Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields

Jonathan T. Barron[1] Ben Mildenhall[1] Matthew Tancik[2]
Peter Hedman[1] Ricardo Martin-Brualla[1] Pratul P. Srinivasan[1]
[1]Google [2]UC Berkeley

Abstract

The rendering procedure used by neural radiance fields (NeRF) samples a scene with a single ray per pixel and may therefore produce renderings that are excessively blurred or aliased when training or testing images observe scene content at different resolutions. The straightforward solution of supersampling by rendering with multiple rays per pixel is impractical for NeRF, because rendering each ray requires querying a multilayer perceptron hundreds of times. Our solution, which we call "mip-NeRF" (à la "mipmap"), extends NeRF to represent the scene at a continuously-valued scale. By efficiently rendering anti-aliased conical frustums instead of rays, mip-NeRF reduces objectionable aliasing artifacts and significantly improves NeRF's ability to represent fine details, while also being 7% faster than NeRF and half the size. Compared to NeRF, mip-NeRF reduces average error rates by 17% on the dataset presented with NeRF and by 60% on a challenging multiscale variant of that dataset that we present. Mip-NeRF is also able to match the accuracy of a brute-force supersampled NeRF on our multiscale dataset while being 22× faster.

1. Introduction

Neural volumetric representations such as neural radiance fields (NeRF) [30] have emerged as a compelling strategy for learning to represent 3D objects and scenes from images for the purpose of rendering photorealistic novel views. Although NeRF and its variants have demonstrated impressive results across a range of view synthesis tasks, NeRF's rendering model is flawed in a manner that can cause excessive blurring and aliasing. NeRF replaces traditional discrete sampled geometry with a continuous volumetric function, parameterized as a multilayer perceptron (MLP) that maps from an input 5D coordinate (3D position and 2D viewing direction) to properties of the scene (volume density and view-dependent emitted radiance) at that location. To render a pixel's color, NeRF casts a single ray through that pixel and out into its volumetric representation, queries the MLP for scene properties at samples along that ray, and composites these values into a single color.

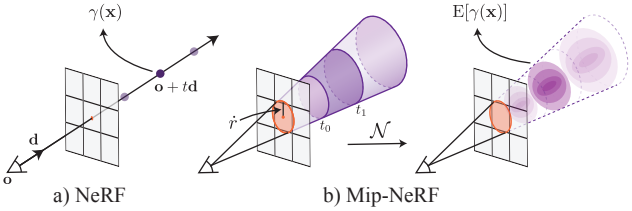

Figure 1: NeRF (a) samples points $\mathbf{x}$ along rays that are traced from the camera center of projection through each pixel, then encodes those points with a positional encoding (PE) γ to produce a feature $\gamma(\mathbf{x})$. Mip-NeRF (b) instead reasons about the 3D *conical frustum* defined by a camera pixel. These conical frustums are then featurized with our integrated positional encoding (IPE), which works by approximating the frustum with a multivariate Gaussian and then computing the (closed form) integral $\mathrm{E}[\gamma(\mathbf{x})]$ over the positional encodings of the coordinates within the Gaussian.

While this approach works well when all training and testing images observe scene content from a roughly constant distance (as done in NeRF and most follow-ups), NeRF renderings exhibit significant artifacts in less contrived scenarios. When the training images observe scene content at multiple resolutions, renderings from the recovered NeRF appear excessively blurred in close-up views and contain aliasing artifacts in distant views. A straightforward solution is to adopt the strategy used in offline raytracing: supersampling each pixel by marching multiple rays through its footprint. But this is prohibitively expensive for neural volumetric representations such as NeRF, which require hundreds of MLP evaluations to render a single ray and several hours to reconstruct a single scene.

In this paper, we take inspiration from the mipmapping approach used to prevent aliasing in computer graphics rendering pipelines. A mipmap represents a signal (typically an image or a texture map) at a set of different discrete downsampling scales and selects the appropriate scale to use for a ray based on the projection of the pixel footprint

onto the geometry intersected by that ray. This strategy is known as *pre*-filtering, since the computational burden of anti-aliasing is shifted from render time (as in the brute force supersampling solution) to a precomputation phase—the mipmap need only be created once for a given texture, regardless of how many times that texture is rendered.

Our solution, which we call mip-NeRF (*multum in parvo* NeRF, as in "mipmap"), extends NeRF to simultaneously represent the prefiltered radiance field for a *continuous* space of scales. The input to mip-NeRF is a 3D Gaussian that represents the region over which the radiance field should be integrated. As illustrated in Figure 1, we can then render a prefiltered pixel by querying mip-NeRF at intervals along a cone, using Gaussians that approximate the conical frustums corresponding to the pixel. To encode a 3D position and its surrounding Gaussian region, we propose a new feature representation: an integrated positional encoding (IPE). This is a generalization of NeRF's positional encoding (PE) that allows a *region* of space to be compactly featurized, as opposed to a single point in space.

Mip-NeRF substantially improves upon the accuracy of NeRF, and this benefit is even greater in situations where scene content is observed at different resolutions (*i.e.* setups where the camera moves closer and farther from the scene). On a challenging multiresolution benchmark we present, mip-NeRF is able to reduce error rates relative to NeRF by 60% on average (see Figure 2 for visualisations). Mip-NeRF's scale-aware structure also allows us to merge the separate "coarse" and "fine" MLPs used by NeRF for hierarchical sampling [30] into a single MLP. As a consequence, mip-NeRF is slightly faster than NeRF ($\sim 7\%$), and has half as many parameters.

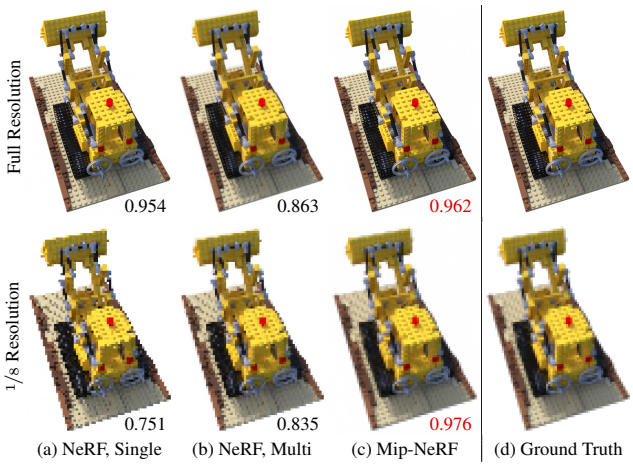

Figure 2: (a, top) A NeRF trained on full-resolution images is capable of producing photorealistic renderings at new view locations, but *only* at the resolution or scale of the training images. (a, bottom) Pulling the camera back and zooming in (or similarly, adjusting the camera intrinsics to reduce image resolution, as is done here) results in renderings that exhibit severe aliasing. (b) Training a NeRF on multi-resolution images ameliorates this issue slightly but results in poor quality renderings across scales: blur at full resolution, and "jaggies" at low resolutions. (c) Mip-NeRF, also trained on multi-resolution images, is capable of producing photorealistic renderings across different scales. SSIMs for each image relative to the ground-truth (d) are inset, with the highest SSIM for both scales shown in red.

2. Related Work

Our work directly extends NeRF [30], a highly influential technique for learning a 3D scene representation from observed images in order to synthesize novel photorealistic views. Here we review the 3D representations used by computer graphics and view synthesis, including recently-introduced continuous neural representations such as NeRF, with a focus on sampling and aliasing.

Anti-aliasing in Rendering Sampling and aliasing are fundamental issues that have been extensively studied throughout the development of rendering algorithms in computer graphics. Reducing aliasing artifacts ("anti-aliasing") is typically done via either supersampling or prefiltering. Supersampling-based techniques [46] cast multiple rays per pixel while rendering in order to sample closer to the Nyquist frequency. This is an effective strategy to reduce aliasing, but it is expensive, as runtime generally scales linearly with the supersampling rate. Supersampling is therefore typically used only in offline rendering contexts. Instead of sampling more rays to match the Nyquist frequency, prefiltering-based techniques use lowpass-filtered versions of scene content to decrease the Nyquist frequency required to render the scene without aliasing. Prefiltering techniques [18, 20, 32, 49] are better suited for realtime rendering, because filtered versions of scene content can be precomputed ahead of time, and the correct "scale" can be used at render time depending on the target sampling rate. In the context of rendering, prefiltering can be thought of as tracing a cone instead of a ray through each pixel [1, 16]: wherever the cone intersects scene content, a precomputed multiscale representation of the scene content (such as a sparse voxel octree [15, 21] or a mipmap [47]) is queried at the scale corresponding to the cone's footprint.

Our work takes inspiration from this line of work in graphics and presents a multiscale scene representation for NeRF. Our strategy differs from multiscale representations used in traditional graphics pipelines in two crucial ways. First, we cannot precompute the multiscale representation because the scene's geometry is not known ahead of time in our problem setting — we are recovering a model of the scene using computer vision, not rendering a predefined CGI asset. Mip-NeRF therefore must learn a prefiltered representation of the scene during training. Second, our notion

of scale is continuous instead of discrete. Instead of representing the scene using multiple copies at a fixed number of scales (like in a mipmap), mip-NeRF learns a single neural scene model that can be queried at arbitrary scales.

Scene Representations for View Synthesis Various scene representations have been proposed for the task of view synthesis: using observed images of a scene to recover a representation that supports rendering novel photorealistic images of the scene from unobserved camera viewpoints. When images of the scene are captured densely, light field interpolation techniques [9, 14, 22] can be used to render novel views without reconstructing an intermediate representation of the scene. Issues related to sampling and aliasing have been thoroughly studied within this setting [7].

Methods that synthesize novel views from sparsely-captured images typically reconstruct explicit representations of the scene's 3D geometry and appearance. Many classic view synthesis algorithms use mesh-based representations along with either diffuse [28] or view-dependent [6, 10, 48] textures. Mesh-based representations can be stored efficiently and are naturally compatible with existing graphics rendering pipelines. However, using gradient-based methods to optimize mesh geometry and topology is typically difficult due to discontinuities and local minima. Volumetric representations have therefore become increasingly popular for view synthesis. Early approaches directly color voxel grids using observed images [37], and more recent volumetric approaches use gradient-based learning to train deep networks to predict voxel grid representations of scenes [12, 25, 29, 38, 41, 53]. Discrete voxel-based representations are effective for view synthesis, but they do not scale well to scenes at higher resolutions.

A recent trend within computer vision and graphics research is to replace these discrete representations with *coordinate-based neural representations*, which represent 3D scenes as continuous functions parameterized by MLPs that map from a 3D coordinate to properties of the scene at that location. Some recent methods use coordinate-based neural representations to model scenes as implicit surfaces [31, 50], but the majority of recent view synthesis methods are based on the volumetric NeRF representation [30]. NeRF has inspired many subsequent works that extend its continuous neural volumetric representation for generative modeling [8, 36], dynamic scenes [23, 33], non-rigidly deforming objects [13, 34], phototourism settings with changing illumination and occluders [26, 43], and reflectance modeling for relighting [2, 3, 40].

Relatively little attention has been paid to the issues of sampling and aliasing within the context of view synthesis using coordinate-based neural representations. Discrete representations used for view synthesis, such as polygon meshes and voxel grids, can be efficiently rendered without aliasing using traditional multiscale prefiltering approaches such as mipmaps and octrees. However, coordinate-based neural representations for view synthesis can currently only be anti-aliased using supersampling, which exacerbates their already slow rendering procedure. Recent work by Takikawa *et al*. [42] proposes a multiscale representation based on sparse voxel octrees for continuous neural representations of implicit surfaces, but their approach requires that the scene geometry be known a priori, as opposed to our view synthesis setting where the only inputs are observed images. Mip-NeRF addresses this open problem, enabling the efficient rendering of anti-aliased images during both training and testing as well as the use of multiscale images during training.

2.1. Preliminaries: NeRF

NeRF uses the weights of a multilayer perceptron (MLP) to represent a scene as a continuous volumetric field of particles that block and emit light. NeRF renders each pixel of a camera as follows: A ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ is emitted from the camera's center of projection $\mathbf{o}$ along the direction $\mathbf{d}$ such that it passes through the pixel. A sampling strategy (discussed later) is used to determine a vector of sorted distances $\mathbf{t}$ between the camera's predefined near and far planes t_n and t_f. For each distance $t_k \in \mathbf{t}$, we compute its corresponding 3D position along the ray $\mathbf{x} = \mathbf{r}(t_k)$, then transform each position using a positional encoding:

$$\gamma(\mathbf{x}) = \Big[\sin(\mathbf{x}), \cos(\mathbf{x}), \ldots, \sin\big(2^{L-1}\mathbf{x}\big), \cos\big(2^{L-1}\mathbf{x}\big)\Big]^{\mathrm{T}} . \quad (1)$$

This is simply the concatenation of the sines and cosines of each dimension of the 3D position $\mathbf{x}$ scaled by powers of 2 from 1 to 2^{L-1}, where L is a hyperparameter. The fidelity of NeRF depends critically on the use of positional encoding, as it allows the MLP parameterizing the scene to behave as an interpolation function, where L determines the bandwidth of the interpolation kernel (see Tancik *et al*. [44] for details). The positional encoding of each ray position $\gamma(\mathbf{r}(t_k))$ is provided as input to an MLP parameterized by weights Θ, which outputs a density τ and an RGB color $\mathbf{c}$:

$$\forall t_k \in \mathbf{t}, \quad [\tau_k, \mathbf{c}_k] = \mathrm{MLP}(\gamma(\mathbf{r}(t_k)); \Theta) . \quad (2)$$

The MLP also takes the view direction as input, which is omitted from notation for simplicity. These estimated densities and colors are used to approximate the volume rendering integral using numerical quadrature, as per Max [27]:

$$\mathbf{C}(\mathbf{r}; \Theta, \mathbf{t}) = \sum_k T_k (1 - \exp(-\tau_k (t_{k+1} - t_k)))\mathbf{c}_k ,$$
$$\text{with} \quad T_k = \exp\left(-\sum_{k'<k} \tau_{k'}(t_{k'+1} - t_{k'})\right) , \quad (3)$$

where $\mathbf{C}(\mathbf{r}; \Theta, \mathbf{t})$ is the final predicted color of the pixel.

With this procedure for rendering a NeRF parameterized by Θ, training a NeRF is straightforward: using a set of

observed images with known camera poses, we minimize the sum of squared differences between all input pixel values and all predicted pixel values using gradient descent. To improve sample efficiency, NeRF trains two separate MLPs, one "coarse" and one "fine", with parameters Θ^c and Θ^f:

$$\min_{\Theta^c,\Theta^f} \sum_{\mathbf{r}\in\mathcal{R}} \Big(\left\| \mathbf{C}^*(\mathbf{r}) - \mathbf{C}(\mathbf{r};\Theta^c,\mathbf{t}^c) \right\|_2^2 + \left\| \mathbf{C}^*(\mathbf{r}) - \mathbf{C}(\mathbf{r};\Theta^f,\text{sort}(\mathbf{t}^c \cup \mathbf{t}^f)) \right\|_2^2 \Big), \tag{4}$$

where $\mathbf{C}^*(\mathbf{r})$ is the observed pixel color taken from the input image, and $\mathcal{R}$ is the set of all pixels/rays across all images. Mildenhall *et al*. construct $\mathbf{t}^c$ by sampling 64 evenly-spaced random t values with stratified sampling. The compositing weights $w_k = T_k\left(1-\exp(-\tau_k(t_{k+1}-t_k))\right)$ produced by the "coarse" model are then taken as a piecewise constant PDF describing the distribution of visible scene content, and 128 new t values are drawn from that PDF using inverse transform sampling to produce $\mathbf{t}^f$. The union of these 192 t values are then sorted and passed to the "fine" MLP to produce a final predicted pixel color.

3. Method

As discussed, NeRF's point-sampling makes it vulnerable to issues related to sampling and aliasing: Though a pixel's color is the integration of all incoming radiance within the pixel's frustum, NeRF casts a single infinitesimally narrow ray per pixel, resulting in aliasing. Mip-NeRF ameliorates this issue by casting a *cone* from each pixel. Instead of performing point-sampling along each ray, we divide the cone being cast into a series of *conical frustums* (cones cut perpendicular to their axis). And instead of constructing positional encoding (PE) features from an infinitesimally small point in space, we construct an *integrated* positional encoding (IPE) representation of the volume covered by each conical frustum. These changes allow the MLP to reason about the size and shape of each conical frustum, instead of just its centroid. The ambiguity resulting from NeRF's insensitivity to scale and mip-NeRF's solution to this problem are visualized in Figure 3. This use of conical frustums and IPE features also allows us to reduce NeRF's two separate "coarse" and "fine" MLPs into a single multiscale MLP, which increases training and evaluation speed and reduces model size by 50%.

3.1. Cone Tracing and Positional Encoding

Here we describe mip-NeRF's rendering and featurization procedure, in which we cast a cone and featurize conical frustums along that cone. As in NeRF, images in mip-NeRF are rendered one pixel at a time, so we can describe our procedure in terms of an individual pixel of interest being rendered. For that pixel, we cast a cone from the camera's center of projection $\mathbf{o}$ along the direction $\mathbf{d}$ that passes through the pixel's center. The apex of that cone lies at $\mathbf{o}$, and the radius of the cone at the image plane $\mathbf{o}+\mathbf{d}$ is parameterized as $\dot{r}$. We set $\dot{r}$ to the width of the pixel in world coordinates scaled by $2/\sqrt{12}$, which yields a cone whose section on the image plane has a variance in x and y that matches the variance of the pixel's footprint. The set of positions $\mathbf{x}$ that lie within a conical frustum between two t values $[t_0, t_1]$ (visualized in Figure 1) is:

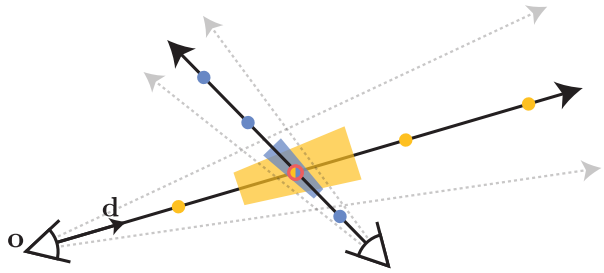

Figure 3: NeRF works by extracting point-sampled positional encoding features (shown here as dots) along each pixel's ray. Those point-sampled features ignore the shape and size of the volume viewed by each ray, so two different cameras imaging the same position at different scales may produce the same ambiguous point-sampled feature, thereby significantly degrading NeRF's performance. In contrast, Mip-NeRF casts cones instead of rays and explicitly models the volume of each sampled conical frustum (shown here as trapezoids), thus resolving this ambiguity.

$$\text{F}(\mathbf{x},\mathbf{o},\mathbf{d},\dot{r},t_0,t_1) = \mathbb{1}\left\{ \left(t_0 < \frac{\mathbf{d}^{\text{T}}(\mathbf{x}-\mathbf{o})}{\|\mathbf{d}\|_2^2} < t_1 \right) \wedge \left(\frac{\mathbf{d}^{\text{T}}(\mathbf{x}-\mathbf{o})}{\|\mathbf{d}\|_2\|\mathbf{x}-\mathbf{o}\|_2} > \frac{1}{\sqrt{1+(\dot{r}/\|\mathbf{d}\|_2)^2}} \right) \right\}, \tag{5}$$

where $\mathbb{1}\{\cdot\}$ is an indicator function: $\text{F}(\mathbf{x},\cdot) = 1$ iff $\mathbf{x}$ is within the conical frustum defined by $(\mathbf{o},\mathbf{d},\dot{r},t_0,t_1)$.

We must now construct a featurized representation of the volume inside this conical frustum. Ideally, this featurized representation should be of a similar form to the positional encoding features used in NeRF, as Mildenhall *et al*. show that this feature representation is critical to NeRF's success [30]. There are many viable approaches for this (see the supplement for further discussion) but the simplest and most effective solution we found was to simply compute the expected positional encoding of all coordinates that lie within the conical frustum:

$$\gamma^*(\mathbf{o},\mathbf{d},\dot{r},t_0,t_1) = \frac{\int \gamma(\mathbf{x})\,\text{F}(\mathbf{x},\mathbf{o},\mathbf{d},\dot{r},t_0,t_1)\,d\mathbf{x}}{\int \text{F}(\mathbf{x},\mathbf{o},\mathbf{d},\dot{r},t_0,t_1)\,d\mathbf{x}}. \tag{6}$$

However, it is unclear how such a feature could be computed efficiently, as the integral in the numerator has no

closed form solution. We therefore approximate the conical frustum with a multivariate Gaussian which allows for an efficient approximation to the desired feature, which we will call an "integrated positional encoding" (IPE).

To approximate a conical frustum with a multivariate Gaussian, we must compute the mean and covariance of $\mathrm{F}(\mathbf{x}, \cdot)$. Because each conical frustum is assumed to be circular, and because conical frustums are symmetric around the axis of the cone, such a Gaussian is fully characterized by three values (in addition to $\mathbf{o}$ and $\mathbf{d}$): the mean distance along the ray μ_t, the variance along the ray σ_t^2, and the variance perpendicular to the ray σ_r^2:

$$\mu_t = t_\mu + \frac{2t_\mu t_\delta^2}{3t_\mu^2 + t_\delta^2}, \quad \sigma_t^2 = \frac{t_\delta^2}{3} - \frac{4t_\delta^4(12t_\mu^2 - t_\delta^2)}{15(3t_\mu^2 + t_\delta^2)^2},$$
$$\sigma_r^2 = \dot{r}^2\left(\frac{t_\mu^2}{4} + \frac{5t_\delta^2}{12} - \frac{4t_\delta^4}{15(3t_\mu^2 + t_\delta^2)}\right). \tag{7}$$

These quantities are parameterized with respect to a midpoint $t_\mu = (t_0 + t_1)/2$ and a half-width $t_\delta = (t_1 - t_0)/2$, which is critical for numerical stability. Please refer to the supplement for a detailed derivation. We can transform this Gaussian from the coordinate frame of the conical frustum into world coordinates as follows:

$$\boldsymbol{\mu} = \mathbf{o} + \mu_t \mathbf{d}, \quad \boldsymbol{\Sigma} = \sigma_t^2\left(\mathbf{d}\mathbf{d}^{\mathrm{T}}\right) + \sigma_r^2\left(\mathbf{I} - \frac{\mathbf{d}\mathbf{d}^{\mathrm{T}}}{\|\mathbf{d}\|_2^2}\right), \tag{8}$$

giving us our final multivariate Gaussian.

Next, we derive the IPE, which is the expectation of a positionally-encoded coordinate distributed according to the aforementioned Gaussian. To accomplish this, it is helpful to first rewrite the PE in Equation 1 as a Fourier feature [35, 44]:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 0 & & 2^{L-1} & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & \cdots & 0 & 2^{L-1} & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & & 0 & 0 & 2^{L-1} \end{bmatrix}^{\mathrm{T}}, \quad \gamma(\mathbf{x}) = \begin{bmatrix} \sin(\mathbf{P}\mathbf{x}) \\ \cos(\mathbf{P}\mathbf{x}) \end{bmatrix}. \tag{9}$$

This reparameterization allows us to derive a closed form for IPE. Using the fact that the covariance of a linear transformation of a variable is a linear transformation of the variable's covariance ($\mathrm{Cov}[\mathbf{A}\mathbf{x}, \mathbf{B}\mathbf{y}] = \mathbf{A}\,\mathrm{Cov}[\mathbf{x}, \mathbf{y}]\mathbf{B}^{\mathrm{T}}$) we can identify the mean and covariance of our conical frustum Gaussian after it has been lifted into the PE basis $\mathbf{P}$:

$$\boldsymbol{\mu}_\gamma = \mathbf{P}\boldsymbol{\mu}, \quad \boldsymbol{\Sigma}_\gamma = \mathbf{P}\boldsymbol{\Sigma}\mathbf{P}^{\mathrm{T}}. \tag{10}$$

The final step in producing an IPE feature is computing the expectation over this lifted multivariate Gaussian, modulated by the sine and the cosine of position. These expectations have simple closed-form expressions:

$$\mathrm{E}_{x\sim\mathcal{N}(\mu,\sigma^2)}[\sin(x)] = \sin(\mu)\exp\left(-(1/2)\sigma^2\right), \tag{11}$$
$$\mathrm{E}_{x\sim\mathcal{N}(\mu,\sigma^2)}[\cos(x)] = \cos(\mu)\exp\left(-(1/2)\sigma^2\right). \tag{12}$$

We see that this expected sine or cosine is simply the sine or cosine of the mean attenuated by a Gaussian function of the variance. With this we can compute our final IPE feature as the expected sines and cosines of the mean and the diagonal of the covariance matrix:

$$\gamma(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathrm{E}_{\mathbf{x}\sim\mathcal{N}(\boldsymbol{\mu}_\gamma, \boldsymbol{\Sigma}_\gamma)}[\gamma(\mathbf{x})] \tag{13}$$
$$= \begin{bmatrix} \sin(\boldsymbol{\mu}_\gamma) \circ \exp(-(1/2)\,\mathrm{diag}(\boldsymbol{\Sigma}_\gamma)) \\ \cos(\boldsymbol{\mu}_\gamma) \circ \exp(-(1/2)\,\mathrm{diag}(\boldsymbol{\Sigma}_\gamma)) \end{bmatrix}, \tag{14}$$

where $\circ$ denotes element-wise multiplication. Because positional encoding encodes each dimension independently, this expected encoding relies on only the marginal distribution of $\gamma(\mathbf{x})$, and only the diagonal of the covariance matrix (a vector of per-dimension variances) is needed. Because $\boldsymbol{\Sigma}_\gamma$ is prohibitively expensive to compute due its relatively large size, we directly compute the diagonal of $\boldsymbol{\Sigma}_\gamma$:

$$\mathrm{diag}(\boldsymbol{\Sigma}_\gamma) = \left[\mathrm{diag}(\boldsymbol{\Sigma}), 4\,\mathrm{diag}(\boldsymbol{\Sigma}), \ldots, 4^{L-1}\,\mathrm{diag}(\boldsymbol{\Sigma})\right]^{\mathrm{T}} \tag{15}$$

This vector depends on just the diagonal of the 3D position's covariance $\boldsymbol{\Sigma}$, which can be computed as:

$$\mathrm{diag}(\boldsymbol{\Sigma}) = \sigma_t^2(\mathbf{d} \circ \mathbf{d}) + \sigma_r^2\left(\mathbf{1} - \frac{\mathbf{d} \circ \mathbf{d}}{\|\mathbf{d}\|_2^2}\right). \tag{16}$$

If these diagonals are computed directly, IPE features are roughly as expensive as PE features to construct.

Figure 4 visualizes the difference between IPE and conventional PE features in a toy 1D domain. IPE features behave intuitively: If a particular frequency in the positional encoding has a period that is larger than the width of the interval being used to construct the IPE feature, then the encoding at that frequency is unaffected. But if the period is smaller than the interval (in which case the PE over that interval will oscillate repeatedly), then the encoding at that frequency is scaled down towards zero. In short, IPE preserves frequencies that are constant over an interval and softly "removes" frequencies that vary over an interval, while PE preserves all frequencies up to some manually-tuned hyperparameter L. By scaling each sine and cosine in this way, IPE features are effectively *anti-aliased* positional encoding features that smoothly encode the size and shape of a volume of space. IPE also effectively removes L as a hyperparameter: it can simply be set to an extremely large value and then never tuned (see supplement).

3.2. Architecture

Aside from cone-tracing and IPE features, mip-NeRF behaves similarly to NeRF, as described in Section 2.1. For each pixel being rendered, instead of a ray as in NeRF, a cone is cast. Instead of sampling n values for t_k along the ray, we sample $n + 1$ values for t_k, computing IPE features for the interval spanning each adjacent pair of sampled t_k values as previously described. These IPE features

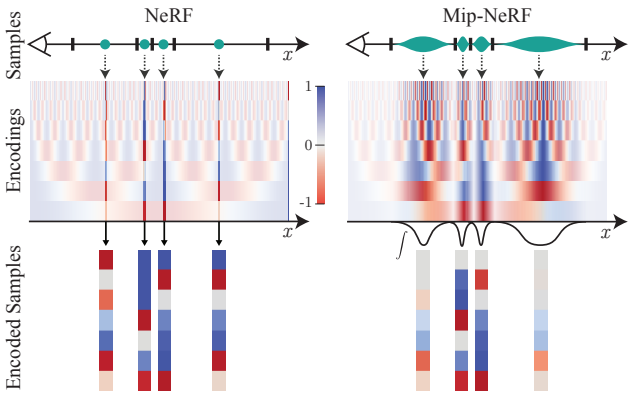

Figure 4: Toy 1D visualizations of the positional encoding (PE) used by NeRF (left) and our integrated positional encoding (IPE) (right). Because NeRF samples points along each ray and encodes all frequencies equally, the high-frequency PE features are aliased, which results in rendering artifacts. By integrating PE features over each interval, the high frequency dimensions of IPE features shrink towards zero when the period of the frequency is small compared to the size of the interval being integrated, resulting in anti-aliased features that implicitly encode the size (and in higher dimensions, the shape) of the interval.

are passed as input into an MLP to produce a density τ_k and a color $\mathbf{c}_k$, as in Equation 2. Rendering in mip-NeRF follows Equation 3.

Recall that NeRF uses a hierarchical sampling procedure with two distinct MLPs, one "coarse" and one "fine" (see Equation 4). This was necessary in NeRF because its PE features meant that its MLPs were only able to learn a model of the scene for *one single scale*. But our cone casting and IPE features allow us to explicitly encode scale into our input features and thereby enable an MLP to learn a *multiscale* representation of the scene. Mip-NeRF therefore uses a single MLP with parameters Θ, which we repeatedly query in a hierarchical sampling strategy. This has multiple benefits: model size is cut in half, renderings are more accurate, sampling is more efficient, and the overall algorithm becomes simpler. Our optimization problem is:

$$\min_{\Theta} \sum_{\mathbf{r}\in\mathcal{R}} \left(\lambda \left\|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}(\mathbf{r};\Theta,\mathbf{t}^c)\right\|_2^2 + \left\|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}(\mathbf{r};\Theta,\mathbf{t}^f)\right\|_2^2\right) \tag{17}$$

Because we have a single MLP, the "coarse" loss must be balanced against the "fine" loss, which is accomplished using a hyperparameter λ (we set $\lambda = 0.1$ in all experiments). As in Mildenhall *et al*. [30], our coarse samples $\mathbf{t}^c$ are produced with stratified sampling, and our fine samples $\mathbf{t}^f$ are sampled from the resulting alpha compositing weights $\mathbf{w}$ using inverse transform sampling. Unlike NeRF, in which the fine MLP is given the sorted union of 64 coarse samples and 128 fine samples, in mip-NeRF we simply sample 128 samples for the coarse model and 128 samples from the fine model (yielding the same number of total MLP evaluations as in NeRF, for fair comparison). Before sampling $\mathbf{t}^f$, we modify the weights $\mathbf{w}$ slightly:

$$w'_k = \frac{1}{2}(\max(w_{k-1}, w_k) + \max(w_k, w_{k+1})) + \alpha\,. \tag{18}$$

We filter $\mathbf{w}$ with a 2-tap max filter followed by a 2-tap blur filter (a "blurpool" [51]), which produces a wide and smooth upper envelope on $\mathbf{w}$. A hyperparameter α is added to that envelope before it is re-normalized to sum to 1, which ensures that some samples are drawn even in empty regions of space (we set $\alpha = 0.01$ in all experiments).

Mip-NeRF is implemented on top of JaxNeRF [11], a JAX [4] reimplementation of NeRF that achieves better accuracy and trains faster than the original TensorFlow implementation. We follow NeRF's training procedure: 1 million iterations of Adam [19] with a batch size of 4096 and a learning rate that is annealed logarithmically from $5 \cdot 10^{-4}$ to $5 \cdot 10^{-6}$. See the supplement for additional details and some additional differences between JaxNeRF and mip-NeRF that do not affect performance significantly and are incidental to our primary contributions: cone-tracing, IPE, and the use of a single multiscale MLP.

4. Results

We evaluate mip-NeRF on the Blender dataset presented in the original NeRF paper [30] and also on a simple multiscale variant of that dataset designed to better probe accuracy on multi-resolution scenes and to highlight NeRF's critical vulnerability on such tasks. We report the three error metrics used by NeRF: PSNR, SSIM [45], and LPIPS [52]. To enable easier comparison, we also present an "average" error metric that summarizes all three metrics: the geometric mean of $\text{MSE} = 10^{-\text{PSNR}/10}$, $\sqrt{1-\text{SSIM}}$ (as per [5]), and LPIPS. We additionally report runtimes (median and median absolute deviation of wall time) as well as the number of network parameters for each variant of NeRF and mip-NeRF. All JaxNeRF and mip-NeRF experiments are trained on a TPU v2 with 32 cores [17].

We constructed our multiscale Blender benchmark because the original Blender dataset used by NeRF has a subtle but critical weakness: all cameras have the same focal length and resolution and are placed at the same distance from the object. As a result, this Blender task is significantly easier than most real-world datasets, where cameras may be more close or more distant from the subject or may zoom in and out. The limitation of this dataset is complemented by the limitations of NeRF: despite NeRF's tendency to produce aliased renderings, it is able to produce excellent results on the Blender dataset because that dataset systematically avoids this failure mode.

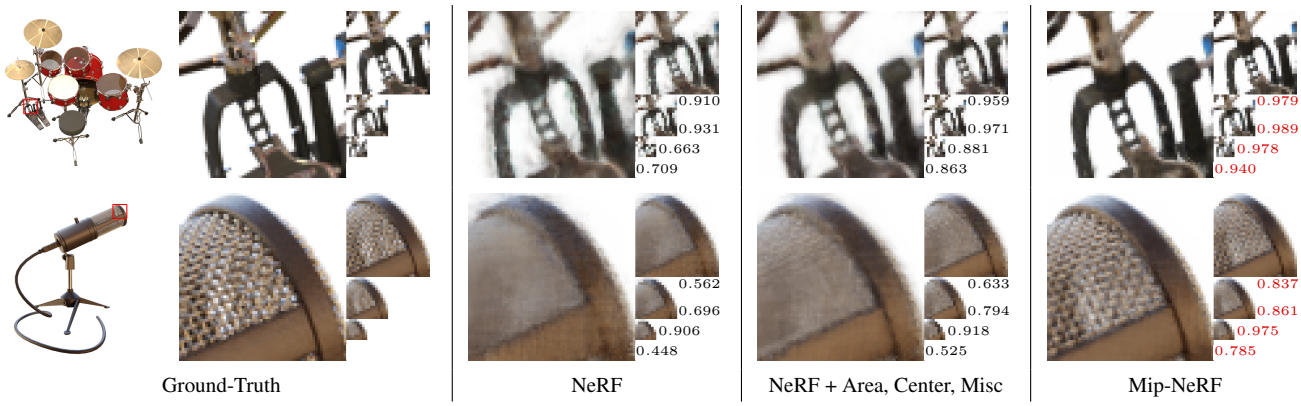

Figure 5: Visualizations of the output of mip-NeRF compared to the ground truth, NeRF, and an improved version of NeRF on test set images from two scenes in our multiscale Blender dataset. We visualize a cropped region of both scenes at 4 different scales, displayed as an image pyramid with the SSIM for each scale shown to its lower right and with the highest SSIM at each scale highlighted in red. Mip-NeRF outperforms NeRF and its improved version by a significant margin, both visually and quantitatively. See the supplement for more such visualizations.

	PSNR ↑				SSIM ↑				LPIPS ↓						
	Full Res.	$1/2$ Res.	$1/4$ Res.	$1/8$ Res.	Full Res.	$1/2$ Res.	$1/4$ Res.	$1/8$ Res.	Full Res.	$1/2$ Res.	$1/4$ Res.	$1/8$ Res	Avg. ↓	Time (hours)	# Params
NeRF (Jax Impl.) [11, 30]	31.196	30.647	26.252	22.533	0.9498	0.9560	0.9299	0.8709	0.0546	0.0342	0.0428	0.0750	0.0288	3.05 ± 0.04	1,191K
NeRF + Area Loss	27.224	29.578	29.445	25.039	0.9113	0.9394	0.9524	0.9176	0.1041	0.0677	0.0406	0.0469	0.0305	3.03 ± 0.03	1,191K
NeRF + Area, Centered Pixels	29.893	32.118	33.399	29.463	0.9376	0.9590	0.9728	0.9620	0.0747	0.0405	0.0245	0.0398	0.0191	3.02 ± 0.05	1,191K
NeRF + Area, Center, Misc.	29.900	32.127	33.404	29.470	0.9378	0.9592	0.9730	0.9622	0.0743	0.0402	0.0243	0.0394	0.0190	2.94 ± 0.02	1,191K
Mip-NeRF	32.629	34.336	35.471	35.602	0.9579	0.9703	0.9786	0.9833	0.0469	0.0260	0.0168	0.0120	0.0114	2.84 ± 0.01	612K
Mip-NeRF w/o Misc.	32.610	34.333	35.497	35.638	0.9577	0.9703	0.9787	0.9834	0.0470	0.0259	0.0167	0.0120	0.0114	2.82 ± 0.03	612K
Mip-NeRF w/o Single MLP	32.401	34.131	35.462	35.967	0.9566	0.9693	0.9780	0.9834	0.0479	0.0268	0.0169	0.0116	0.0115	3.40 ± 0.01	1,191K
Mip-NeRF w/o Area Loss	33.059	34.280	33.866	30.714	0.9605	0.9704	0.9747	0.9679	0.0427	0.0256	0.0213	0.0308	0.0139	2.82 ± 0.01	612K
Mip-NeRF w/o IPE	29.876	32.160	33.679	29.647	0.9384	0.9602	0.9742	0.9633	0.0742	0.0393	0.0226	0.0378	0.0186	2.79 ± 0.01	612K

Table 1: A quantitative comparison of mip-NeRF and its ablations against NeRF and several NeRF variants on the test set of our multiscale Blender dataset. See the text for details.

Multiscale Blender Dataset Our multiscale Blender dataset is a straightforward modification to NeRF's Blender dataset, designed to probe aliasing and scale-space reasoning. This dataset was constructed by taking each image in the Blender dataset, box downsampling it a factor of 2, 4, and 8 (and modifying the camera intrinsics accordingly), and combining the original images and the three downsampled images into one single dataset. Due to the nature of projective geometry, this is similar to re-rendering the original dataset where the distance to the camera has been increased by scale factors of 2, 4, and 8. When training mip-NeRF on this dataset, we scale the loss of each pixel by the area of that pixel's footprint in the original image (the loss for pixels from the $1/4$ images is scaled by 16, etc) so that the few low-resolution pixels have comparable influence to the many high-resolution pixels. The average error metric for this task uses the arithmetic mean of each error metric across all four scales.

The performance of mip-NeRF for this multiscale dataset can be seen in Table 1. Because NeRF is the state of the art on the Blender dataset (as will be shown in Table 2), we evaluate against only NeRF and several improved versions of NeRF: "Area Loss" adds the aforementioned scaling of the loss function by pixel area used by mip-NeRF, "Centered Pixels" adds a half-pixel offset added to each ray's direction such that rays pass through the center of each pixel (as opposed to the corner of each pixel as was done in Mildenhall *et al.*) and "Misc" adds some small changes that slightly improve the stability of training (see supplement). We also evaluate against several ablations of mip-NeRF: "w/o Misc" removes those small changes, "w/o Single MLP" uses NeRF's two-MLP training scheme from Equation 4, "w/o Area Loss" removes the loss scaling by pixel area, and "w/o IPE" uses PE instead of IPE, which causes mip-NeRF to use NeRF's ray-casting (with centered pixels) instead of our cone-casting.

Mip-NeRF reduces average error by 60% on this task and outperforms NeRF by a large margin on all metrics and all scales. "Centering" pixels improves NeRF's performance substantially, but not enough to approach mip-NeRF. Removing IPE features causes mip-NeRF's performance to degrade to the performance of "Centered" NeRF, thereby demonstrating that cone-casting and IPE features are the primary factors driving performance (though the area loss contributes substantially). The "Single MLP" mip-NeRF ablation performs well but has twice as many parameters and is nearly 20% slower than mip-NeRF (likely due to this ablation's need to sort t values and poor hardware through-

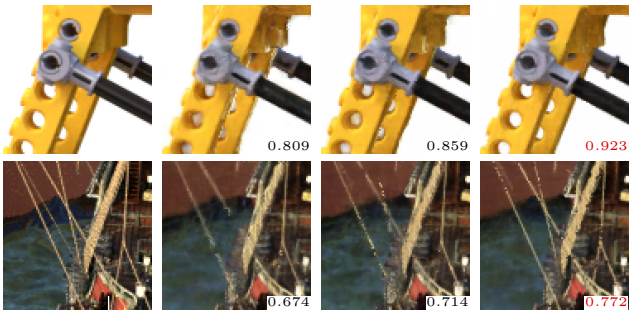

Figure 6: Even on the less challenging single-scale Blender dataset of Mildenhall *et al*. [30], mip-NeRF significantly outperforms NeRF and our improved version of NeRF, particularly on small or thin objects such as the holes of the LEGO truck (top) and the ropes of the ship (bottom).

	PSNR ↑	SSIM ↑	LPIPS ↓	Avg. ↓	Time (hours)	# Params
SRN [39]	22.26	0.846	0.170	0.0735	-	-
Neural Volumes [25]	26.05	0.893	0.160	0.0507	-	-
LLFF [29]	24.88	0.911	0.114	0.0480	~0.16	-
NSVF [24]	31.74	0.953	0.047	0.0190	-	3.2M - 16M
NeRF (TF Impl.) [30]	31.01	0.947	0.081	0.0245	>12	1,191K
NeRF (Jax Impl.) [11, 30]	31.74	0.953	0.050	0.0194	3.05 ± 0.01	1,191K
NeRF + Centered Pixels	32.30	0.957	0.046	0.0178	2.99 ± 0.06	1,191K
NeRF + Center, Misc.	32.28	0.957	0.046	0.0178	3.06 ± 0.03	1,191K
Mip-NeRF	33.09	0.961	0.043	0.0161	2.89 ± 0.00	612K
Mip-NeRF w/o Misc.	33.04	0.960	0.043	0.0162	2.89 ± 0.01	612K
Mip-NeRF w/o Single MLP	32.71	0.959	0.044	0.0168	3.63 ± 0.02	1,191K
Mip-NeRF w/o IPE	32.48	0.958	0.045	0.0173	2.84 ± 0.00	612K

Table 2: A comparison of mip-NeRF and its ablations against several baseline algorithms and variants of NeRF on the single-scale Blender dataset of Mildenhall *et al*. [30]. Training times taken from prior work (when available) are indicated in gray, as they are not directly comparable.

put due to its changing tensor sizes across its "coarse" and "fine" scales). Mip-NeRF is also $\sim 7\%$ faster than NeRF. See Figure 9 and the supplement for visualizations.

Blender Dataset Though the sampling issues that mip-NeRF was designed to fix are most prominent in the Multiscale Blender dataset, mip-NeRF also outperforms NeRF on the easier single-scale Blender dataset presented in Mildenhall *et al*. [30], as shown in Table 2. We evaluate against the baselines used by NeRF, NSVF [24], and the same variants and ablations that were used previously (excluding "Area Loss", which is not used by mip-NeRF for this task). Though less striking than the multiscale Blender dataset, mip-NeRF is able to reduce average error by $\sim 17\%$ compared to NeRF while also being faster. This improvement in performance is most visually apparent in challenging cases such as small or thin structures, as shown in Figure 6.

Supersampling As discussed in the introduction, mip-NeRF is a prefiltering approach for anti-aliasing. An alternative approach is supersampling, which can be accomplished in NeRF by casting multiple jittered rays per pixel. Because our multiscale dataset consists of downsampled

	PSNR ↑					Avg. Time
	Full Res.	1/2 Res.	1/4 Res.	1/8 Res.	Mean	(sec./MP)
NeRF + Area, Center, Misc.	29.90	32.13	33.40	29.47	31.23	2.61
SS NeRF + Area, Center, Misc.	32.25	34.27	35.99	35.73	34.56	55.52
Mip-NeRF	32.60	34.30	35.41	35.55	34.46	2.48
SS Mip-NeRF	32.60	34.78	36.59	36.16	35.03	52.75

Table 3: A comparison of mip-NeRF and our improved NeRF variant where both algorithms are supersampled ("SS"). Mip-NeRF nearly matches the accuracy of "SS NeRF" while being $22\times$ faster. Adding supersampling to mip-NeRF improves its accuracy slightly. We report times for rendering the test set, normalized to seconds-per-megapixel (training times are the same as Tables 1 and 2).

versions of full-resolution images, we can construct a "supersampled NeRF" by training a NeRF (the "NeRF + Area, Center, Misc." variant that performed best previously) using *only* full-resolution images, and then rendering *only* full-resolution images, which we then manually downsample. This baseline has an unfair advantage: we manually remove the low-resolution images in the multiscale dataset, which would otherwise degrade NeRF's performance as previously demonstrated. This strategy is not viable in most real-world datasets, as it is usually not possible to known a-priori which images correspond to which scales of image content. Despite this baseline's advantage, mip-NeRF matches its accuracy while being $\sim 22\times$ faster (see Table 3).

5. Conclusion

We have presented mip-NeRF, a multiscale NeRF-like model that addresses the inherent aliasing of NeRF. NeRF works by casting rays, encoding the positions of points along those rays, and training separate neural networks at distinct scales. In contrast, mip-NeRF casts *cones*, encodes the positions *and sizes* of conical frustums, and trains a *single* neural network that models the scene at multiple scales. By reasoning explicitly about sampling and scale, mip-NeRF is able to reduce error rates relative to NeRF by 60% on our own multiscale dataset, and by 17% on NeRF's single-scale dataset, while also being 7% faster than NeRF. Mip-NeRF is also able to match the accuracy of a brute-force supersampled NeRF variant, while being $22\times$ faster. We hope that the general techniques presented here will be valuable to other researchers working to improve the performance of raytracing-based neural rendering models.

Acknowledgements We thank Janne Kontkanen and David Salesin for their comments on the text, Paul Debevec for constructive discussions, and Boyang Deng for JaxNeRF. MT is funded by an NSF Graduate Fellowship.

References

[1] John Amanatides. Ray tracing with cones. *SIGGRAPH*, 1984.

[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.

[3] Mark Boss, Raphael Braun, Varun Jampani, Jonathan T. Barron, Ce Liu, and Hendrik P. A. Lensch. NeRD: Neural reflectance decomposition from image collections. *arXiv cs.CV 2012.03918*, 2020.

[4] 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`.

[5] Dominique Brunet, Edward R Vrscay, and Zhou Wang. On the mathematical properties of the structural similarity index. *IEEE TIP*, 2011.

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

[7] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan, and Heung-Yeung Shum. Plenoptic sampling. *SIGGRAPH*, 2000.

[8] 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.

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

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

[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`.

[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.

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

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

[15] Eric Heitz and Fabrice Neyret. Representing appearance and pre-filtering subpixel data in sparse voxel octrees. *Eurographics / ACM SIGGRAPH Symposium on High Performance Graphics*, 2012.

[16] Homan Igehy. Tracing ray differentials. *SIGGRAPH*, 1999.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *International Symposium on Computer Architecture*, 2017.

[18] A. S. Kaplanyan, S. Hill, A. Patney, and A. Lefohn. Filtering distributions of normals for shading antialiasing. *HPG*, 2016.

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

[20] Alexandr Kuznetsov, Krishna Mullia, Zexiang Xu, Miloš Hašan, and Ravi Ramamoorthi. NeuMIP: Multi-resolution neural materials. *ACM Trans. Graph.*, 2021.

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

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

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

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

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

[26] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. NeRF in the Wild: Neural radiance fields for unconstrained photo collections. *CVPR*, 2021.

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

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

[29] 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. *SIGGRAPH*, 2019.

[30] 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.

[31] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3D representations without 3d supervision. *CVPR*, 2020.

[32] Marc Olano and Dan Baker. LEAN mapping. *SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010.

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

[34] 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.

[35] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *NeurIPS*, 2007.

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

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

[38] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Niessner, Gordon Wetzstein, and Michael Zollhöfer. DeepVoxels: Learning persistent 3D feature embeddings. *CVPR*, 2019.

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

[40] 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.

[41] 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.

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

[43] Matthew Tancik, Ben Mildenhall, Terrance Wang, Divi Schmidt, Pratul P. Srinivasan, Jonathan T. Barron, and Ren Ng. Learned initializations for optimizing coordinate-based neural representations. *CVPR*, 2021.

[44] 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.

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

[46] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 1980.

[47] Lance Williams. Pyramidal parametrics. *Computer Graphics*, 1983.

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

[49] Lifan Wu, Shuang Zhao, Ling-Qi Yan, and Ravi Ramamoorthi. Accurate appearance preserving prefiltering for rendering displacement-mapped surfaces. *ACM TOG*, 2019.

[50] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *NeurIPS*, 2020.

[51] Richard Zhang. Making convolutional networks shift-invariant again. *ICML*, 2019.

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

[53] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *SIGGRAPH*, 2018.

A. Conical Frustum Integral Derivations

In order to derive formulas for the various moments of the uniform distribution over a conical frustum, we consider an axis-aligned cone parameterized as $(x, y, z) = \varphi(r, t, \theta) = (rt\cos\theta, rt\sin\theta, t)$ for $\theta \in [0, 2\pi)$, $t \geq 0$, $|r| \leq \dot{r}$. This change of variables from Cartesian space gives us a differential term:

$$dx\,dy\,dz = |\det(D\varphi)(r,t,\theta)|dr\,dt\,d\theta \tag{19}$$

$$= \begin{vmatrix} t\cos\theta & t\sin\theta & 0 \\ r\cos\theta & r\sin\theta & 1 \\ -rt\sin\theta & rt\cos\theta & 0 \end{vmatrix} dr\,dt\,d\theta \tag{20}$$

$$= (rt^2\cos^2\theta + rt^2\sin\theta)dr\,dt\,d\theta \tag{21}$$

$$= rt^2 dr\,dt\,d\theta\,. \tag{22}$$

The volume of the conical frustum (which serves as the normalizing constant for the uniform distribution) is:

$$V = \int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} rt^2\,dr\,dt\,d\theta \tag{23}$$

$$= \frac{\dot{r}^2}{2}\cdot\frac{t_1^3 - t_0^3}{3}\cdot 2\pi \tag{24}$$

$$= \pi\dot{r}^2\frac{t_1^3 - t_0^3}{3} \tag{25}$$

Thus the probability density function for points uniformly sampled from the conical frustum is rt^2/V. The first moment of t is:

$$E[t] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} t\cdot rt^2\,dr\,dt\,d\theta \tag{26}$$

$$= \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} rt^3\,dr\,dt\,d\theta \tag{27}$$

$$= \frac{1}{V}\cdot\pi\dot{r}^2\frac{t_1^4 - t_0^4}{4} \tag{28}$$

$$= \frac{3(t_1^4 - t_0^4)}{4(t_1^3 - t_0^3)}\,. \tag{29}$$

The moments of x and y are both zero by symmetry. The second moment of t is

$$E[t^2] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} t^2\cdot rt^2\,dr\,dt\,d\theta \tag{30}$$

$$= \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} rt^4\,dr\,dt\,d\theta \tag{31}$$

$$= \frac{1}{V}\cdot\pi\dot{r}^2\frac{t_1^5 - t_0^5}{5} \tag{32}$$

$$= \frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)}\,. \tag{33}$$

And the second moment of x is:

$$E[x^2] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} (rt\cos\theta)^2\cdot rt^2\,dr\,dt\,d\theta \tag{34}$$

$$= \frac{1}{V}\int_{t_0}^{t_1}\int_0^{\dot{r}} r^3t^4\int_0^{2\pi}\cos^2\theta\,d\theta\,dr\,dt \tag{35}$$

$$= \frac{1}{V}\cdot\frac{\dot{r}^4}{4}\cdot\frac{t_1^5 - t_0^5}{5}\cdot\pi \tag{36}$$

$$= \frac{\dot{r}^2}{4}\cdot\frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)}\,. \tag{37}$$

The second moment of y is the same by symmetry. All cross terms in the covariance are z, also by symmetry.

With these moments defined, we can construct the mean and covariance for a random point within our conical frustum. The mean along the ray direction μ_t is simply the first moment with respect to t:

$$\mu_t = \frac{3(t_1^4 - t_0^4)}{4(t_1^3 - t_0^3)}\,. \tag{38}$$

The variance of the conical frustum with respect to t follows from the definition of variance as $\mathrm{Var}(t) = \mathrm{E}[t^2] - \mathrm{E}[t]^2$:

$$\sigma_t^2 = \frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)} - \mu_t^2\,. \tag{39}$$

The variance of the conical frustum with respect to its radius r is equal to the variance of the frustum with respect to x or (by symmetry) y. Since the first moment with respect to x is zero, the variance is equal to the second moment:

$$\sigma_r^2 = \dot{r}^2\left(\frac{3(t_1^5 - t_0^5)}{20(t_1^3 - t_0^3)}\right). \tag{40}$$

Computing all three of these quantities in their given form is numerically unstable — the ratio of the differences between t_1 and t_0 raised to large powers is difficult to compute accurately when t_0 and t_1 are near each other, which occurs frequently during training. Using these quantities in practice often produces 0 or NaN instead of accurate values, which causes training to fail. We therefore reparameterize these equations as a function of the center and spread of t_0 and t_1: $t_\mu = (t_0 + t_1)/2$, $t_\delta = (t_1 - t_0)/2$. This allows us to rewrite each mean and variance as a first-order term that is then corrected by higher-order terms, which are scaled by t_δ. This gives us stable and accurate values even when t_δ is small. Our reparameterized values are:

$$\mu_t = t_\mu + \frac{2t_\mu t_\delta^2}{3t_\mu^2 + t_\delta^2}\,, \qquad \sigma_t^2 = \frac{t_\delta^2}{3} - \frac{4t_\delta^4(12t_\mu^2 - t_\delta^2)}{15(3t_\mu^2 + t_\delta^2)^2}\,,$$

$$\sigma_r^2 = \dot{r}^2\left(\frac{t_\mu^2}{4} + \frac{5t_\delta^2}{12} - \frac{4t_\delta^4}{15(3t_\mu^2 + t_\delta^2)}\right). \tag{41}$$

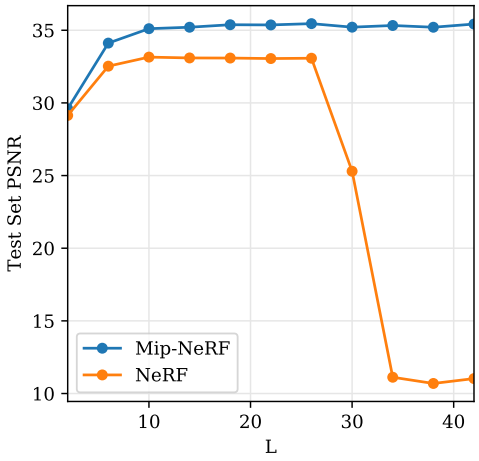

Figure 7: PSNRs for NeRF and mip-NeRF on the test set of the *lego* scene, as we vary the positional encoding degree L. In NeRF, performance decreases due to overfitting for large values of L, but in mip-NeRF this parameter is effectively removed from tuning — it can just be set to a large value and forgotten, because IPE features "tune" their own frequencies automatically.

Note that our multivariate Gaussian approximation of a conical frustum will be inaccurate if there is a significant difference between the base and top radii of the frustum, which will be true for frustums that are very near the camera's center of projection when the camera FOV is large. This is highly uncommon in most datasets, but may be an issue if one were to use mip-NeRF in unusual circumstances, such as macro photography with a fisheye lens.

B. The L Hyperparameter in PE and IPE

IPE features can be viewed as a generalization of PE features: $\gamma(\mathbf{x}) = \gamma(\boldsymbol{\mu} = \mathbf{x}, \boldsymbol{\Sigma} = \mathbf{0})$. Or more rigorously, PE features can be thought of as "hard" IPE features in which all points are assumed to have identical isotropic covariance matrices whose variance has been heuristically determined by the L hyperparameter: the value of L determines the frequency at which PE features are truncated, just as the Gaussian function of variance in IPE serves as a "soft" truncation of IPE features. Because the "soft" maximum frequency of IPE features is determined entirely by the geometry and intrinsics of the camera, IPE features do not depend on the L hyperparameter, and so using IPE features removes the need for tuning L. This is because in PE the L parameter determines where the high frequencies in the PE are truncated, but in IPE those high frequencies are naturally attenuated by the size of the multivariate Gaussian used as input to the encoding: the smaller the Gaussian, the more high frequencies will be retained. To demonstrate this, we performed as "sweep" of L in both mip-NeRF and NeRF, and report the test-set PSNR for a single scene, which is visualized in Figure 7. We see that in NeRF, there is a range of values for L in which performance is maximized, but values that are too large or too small will hurt performance. But in mip-NeRF, we see that L can be set to an arbitrarily large value and performance is unaffected. In practice, in all mip-NeRF experiments in the paper we set $L = 16$, which is a value that results in the last dimension of all IPE features constructed during training to be less than numerical epsilon.

C. Hyperparameters

In all experiments in the paper we take care to use exactly the same set of hyperparameters that were used in Mildenhall *et al.* [30], so as to isolate the specific contributions of mip-NeRF as they relate to cone-casting and IPE features. The three relevant hyperparameters that govern mip-NeRF's behavior are: 1) the number of samples N drawn at each of the two levels ($N = 128$), 2) the histogram "padding" hyperparameter α on the coarse transmittance weights that are used to sample the fine t values ($\alpha = 0.01$), and 3) the multiplier λ on the "coarse" component of the loss function ($\lambda = 0.1$). And though mip-NeRF adds these three hyperparameters, it also deprecates three NeRF hyperparameters that are no longer used: 1) The number of samples N_c drawn for the "coarse" MLP ($N_c = 64$), 2) The number of samples N_f drawn for the "fine" MLP ($N_f = 128$), and 3) The degree L used for the spatial positional encoding ($L = 10$). The α parameter used by mip-NeRF serves a similar purpose as the balance between N_c and N_f did in NeRF — a larger value of α biases the final samples used during rendering towards a uniform distribution, just as a larger value of N_c biases the final samples (which are the sorted union of the uniform coarse samples and the biased fine samples) towards a uniform distribution. Mip-NeRF's multiplier λ has no analog in NeRF, as NeRF's usage of two distinct MLPs means that the "coarse" and "fine" losses in NeRF do not need to be balanced — thankfully, though mip-NeRF adds the need to tune this new hyperparameter λ, it simultaneously removes the need to tune the L hyperparameter as discussed in Section B, so the total number of hyperparameters that require tuning remains constant across the two models.

Before running the experiments in the paper, we briefly tuned the α and λ hyperparameters by hand on the validation set of the *lego* scene. N was not tuned, and was just set to 128 such that the total number of MLP evaluations used by mip-NeRF matched the total number used by NeRF.

D. Forward-Facing Scenes

Note that this paper does not evaluate on the LLFF dataset [29], which consists of scenes captured by a "forward-facing" handheld cellphone camera. For these

scenes, NeRF trained and evaluated models in a "normalized device coordinates" (NDC) space. NDC coordinates work by nonlinearly warping a frustum-shaped space into a unit cube, which sidesteps some otherwise challenging design decisions (such as how an unbounded 3D space should be represented using positional encoding). NDC coordinates can only be used for these "forward-facing" scenes; in scenes where the camera rotates significantly (which is the case for the vast majority of 3D datasets) NeRF uses conventional 3D "world coordinates". One interesting consequence of NDC space is that the 3D volume corresponding to a pixel is *not* a frustum, but is instead a rectangle — in NDC the spatial support of a pixel in the xy plane *does not* increase with the distance from the image plane, as it would in conventional projective geometry.

We briefly experimented with a variant of mip-NeRF that works in NDC space by casting *cylinders* instead of cones. The average PSNR achieved by JaxNeRF on this task is 26.843, and this cylinder-casting variant of mip-NeRF achieves an average PSNR of 26.838. Because this mip-NeRF variant roughly matches the accuracy of NeRF, the only substantial benefit it appears to provide is removing the need to tune the L parameter in positional encoding. This result provides some insight into why NeRF works so well on forward-facing scenes: in NDC space there is little difference between NeRF's "incorrect" aliased approach of casting rays and tuning the L hyperparameter (which as discussed in Section B, is approximately equivalent to using IPE features with isotropic Gaussians) and the more "correct" anti-aliased approach of mip-NeRF. In essence, NeRF is already able to get most of the benefit provided by cone-casting and IPE features in NDC space, because in NDC space NeRF's aliased model is already very similar to mip-NeRF's approach. This interplay between scene parameterization and anti-aliasing suggests that a signal processing analysis of coordinate spaces in neural rendering problems may provide additional unexpected benefits or insights.

E. Model Details

The primary contributions of this paper are the use of cone tracing, integrated positional encoding features, and our use of a single unified multiscale model (as opposed to NeRF's separate per-scale models), which together allow mip-NeRF to better handle multiscale data and reduce aliasing. Additionally, mip-NeRF includes a small number of changes that do not meaningfully change mip-NeRF's accuracy or speed, but slightly simplify our method and increase its robustness during optimization. These "miscellaneous" changes, as noted by the "w/o Misc." ablation in the main paper, do not significantly affect mip-NeRF's performance, but are described here in full for the sake of reproducibility with the hopes that future work will find them useful.

E.1. Identity Concatenation

In the original NeRF paper, the input to the MLP is not just the positional encoding of the position and view direction, but is instead the concatenation of the positional encoding with the position and view direction being encoded. We found this "identity" encoding to not contribute meaningfully to performance or speed, and its presence makes the formalization of our IPE features somewhat challenging, so this in mip-NeRF this identity mapping is removed and the only input to the MLP is the integrated positional encoding itself.

E.2. Activation Functions

In the original NeRF paper, the activation functions used by the MLP to construct the predicted density τ and color $\mathbf{c}$ are a ReLU and a sigmoid, respectively. Instead of a ReLU as the activation function to produce τ, we use a shifted softplus: $\log(1 + \exp(x - 1))$. We found that using a softplus yielded a smoother optimization problem that is less prone to catastrophic failure modes in which the MLP emits negative values everywhere (in which case all gradients from τ are zero and optimization will fail). The shift by -1 within the softplus is equivalent to initializing the biases that produce τ in mip-NeRF to -1, and this causes initial τ values to be small. Initializing the density of the NeRF to small values results in slightly faster optimization at the beginning of training, as dense scene content causes gradients from scene content "behind" that dense content to be suppressed. Instead of a sigmoid to produce color $\mathbf{c}$, we use a "widened" sigmoid that saturates slightly outside of $[0, 1]$ (the range of input RGB intensities): $(1 + 2\epsilon)/(1 + \exp(-x)) - \epsilon$, with $\epsilon = 0.001$. This avoids an uncommon failure mode in which training tries to explain away a black or white pixel by saturating network activations into the tails of the sigmoid where the gradient is zero, which may cause optimization to fail. By having the network saturate at values slightly outside of the range of input values, activations are never encouraged to saturate. These changes to activation functions have little effect on performance, but we found that they improved training stability when using large learning rates (though all results in this paper use the same lower learning rates used by Mildenhall *et al*. [30] for fair comparison).

E.3. Optimization

In all experiments we train mip-NeRF and JaxNeRF using the default training procedure specified in the JaxNeRF codebase: 1 million iterations of Adam [19] with a batch size of 4096 and a learning rate that is annealed logarithmically from $\eta_0 = 5 \cdot 10^{-4}$ to $\eta_n = 5 \cdot 10^{-6}$. We additionally "warm up" the learning rate using the functionality provided by JaxNeRF, which does not improve the performance of mip-NeRF itself, but which we found to improve the stability of some of the mip-NeRF ablations. To allow

	PSNR ↑				SSIM ↑				LPIPS ↓					Train Time	Test Time	
	Full Res.	1/2 Res.	1/4 Res.	1/8 Res.	Full Res.	1/2 Res.	1/4 Res.	1/8 Res.	Full Res.	1/2 Res.	1/4 Res.	1/8 Res	Avg. ↓	(hours)	(sec/MP)	# Params
NeRF + Area, Center, 1× SS	27.471	28.016	27.816	26.657	0.9187	0.9301	0.9365	0.9304	0.1064	0.0924	0.0934	0.1064	0.0362	2.85	2.61	1,191K
NeRF + Area, Center, 4× SS	28.424	29.420	29.863	29.233	0.9297	0.9426	0.9526	0.9547	0.0807	0.0598	0.0530	0.0536	0.0259	17.69	10.44	1,191K
NeRF + Area, Center, 16× SS	31.566	33.116	33.982	32.933	0.9524	0.9660	0.9753	0.9768	0.0537	0.0316	0.0227	0.0216	0.0144	37.18	41.76	1,191K
Mip-NeRF	32.629	34.336	35.471	35.602	0.9579	0.9703	0.9786	0.9833	0.0469	0.0260	0.0168	0.0120	0.0114	2.79	2.48	612K

Table 4: Here we evaluate mip-NeRF against an extension of NeRF in which brute-force supersampling with jittered rays is used during training and evaluation, on our multiscale Blender dataset ("16× SS" indicates that 16 rays are cast per pixel, etc). Mip-NeRF is able to outperform this baseline by a significant margin in terms of quality, while also being 13× faster to train and 16× faster to evaluate.

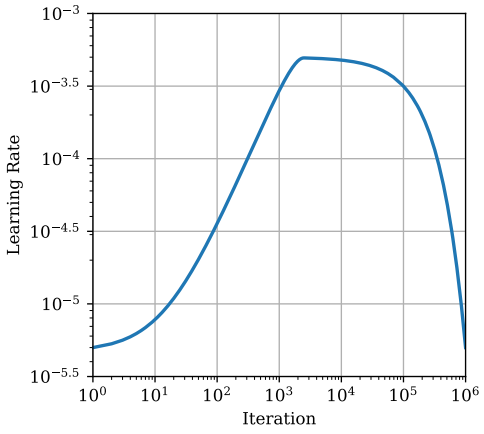

Figure 8: The learning rate used in all JaxNeRF and mip-NeRF experiments.

our ablations to be competitive, and to enable a fair comparison across all models, we therefore use this warm up strategy in all mip-NeRF and JaxNeRF experiments. Because the warm up procedure in JaxNeRF is not described in its documentation [11], for the sake of reproducibility we will describe it here. For the first $n_w = 2500$ iterations of optimization, we scale the basic learning rate by an additional scale factor that is smoothly annealed between $\lambda_w = 0.01$ and 1 during this warm up period. The learning rate at iteration i during training is:

$$\begin{aligned}\eta_i =&(\lambda_w + (1-\lambda_w)\sin((\pi/2)\,\mathrm{clip}(i/n_w, 0, 1))) \\ &\times (\exp((1-i/n)\log(\eta_0) + (i/n)\log(\eta_n)))\end{aligned} \tag{42}$$

See Figure 8 for a visualization.

E.4. View Dependent Effects

We handle viewing directions exactly as was done in NeRF: the ray direction $\mathbf{d}$ is normalized, positionally encoded ($L = 4$), and injected into the last layer of the MLP after τ is predicted but before $\mathbf{c}$ is predicted. This is omitted from our notation in the main paper for simplicity's sake. see Mildenhall *et al*. for details [30].

F. Supersampling Baseline

In the main paper we presented a generous baseline approach in which NeRF is trained on only full-resolution images (thereby sidestepping its poor performance when trained on multi-resolution data) and then evaluated on our multiscale Blender dataset by brute-force supersampling: rendering a full-resolution image that is then downsampled to match the resolution of the ground truth. This roughly matches the performance of mip-NeRF, but is 22× slower and relies on "oracle" scale information that does not exist for most datasets. Here we explore an alternative supersampling baseline, in which we train an extension of NeRF on the multiscale dataset while supersampling during *both* training and evaluation: for every pixel we cast multiple jittered rays (sampled uniformly at random) through the spatial footprint of each pixel, render each ray with the NeRF, and then use the mean of those rendered values as the predicted color of that pixel in the loss function. As shown by the results of this experiment (Table 4) this brute-force supersampling model not only performs worse than mip-NeRF even when casting as many as 16 rays per pixel, but is also significantly more expensive during both training and evaluation.

G. Alternative Gaussian Positional Encoding

During experimentation we explored alternative approaches for featurizing the mean and covariance matrix of the multivariate Gaussians used by mip-NeRF. One such alternative strategy is to simply apply positional encoding to the mean and to the (signed) square root of the elements of the covariance matrix, and use the concatenation of the two as input. Specifically, we compute the positional encoding of $\boldsymbol{\mu}$ with $L = 12$, and compute the positional encoding of $\mathrm{vec}(\mathrm{triu}(\mathrm{sign}(\boldsymbol{\Sigma}) \circ \sqrt{|\boldsymbol{\Sigma}|}))$ with $L = 2$. We found that this approach performs comparably to the IPE features presented in the main paper, as shown in Table 5. We chose to advocate for IPE features in the main paper instead of this concatenation alternative because 1) IPE features are more compact (thereby reducing model size and evaluation time), 2) IPE features are easy to justify and reason about (as they approximate an expectation of positional encoding features with respect to a conical frustum), and 3) IPE features have

Multiscale Blender	PSNR ↑	SSIM ↑	LPIPS ↓	Avg. ↓
Integrated PE	**34.51**	**0.973**	**0.025**	**0.0113**
Concatenated PE	34.40	**0.973**	**0.025**	0.0114

Blender	PSNR ↑	SSIM ↑	LPIPS ↓	Avg. ↓
Integrated PE	**33.09**	**0.961**	0.043	0.0161
Concatenated PE	**33.09**	**0.961**	**0.042**	**0.0160**

Table 5: An evaluation of the IPE features against an alternative approach in which the mean and covariance of the multivariate Gaussian corresponding to a conical frustum are positionally encoded and concatenated. Both approaches perform comparably on the multiscale and single-scale Blender datasets.

no hyperparameters (while this concatenation alternative is sensitive to its two L hyperparameters and the design decisions used when parameterizing $\mathbf{\Sigma}$).

This experiment with using this alternative to IPE also provides some insight into the inner workings of mip-NeRF. While IPE features are insensitive to the off-diagonal elements of $\mathbf{\Sigma}$, this concatenation alternative should endow the MLP with the ability to reason about the correlation of dimensions of the multivariate Gaussian. The fact that this ability does not improve accuracy may suggest that correlation is not a helpful cue, which contradicted the intuition of the authors. Additionally, this experiment reinforces the assertions made in the paper that the reason for mip-NeRF's improved performance is its explicit modeling of conical frustums, as opposed to NeRF's usage of point samples along a ray. Though it is critical that the geometry of image formation be modeled accurately, there are likely many effective ways to featurize that geometry.

H. Additional Results

Multiscale Blender Dataset. To demonstrate the relative accuracy of mip-NeRF compared to NeRF on each individual scene in the multiscale Blender dataset, the error metrics for each individual scene are provided in Table 6. Mip-NeRF yields a significant reduction in error compared to NeRF across all scenes. Renderings produced by mip-NeRF and baseline algorithms compared to the ground truth can be visually inspected in Figures 9 and 10.

Blender Dataset. Test-set error metrics for each individual scene in the (single scale) Blender dataset of Mildenhall *et al*. [30] can be seen in Table 7. Mip-NeRF yields lower error rates than NeRF on all scenes and all metrics.

	Average PSNR							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
NeRF (Jax Implementation) [11, 30]	29.923	23.273	27.153	32.001	27.748	26.295	28.401	26.462
NeRF + Area Loss	30.277	24.032	27.149	32.025	27.602	26.533	28.120	26.834
NeRF + Area, Centered Pixels	33.460	25.802	30.400	35.672	31.606	30.155	32.633	30.019
NeRF + Area, Center, Misc.	33.394	25.874	30.369	35.641	31.646	30.184	32.601	30.092
Mip-NeRF	37.141	27.021	33.188	39.313	35.736	32.558	38.036	33.083
Mip-NeRF w/o Misc.	37.275	26.979	33.160	39.357	35.749	32.563	37.997	33.078
Mip-NeRF w/o Single MLP	37.310	26.922	33.045	39.378	35.605	32.635	38.016	33.011
Mip-NeRF w/o Area Loss	35.188	26.063	32.542	37.165	34.319	31.004	35.922	31.636
Mip-NeRF w/o IPE	33.559	25.864	30.499	35.793	31.728	30.272	32.736	30.276

	Average SSIM							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
NeRF (Jax Implementation) [11, 30]	0.9436	0.8908	0.9423	0.9586	0.9256	0.9335	0.9580	0.8607
NeRF + Area Loss	0.9488	0.9028	0.9429	0.9622	0.9274	0.9372	0.9592	0.8610
NeRF + Area, Centered Pixels	0.9710	0.9310	0.9705	0.9794	0.9643	0.9670	0.9800	0.8994
NeRF + Area, Center, Misc.	0.9707	0.9318	0.9705	0.9793	0.9646	0.9671	0.9799	0.9004
Mip-NeRF	0.9875	0.9450	0.9836	0.9880	0.9843	0.9767	0.9928	0.9221
Mip-NeRF w/o Misc.	0.9877	0.9448	0.9835	0.9880	0.9842	0.9767	0.9927	0.9227
Mip-NeRF w/o Single MLP	0.9875	0.9432	0.9829	0.9876	0.9836	0.9763	0.9922	0.9211
Mip-NeRF w/o Area Loss	0.9817	0.9371	0.9823	0.9849	0.9792	0.9731	0.9911	0.9175
Mip-NeRF w/o IPE	0.9714	0.9322	0.9713	0.9796	0.9658	0.9678	0.9804	0.9039

	Average LPIPS							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
NeRF (Jax Implementation) [11, 30]	0.0347	0.0689	0.0324	0.0279	0.0410	0.0452	0.0307	0.0948
NeRF + Area Loss	0.0414	0.0762	0.0438	0.0365	0.0568	0.0499	0.0444	0.1139
NeRF + Area, Centered Pixels	0.0281	0.0593	0.0264	0.0240	0.0348	0.0330	0.0249	0.0865
NeRF + Area, Center, Misc.	0.0283	0.0586	0.0264	0.0241	0.0346	0.0330	0.0249	0.0850
Mip-NeRF	0.0111	0.0439	0.0135	0.0121	0.0127	0.0186	0.0065	0.0624
Mip-NeRF w/o Misc.	0.0111	0.0436	0.0136	0.0123	0.0127	0.0186	0.0066	0.0620
Mip-NeRF w/o Single MLP	0.0113	0.0443	0.0142	0.0122	0.0132	0.0187	0.0068	0.0628
Mip-NeRF w/o Area Loss	0.0171	0.0503	0.0146	0.0151	0.0163	0.0259	0.0095	0.0665
Mip-NeRF w/o IPE	0.0276	0.0578	0.0259	0.0240	0.0340	0.0320	0.0231	0.0829

Table 6: Per-scene results on the test set images of the multiscale Blender dataset presented in this work. We report the arithmetic mean of each metric averaged over the four scales used in the dataset.

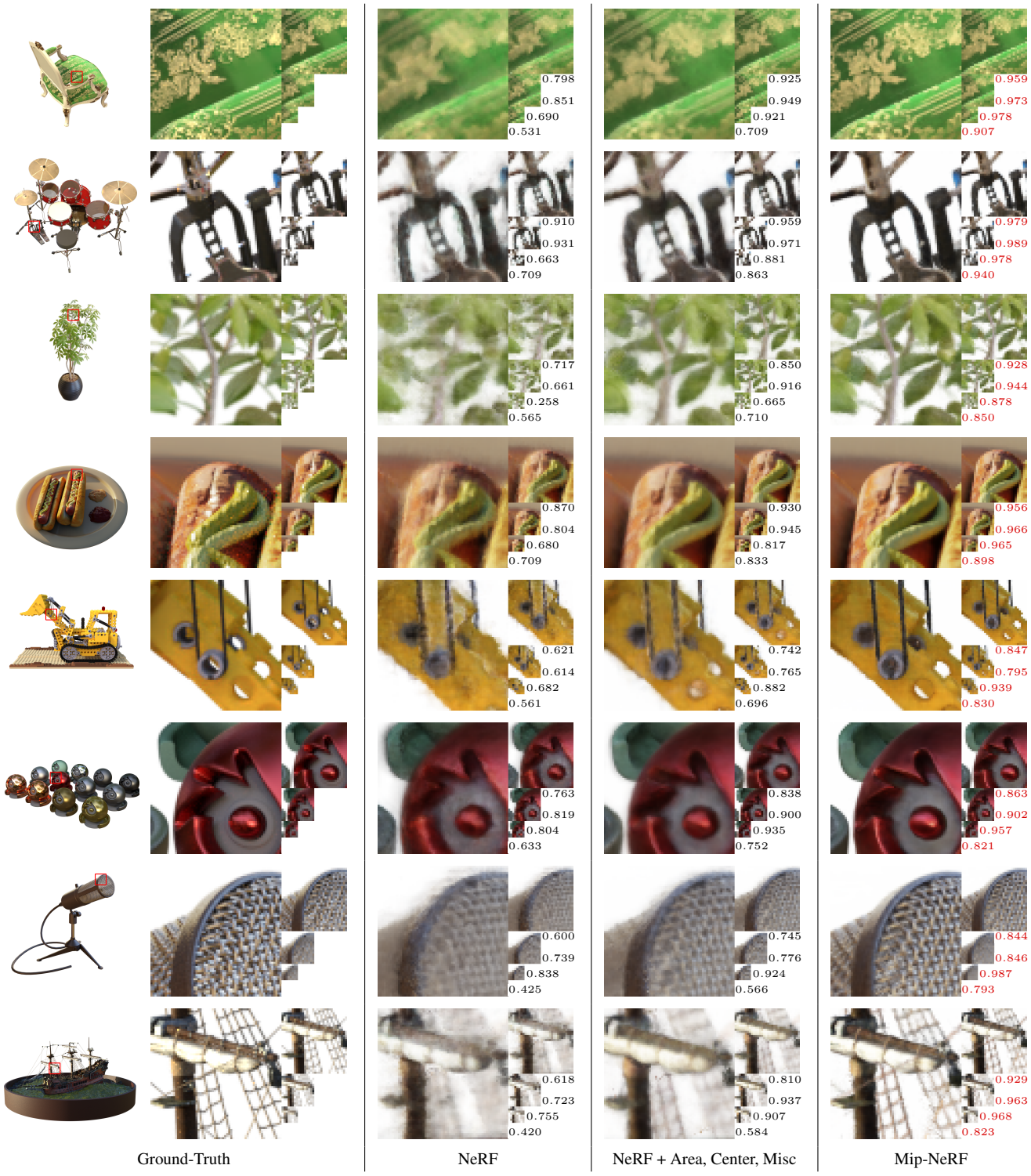

Figure 9: Visualizations of the output renderings from mip-NeRF compared to the ground truth, NeRF, and our improved version of NeRF, on test set images from the 8 scenes in our multiscale Blender dataset. We visualize a cropped region of each scene for better visualization, and render out that scene at 4 different resolutions, displayed as an image pyramid. The SSIM for each scale of each image pyramid truth is shown to its lower right, with the highest SSIM for each algorithm at each scale highlighted in red.

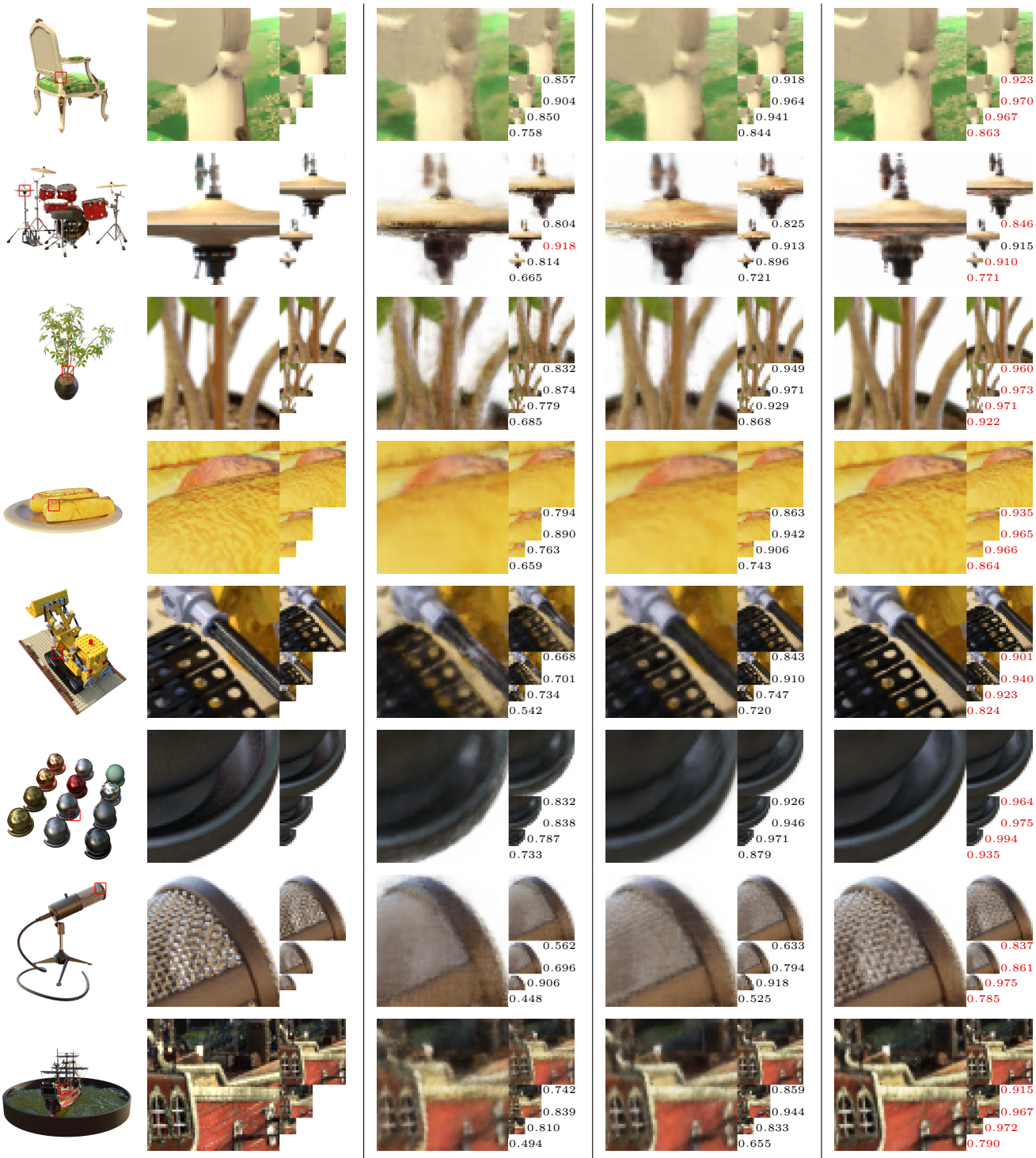

Figure 10: Additional visualizations of the output renderings from mip-NeRF compared to the ground truth, NeRF, and an improved version of NeRF presented in this work, on test set images from the 8 scenes in our multiscale Blender dataset, in the same format as Figure 9.

	PSNR							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
SRN [39]	26.96	17.18	20.73	26.81	20.85	18.09	26.85	20.60
Neural Volumes [25]	28.33	22.58	24.79	30.71	26.08	24.22	27.78	23.93
LLFF [29]	28.72	21.13	21.79	31.41	24.54	20.72	27.48	23.22
NSVF [24]	33.19	25.18	31.23	37.14	32.29	32.68	34.27	27.93
NeRF (TF Implementation) [30]	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65
NeRF (Jax Implementation) [11, 30]	34.17	25.08	30.39	36.82	33.31	30.03	34.78	29.30
NeRF + Centered Pixels	34.88	25.17	31.02	37.13	34.39	30.50	35.38	29.95
NeRF + Center, Misc.	34.94	25.19	31.05	37.15	34.12	30.47	35.33	29.95
Mip-NeRF	35.14	25.48	33.29	37.48	35.70	30.71	36.51	30.41
Mip-NeRF w/o Single MLP	35.07	25.28	32.52	37.34	34.93	30.38	35.59	30.55
Mip-NeRF w/o Misc.	35.16	25.46	32.96	37.55	35.68	30.69	36.32	30.47
Mip-NeRF w/o IPE	35.10	25.23	31.30	37.17	34.89	30.56	35.75	29.85
Mip-NeRF, Stopped Early	34.21	25.23	30.79	36.89	33.72	29.86	35.02	29.44

	SSIM							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
SRN [39]	0.910	0.766	0.849	0.923	0.809	0.808	0.947	0.757
Neural Volumes [25]	0.916	0.873	0.910	0.944	0.880	0.888	0.946	0.784
LLFF [29]	0.948	0.890	0.896	0.965	0.911	0.890	0.964	0.823
NSVF [24]	0.968	0.931	0.973	0.980	0.960	0.973	0.987	0.854
NeRF (TF Implementation) [30]	0.967	0.925	0.964	0.974	0.961	0.949	0.980	0.856
NeRF (Jax Implementation) [11, 30]	0.975	0.925	0.967	0.979	0.968	0.953	0.987	0.869
NeRF + Centered Pixels	0.979	0.928	0.971	0.980	0.973	0.956	0.989	0.877
NeRF + Center, Misc.	0.979	0.927	0.971	0.980	0.972	0.956	0.989	0.877
Mip-NeRF	0.981	0.932	0.980	0.982	0.978	0.959	0.991	0.882
Mip-NeRF w/o Single MLP	0.980	0.929	0.977	0.981	0.976	0.956	0.989	0.883
Mip-NeRF w/o Misc.	0.981	0.932	0.979	0.982	0.978	0.959	0.991	0.883
Mip-NeRF w/o IPE	0.981	0.929	0.972	0.981	0.975	0.958	0.990	0.878
Mip-NeRF, Stopped Early	0.976	0.927	0.969	0.979	0.969	0.954	0.988	0.869

	LPIPS							
	chair	*drums*	*ficus*	*hotdog*	*lego*	*materials*	*mic*	*ship*
SRN [39]	0.106	0.267	0.149	0.100	0.200	0.174	0.063	0.299
Neural Volumes [25]	0.109	0.214	0.162	0.109	0.175	0.130	0.107	0.276
LLFF [29]	0.064	0.126	0.130	0.061	0.110	0.117	0.084	0.218
NSVF [24]	0.043	0.069	0.017	0.025	0.029	0.021	0.010	0.162
NeRF (TF Implementation) [30]	0.046	0.091	0.044	0.121	0.050	0.063	0.028	0.206
NeRF (Jax Implementation) [11, 30]	0.026	0.071	0.032	0.030	0.031	0.047	0.012	0.150
NeRF + Centered Pixels	0.022	0.069	0.028	0.028	0.026	0.043	0.010	0.143
NeRF + Center, Misc.	0.022	0.069	0.028	0.028	0.027	0.044	0.011	0.142
Mip-NeRF	0.021	0.065	0.020	0.027	0.021	0.040	0.009	0.138
Mip-NeRF w/o Single MLP	0.022	0.067	0.023	0.028	0.024	0.044	0.011	0.135
Mip-NeRF w/o Misc.	0.021	0.066	0.022	0.026	0.021	0.040	0.009	0.136
Mip-NeRF w/o IPE	0.020	0.068	0.027	0.028	0.024	0.041	0.009	0.142
Mip-NeRF, Stopped Early	0.027	0.074	0.035	0.031	0.035	0.046	0.013	0.155

Table 7: Per-scene results on the test set images of the (single-scale) Blender dataset from Mildenhall *et al*. [30]