synthetic 360° Scenes.

| | MLP | $\mathcal{L}_s$ | Defer | Mean | *Chair* | *Drums* | *Ficus* | *Hotdog* | *Lego* | *Materials* | *Mic* | *Ship* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ours | ✓ | ✓ | ✓ | 1.73 | 0.78 | 0.86 | 4.99 | 0.53 | 0.74 | 0.98 | 1.18 | 3.78 |
| 1) | | ✓ | | 1.73 | 0.78 | 0.86 | 4.99 | 0.53 | 0.74 | 0.98 | 1.18 | 3.78 |
| 2) | ✓ | | ✓ | 4.26 | 2.44 | 3.25 | 7.57 | 4.17 | 3.68 | 4.91 | 2.33 | 5.77 |
| 3) | ✓ | ✓ | | 1.73 | 0.78 | 0.86 | 4.99 | 0.53 | 0.74 | 0.98 | 1.18 | 3.78 |

Table 20: Performance ablation (GPU Memory in GB ↓), Synthetic 360° Scenes.