DIVeR: Real-time and Accurate Neural Radiance Fields with Deterministic Integration for Volume Rendering

Liwen Wu Jae Yong Lee Anand Bhattad Yu-Xiong Wang David Forsyth
University of Illinois at Urbana-Champaign
{liwenwu2, lee896, bhattad2, yxw, daf}@illinois.edu

Abstract

DIVeR builds on the key ideas of NeRF and its variants – density models and volume rendering – to learn 3D object models that can be rendered realistically from small numbers of images. In contrast to all previous NeRF methods, DIVeR uses deterministic rather than stochastic estimates of the volume rendering integral. DIVeR's representation is a voxel based field of features. To compute the volume rendering integral, a ray is broken into intervals, one per voxel; components of the volume rendering integral are estimated from the features for each interval using an MLP, and the components are aggregated. As a result, DIVeR can render thin translucent structures that are missed by other integrators. Furthermore, DIVeR's representation has semantics that is relatively exposed compared to other such methods – moving feature vectors around in the voxel space results in natural edits. Extensive qualitative and quantitative comparisons to current state-of-the-art methods show that DIVeR produces models that (1) render at or above state-of-the-art quality, (2) are very small without being baked, (3) render very fast without being baked, and (4) can be edited in natural ways. Our real-time code is available at: `https://github.com/lwwu2/diver-rt`

1. Introduction

Turning a small set of images into a renderable model of a scene is an important step in scene generation, appearance modeling, relighting, and computational photography. The task is well-established and widely studied; what form the model should take is still very much open, with models ranging from explicit representations of geometry and material through plenoptic function models [1]. Plenoptic functions are hard to smooth, but neural radiance field (NeRF) [25] demonstrates that a Multi Layer Perceptron (MLP) with positional encoding is an exceptionally good smoother, resulting in an explosion of variants (details in related work). All use one key trick: the scene is modeled as density and color functions, rendered using stochastic estimates of volume rendering integrals. We describe an alternative approach, *deterministic integration for volume rendering* (DIVeR), which is competitive in speed and accuracy with the state of the art.

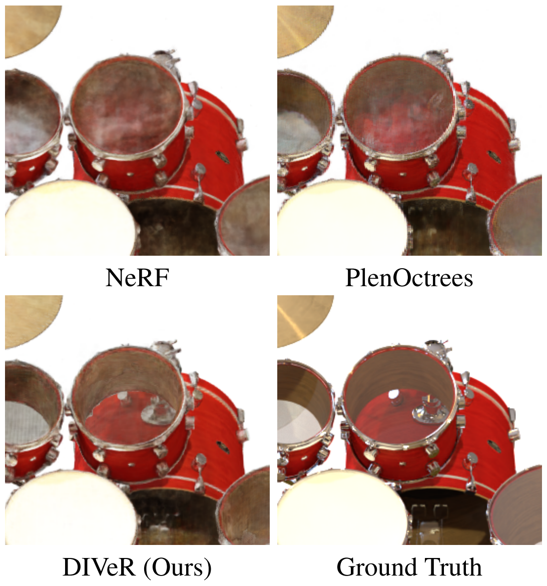


Figure 1. **Monte Carlo vs. feature integration**. Methods like NeRF [25] and PlenOctrees [55] that use Monte Carlo (stochastic) integrators fail to render the translucent drumhead; it is thin, and so is hit by few samples. NeRF's estimator will not model it with practical numbers of samples. In contrast, our method uses a deterministic integrator which directly estimates the section of volume rendering integral near the drum plane through feature integration (Sec. 4.2); this uses fewer calls to the integrator and still successfully models the transparency.

We use a deterministic integrator because stochastic estimates of integrals present problems. Samples may miss important effects (Fig. 1). Fixing this by increasing the sampling rate is costly: accuracy improves slowly in the number of samples N, (for Monte Carlo methods, standard deviation goes as $1/\sqrt{N}$ [4]), but the cost of computation grows linearly. In contrast, our integrator combines per-voxel estimates of the volume rendering integral into a single estimate using alpha blending (Sec. 4.2).

Like NSVF [21], we use a voxel based representation of the color and density. Rather than represent functions, we provide a feature vector at each voxel vertex. The feature vectors at the vertices of a given voxel are used by an MLP to compute the deterministic integrator estimate for the section of any ray passing through the voxel. Mainly, a model is learned by gradient descent on feature vectors and MLP parameters to minimize the prediction error for the training views; it is rendered by querying the resulting structure with new rays. Sec. 4 provides the details.

Given similar computational resources, our model is efficient to train, likely because the deterministic integration can fit the integral better, and there is no gradient noise produced by stochastic integral estimates. As Sec. 5 shows, the procedure results in very small models ($\sim$ 64MB) which render very fast ($\sim$ 50 FPS on a single 1080 Ti GPU) and have comparable PSNR with the best NeRF models.

2. Background

NeRF [25] represents 3D scenes with a density field $\sigma(\mathbf{x})$ and a color field $\mathbf{c}(\mathbf{x}, \mathbf{d})$ which are functions of 3D position $\mathbf{x}$ and view direction $\mathbf{d}$ encoded by an MLP with weights $\mathbf{w}$. To render a pixel, a ray $\mathbf{r}(t) = \mathbf{o} + \mathbf{d}t$ is shot from the camera center $\mathbf{o}$ through the pixel center in direction $\mathbf{d}$ and follows the volume rendering equation [14] to accumulate the radiance:

$$\hat{\mathbf{c}}(\mathbf{r}) = \int_0^\infty e^{-\int_0^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt. \quad (1)$$

Closed-form solutions for Eq. 1 are not available, so NeRF uses Monte Carlo integration by randomly sampling n points $\mathbf{x}_i = \mathbf{r}(t_i), i = 1, \dots, n$ along the ray from eye to far with their radiance and density values $(\mathbf{c}_i, \sigma_i) = \text{MLP}_{\mathbf{w}}(\mathbf{x}_i, \mathbf{d}_i)$. The radiance and density function is then treated as constant in every interval $[t_{i-1}, t_i]$, and an approximation of the volume rendering equation is given as:

$$\hat{\mathbf{c}}(\mathbf{r}) = \sum_{i=1}^{n} \prod_{j=1}^{i-1} (1 - \alpha_j)\alpha_i \mathbf{c}_i \quad (2)$$

$$\alpha_i = 1 - e^{-\sigma_i \delta_i}, \quad (3)$$

where α_i denotes the accumulated alpha values along the interval and $\delta_i = \|\mathbf{x}_{i+1} - \mathbf{x}_i\|_2$ is the interval length. During training, NeRF learns to adjust the density and color fields to produce training images, which is achieved by optimizing weights $\mathbf{w}$ in respect to squared error between rendered pixel and its ground truth:

$$L = \sum_k \|\hat{\mathbf{c}}(\mathbf{r}_k) - \hat{\mathbf{c}}_{\text{gt}}(\mathbf{r}_k)\|_2^2. \quad (4)$$

3. Related Work

Novel view synthesis: All scene modeling methods attempt to exploit regularities in the plenoptic function (the spectral radiance traveling in any direction at any point) of a scene. One approach is to compute an explicit geometric representations (point clouds [2, 48]; meshes [36, 37, 46]) usually obtained from some 3D reconstruction algorithms (*e.g.* COLMAP [39]). The geometry can then carry deep features, which are projected and then processed by neural networks to get the image. Building an end-to-end optimizable pipeline is hard, however. Alternatively, one could use voxel grids [22, 34, 41], where scene observations are encoded as 3D features and processed by 3D, 2D Convolutional Neural Networks (CNNs) to get the rendered images, yielding cleaner training but a very memory intensive model which is unsuitable for high resolution rendering.

Multi-plane images (MPIs) [24, 44, 51, 59] offer novel views without requiring a precise geometry proxy. One represents a scene as a set of parallel RGBA images and synthesizes a novel view by warping the images to the target view and then alpha blending them; this fails when the view changes are too large. Image based rendering (IBR) approaches [6, 7, 49] render a view by interpolating the nearby observations directly. Most IBR methods generalize well to unseen data, such that a new scene in an IBR model can be rendered off-the-shelf or with a few epochs of fine-tuning.

An alternative is to represent proxies for the plenoptic function using neural networks, and then raycast. [15, 42, 54] use signed distance field like functions, and [28, 38] represent the scene geometry as an occupancy field. NeRF [25] models the plenoptic function using an MLP to encode a density field (of position) and a color field (of position and direction). The radiance at a point in a direction is given by a volume rendering integral [14]. Training is done by adjusting the MLP parameters to produce the right answer for a given set of images. The method can produce photo-realistic rendering on complex scenes, including rendering transparent surfaces and view dependent effects, but takes a long time to train and evaluate.

NeRF has resulted in a rich collection of variants. [3, 43, 58] modify the NeRF to allow control of surface materials and lighting; NeRF-W [23] augments the inputs with image features to help resolve ambiguity between photos in the wild. [29, 30, 32] show how to model deformation, and [10, 11, 18, 19, 52] apply NeRF to 4D videos. Finally, [6, 7, 45, 49, 56] try to improve the generalizability and training speed, and [5, 27, 35, 40, 47] adopt the architecture to generative models.

Rendering NeRF faster: NeRF's stochastic integrator not only misses thin structures (which are hard to find with samples, Figure 1), but also presents efficiency problems. The main strategy for improving the efficiency of NeRF (as in any MC integrator) is coming up with better importance functions [4]. NSVF [21] significantly reduces the number of samples (equivalently, MLP calls; render time) by imposing a voxel grid on the density, and then pruning voxels with empty density at training time. An alternative is a depth oracle that ensures that MLP samples occur only close to points with high density [26]. AutoInt [20] further offers a more efficient estimator for the volume rendering integral by constructing an approximate antiderivative (though the absorption integral must still be approximated), which allows fewer MLP queries at rendering time. In contrast to these

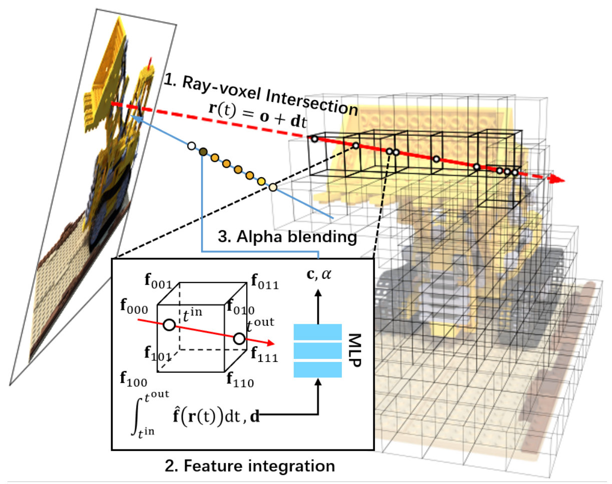


Figure 2. **Rendering pipeline overview of our DIVeR.** To render a ray, we first find its intersection with voxels. For each voxel, features at its eight vertices represent a trilinear function. We integrate this trilinear function from the ray's intersection at the entry to exit, passing the result to an MLP that decodes to color and alpha values for the voxel. We obtain the final integral estimate for the ray by accumulating color and alpha values along the ray.

methods, we use a deterministic integral estimator.

But pure importance based methods cannot render in real-time, because they rely on MLPs that are relatively expensive to evaluate. FastNeRF [12] discretizes continuous fields into bins and caches the bins that have been evaluated for subsequent frames. PlenOctrees [55] and SNeRG [13] pre-bake the results of the NeRF into sparse voxels and use efficient ray marching to achieve interactive frame rate. These methods achieve real-time rendering at a cost of noticeable loss in quality (ours does not), or of requiring a high-resolution voxel grid and so a large storage cost (ours does not). Alternative strategies include: caching MLP calls into MPIs (Nex [51]); and speeding up MLP evaluation by breaking one MLP into many small local specialist MLPs (KiloNeRF [33]). In contrast, we use the representation and MLP obtained at training time.

4. Method

As shown in the overall rendering pipeline (Fig. 2), our DIVeR method differs from the NeRF style models in two important ways: (1) we represent the fields as a voxel grid of feature vectors $\mathbf{f}_{ijk}$, and (2) we use a decoder MLP with learnable weight $\mathbf{w}$ (Fig. 6) to design a deterministic integrator to estimate partial integrals of any fields of the scene. To estimate the volume rendering integral for a particular ray, we decompose it into intervals corresponding to the voxels the ray passes through. We then let each interval report an approximate estimate of the voxel's contribution and accumulate them to the rendering result. The learning of the fields is done by adjusting $\mathbf{f}_{ijk}$ and $\mathbf{w}$ to produce close approximations of the observed images.

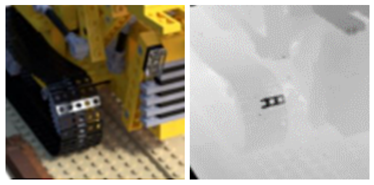


Figure 3. **The simple voxel model overfits:** Training a model with independent $\mathbf{f}_{ijk}$ has a strong tendency to overfit. In this example, the model has interpreted a gloss feature as empty space. Our regularization procedure is explained in the text.

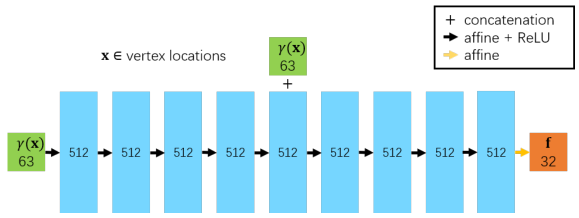


Figure 4. **The implicit MLP** generates correlated vertex features by taking a positional encoding of vertex location and producing a feature vector.

4.1. Voxel based deep implicit fields

As in NSVF [21], the feature vectors are placed at vertices of the voxel grid; feature values inside each voxel are given by the trilinear interpolation of the voxel's eight corners, which yields a piecewise trilinear feature function $\mathbf{f}(\mathbf{x})$. The voxel grid can be thought of as a 3D cache of intermediate sums of NeRF's MLP, which explains why inference should be fast (Sec. 5.3) but still can model complicated spatial behaviors compactly (Sec. 5.2). Because voxels in the empty space make no contribution to the volume rendering, the voxel grid can also be stored in a sparse representation (Sec. 4.5), which further speeds up the rendering and reduces the storage cost (Sec. 5.3).

Initializing voxel features using implicit MLP: If each $\mathbf{f}_{ijk}$ is trained independently and randomly initialized, our representation scheme tends to overfit during the training (Fig. 3). This suggests that the optimization of each $\mathbf{f}_{ijk}$ should be *correlated*, but it is not obvious which correlation strategy should be applied. Instead, we take an MLP that accepts the positional encoded vertex position on the voxel grid to output the feature vector at that position (the implicit MLP, with parameters $\mathbf{w}_r$; see Fig. 4) to correlate each feature vector implicitly. Although an MLP can in principle approximate any function, there is overwhelming experimental evidence that the approximated function tends to be smooth (*e.g.* [51]), which makes it unsuitable for rendering high frequency details. Therefore, we first train the implicit MLP to generate a reasonable initialization of $\mathbf{f}_{ijk}$ placed in the corresponding voxel grid vertex, then discard the regularization MLP and directly optimize on $\mathbf{f}_{ijk}$ explicitly. Experiments show this 'implicit-explicit' strategy prevents overfitting while preserving high-frequency contents.

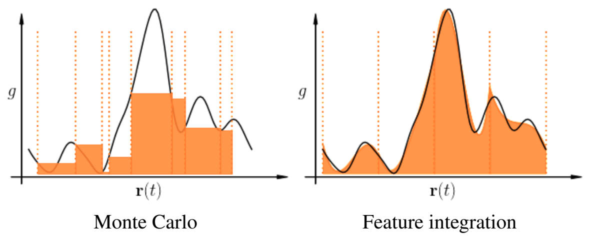


Figure 5. **Integration strategy comparison**. Monte Carlo method fills an interval by a constant; Feature integration fits the interval with trilinear functions that are analytically integratable and blends them using an MLP.

4.2. Feature integration

Intersecting a ray with the voxel grid yields a set of intervals, which are processed separately by our integrator. Write $(t_1^{\text{in}}, t_1^{\text{out}}), \ldots, (t_n^{\text{in}}, t_n^{\text{out}})$ for parameter values defining these intervals, from eye to far end. For interval i, we obtain density σ_i and radiance $\mathbf{c}_i$ by passing the normalized integral of $\mathbf{f}(\mathbf{x})$ along the interval to the MLP. Let $\mathbf{f}_1^i, \ldots, \mathbf{f}_8^i$ be the feature vectors at corners of the voxel the interval passes through and $\chi_1(\mathbf{x}), \ldots, \chi_8(\mathbf{x})$ be the corresponding trilinear interpolation weights, so:

$$(\sigma_i, \mathbf{c}_i) = \text{MLP}_{\mathbf{w}}(\int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \hat{\mathbf{f}}(\mathbf{r}(t))dt, \mathbf{d}) \tag{5}$$

$$\text{where} \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \hat{\mathbf{f}}(\mathbf{r}(t))dt = \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \sum_{k=1}^{8} \mathbf{f}_k^i \frac{\chi_k(\mathbf{r}(t))}{|t_i^{\text{out}} - t_i^{\text{in}}|} dt = \sum_{k=1}^{8} \mathbf{f}_k^i \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \frac{\chi_k(\mathbf{r}(t))}{|t_i^{\text{out}} - t_i^{\text{in}}|} dt. \tag{6}$$

Here $\mathbf{w}$ are the learnable weights of the MLP, and we incorporate viewing direction $\mathbf{d}$ to model the view dependent effect. These approximations are accumulated into a single value of the integral by

$$\hat{\mathbf{c}}(\mathbf{r}) = \sum_{i=1}^{n} \prod_{j=1}^{i-1} (1 - \alpha_j) \alpha_i \mathbf{c}_i \tag{7}$$

$$\alpha_i = 1 - e^{-\sigma_i} \tag{8}$$

$$\sigma_i = \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \sigma(\mathbf{r}(t))dt, \mathbf{c}_i = \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \mathbf{c}(\mathbf{r}(t), \mathbf{d})dt \tag{9}$$

(which is an approximation of Eq. 1, see the supplementary). Notice that, if the MLP had no hidden layers, and the integrand was a known function, we would be adjusting components of a basis function expansion of the integrand to produce the approximation.

Our integrator has two advantages over MC. First, we get a slightly better estimate per interval (the MC estimate assumes fields inside an interval are constant; ours fits them using an MLP; see Fig. 5), and this manifests in better rendering quality (Sec. 5.4). Second, because the integrator is deterministic, the error in integral estimates is deterministic, and so is the gradient, which may help learning; our experience has been that our method has vanishing gradients less often than standard NeRF, and is less sensitive to the choice of learning rate.

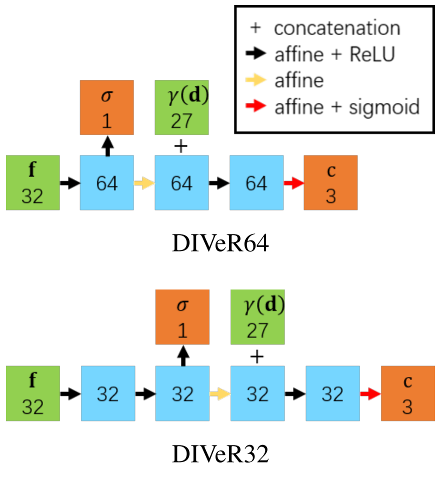


Figure 6. **Decoder architecture**. DIVeR64 has around 8K parameters; DIVeR32 has around 4k parameters. Both MLPs take integrated features and positional encoded viewing direction as inputs and output corresponding integrated density and color.

4.3. Architecture

We choose the feature dimension to be 32, and the voxel grid size varies according to the target image resolution. The grid is relatively coarse and can be represented very efficiently with a sparse representation (Sec. 5.1). As shown in Fig. 6, we investigate two different MLP decoders: DIVeR32 and DIVeR64. Similar to [25], we apply positional encoding to the viewing direction $\mathbf{d}$, but we directly pass the integrated feature $\mathbf{f}$ into the MLP without positional encoding. We use 10 bands for positional encoding in the implicit regularization MLP and 4 bands in the decoder MLP. Because the architectures are tiny, one call of MLP takes less than 1ms, which allows MLP evaluation to happen in real time.

4.4. Training

We optimize $\mathbf{f}_{ijk}$, $\mathbf{w}$, and $\mathbf{w}_r$ for each scene. During a training step, we randomly sample a batch of rays from the training set and follow the procedure described in Sec. 4.2 to render the color, and then apply gradient descent on $\mathbf{f}_{ijk}$ and $\mathbf{w}$ using Eq. 4. We want the voxel grid to be sparse, and so discourage the model from predicting background color in empty space using the regularization loss of [13]:

$$L_{\text{sparsity}} = \lambda_s \sum_i \log(1 + \frac{\sigma_i^2}{0.5}), \tag{10}$$

where σ_i denotes the ith accumulated density; λ_s is the regularization weight. In contrast to NeRF, we do not need hierarchical volume sampling because we use deterministic integration.

Coarse to fine: We speed up training with a coarse to fine procedure. Early in training, it is sufficient to use coarse resolution images to determine whether particular regions are empty using the culling strategy discussed in Sec. 4.5. Based on the coarse occupancy map, we then train high resolution images and efficiently skip the empty space. When we do so, we discard the features and MLP weights trained on the coarse images (which are trained to ignore fine details).

4.5. Inference time optimization

To avoid querying voxels that have no effect on the image (empty voxels; occluded voxels), we follow [55] by recording maximum blended alpha $\prod_{j=1}^{i-1}(1-\alpha_j)\alpha_i$ for each voxel from training views, and then culling all voxels with maximum blended alphas below the threshold $\tau_{\text{vis}} = 0.01$. This culls 98% of voxels on average but preserves transparent surfaces. We cull after the coarse training step (to accelerate fine-scale training) and then again after fine-scale training.

To avoid working on voxels occluded to a certain camera view, we evaluate intervals from the eye and stop working on a ray when a transmittance estimate $\prod_i(1-\alpha_i)$ falls below a threshold ($\tau_{\text{t}} = 0.01$). Furthermore, if an interval's alpha is below τ_{t}, there is no need to evaluate color.

In contrast to other voxel based real-time applications, we do not need to convert the trained model (so there is no precision loss, *etc*. from discretizing the model). While in principle, our inference time optimizations must result in loss of accuracy, the results of Sec. 5.3 suggest that this loss is negligible.

5. Experiments

We evaluate using both the offline rendering task (FPS$\leq$20) and the real-time rendering task (FPS$>$20). We use the NeRF-synthetic dataset [25] (synthetic images of size 800 $\times$ 800 with camera poses); a subset of the Tanks and Temples dataset [17] and the BlendedMVS dataset [53] (chosen by NSVF authors [21]). Tanks and Temples images are 1920 $\times$ 1080; BlendedMVS images are 768 $\times$ 576. Backgrounds in both datasets are cropped by NSVF. The qualitative results of our experiments can be seen in Fig. 7 and Fig. 8. In all quantitative measurements, we mark the best result by bold font and second best by italic font with underline.

5.1. Implementation detail

Training: We use PyTorch [31] for network optimization and customized CUDA kernels to accelerate ray-voxel intersection. For high resolution image training, both implicit and explicit models use a voxel grid of size 256^3 for NeRF-synthetic and BlendedMVS, and 320^3 for Tanks and Temples. For coarse model training, we take the voxel grid and images at 1/4 of the fine model scale. We follow NSVF's [21] strategy to sample rays from the training set, and we choose a batch size of 1024 pixels for coarse training, 6144 for fine training of Tanks and Temples, and 8192 for fine training of other datasets. The coarse model is trained for 5 epochs first explicitly, and then we train the model with implicit MLP until the validation loss has almost converged. Finally, we train the explicit grid initialized from the implicit model and stop the training when the total training time reaches 3 days. In total, the peak GPU memory usage is around 40GB. We use the Adam [16] optimizer with a learning rate of 5e-4 for the fine model, 1e-3 for the coarse model, and λ_s =1e-5 in the sparsity regularization loss.

	Method	PSNR ↑	SSIM ↑	LPIPS ↓
NeRF-Synthetic	NeRF [25]	31.00	0.947	0.081
	JaxNeRF [9]	31.65	0.952	0.051
	AutoInt [20]	25.55	0.911	0.170
	NSVF [21]	*31.74*	*0.953*	*0.047*
	DIVeR64	**32.32**	**0.960**	**0.032**
BlendedMVS	NeRF [25]	24.15	0.828	0.192
	JaxNeRF [9]	-	-	-
	AutoInt [20]	-	-	-
	NSVF [21]	*26.90*	*0.898*	*0.113*
	DIVeR64	**27.25**	**0.910**	**0.073**
Tanks & Temples	NeRF [25]	25.78	0.864	0.198
	JaxNeRF [9]	27.94	0.904	0.168
	AutoInt [20]	-	-	-
	NSVF [21]	**28.40**	*0.900*	*0.153*
	DIVeR64	*28.18*	**0.912**	**0.116**

Table 1. **Quantitative results on different benchmarks** show ours (DIVeR64) is overall the best compared with NeRF and its variant for offline rendering. '-' means no publicly available results. (**Best**; *Second best*).

Real-time application: Our real-time application is implemented by using CUDA and Python, with all the operations being parallelized per image pixel. For each frame and each pixel, ray marching finds a fixed number of hits on the voxel grid; the MLP is evaluated for each hit, and the result is then blended to the image buffer. This sequence is repeated until the ray termination criteria is reached (Sec. 4.5).

Storage: Because the voxel grid is sparse, we need to store only: indices and values of feature vectors for non-empty voxels; a binary occupancy mask; and the MLP weight. At inference, we keep feature vectors in a 1D array and then build a dense 3D array that stores the indices to the specific feature value, thereby reducing GPU memory demand without much sacrifice of performance.

5.2. Offline rendering

We evaluate the offline model by measuring the similarity between rendered and ground truth images using PSNR, SSIM [50], and LPIPS [57]. We use our DIVeR64 model for all scenes.

Baselines: We compare with original NeRF [25]; the reimplementation in Jax [9]; AutoInt [20]; and NSVF [21] (which uses similar voxel grid features). Pre-trained models from

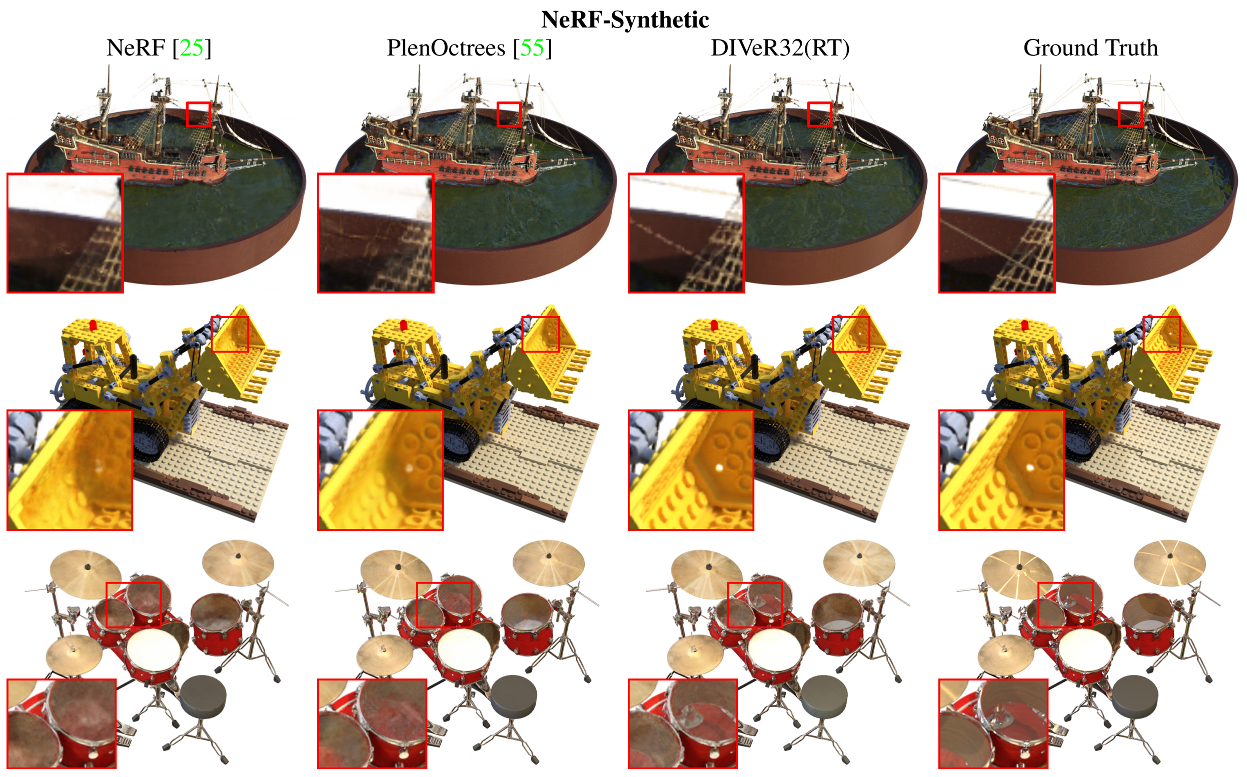


Figure 7. **Qualitative rendering results** show that our method successfully models fine and translucent structures (shrouds and ratlines on the ship; studs on lego; drumhead) which NeRF and PlenOctrees find hard. Our method is slightly slower than PlenOctrees (real-time) but much smaller; NeRF runs offline.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
NeRF-SH [55]	31.57	0.952	0.063
JaxNeRF+ [13]	**33.00**	**0.962**	*0.038*
NeRF [25]	31.00	0.947	0.081
DIVeR32	*32.16*	*0.958*	**0.032**

Table 2. **Image quality comparisons to real-time pre-trained models** on NeRF-synthetic strongly support our method (almost as good as JaxNeRF+). In Tab. 3, we show the performance of their corresponding baked real-time applications.

real-time NeRF variants produce good rendering quality but are trained and evaluated on very large computational resources (for example, JaxNeRF+ [13] doubles the feature size of NeRF's MLP and takes 5 times more samples for volume rendering, which is impractical for evaluation on a standard GPU). Therefore, we exclude them from the baseline models.

Results: DIVeR rendering quality is comparable with other offline baselines, while its architecture is much simpler (Tab. 1). Our PSNR is only slightly worse than that of NSVF on Tanks and Temples; but we use a much simpler decoder MLP.

Method	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑	MB ↓	GPU GB ↓
PlenOctrees [55]	*31.71*	**0.958**	0.053	*76±66*	1930	1.65±1.09
SNeRG [13]	30.38	*0.950*	0.050	**98±37**	*84*	1.73±1.48
FastNeRF [12]	29.97	0.941	0.053	-	-	-
KiloNeRF [33]	31.00	0.*950*	**0.030**	28±12	161	*1.68±0.27*
DIVeR32(RT)	**32.12**	**0.958**	*0.033*	47±20	**68**	**1.07**±**0.06**

Table 3. **Comparisons to other real-time variants** on NeRF-synthetic show that our method produces small models with very low GPU demand and renders very fast with very strong image quality metrics. PlenOctrees is baked from NeRF-SH; SNeRG is baked from JaxNeRF+; KiloNeRF and FastNeRF directly convert from the original NeRF. Performance measurements of FastNeRF are not publicly available.

5.3. Real-time rendering

For the real-time rendering task, we use our DIVeR32 model with the inference time optimization described in Sec. 4.5. Besides rendering quality, we show the inference time efficiency by running all the models on a GTX1080 GPU and recording their FPS and GPU memory usage. To compare the compactness of the architecture, we report the average memory usage for storing a scene. As most real-time models are converted from some pre-trained models, we also show the rendering quality of those models and compare the precision loss after the conversion. For the models that have

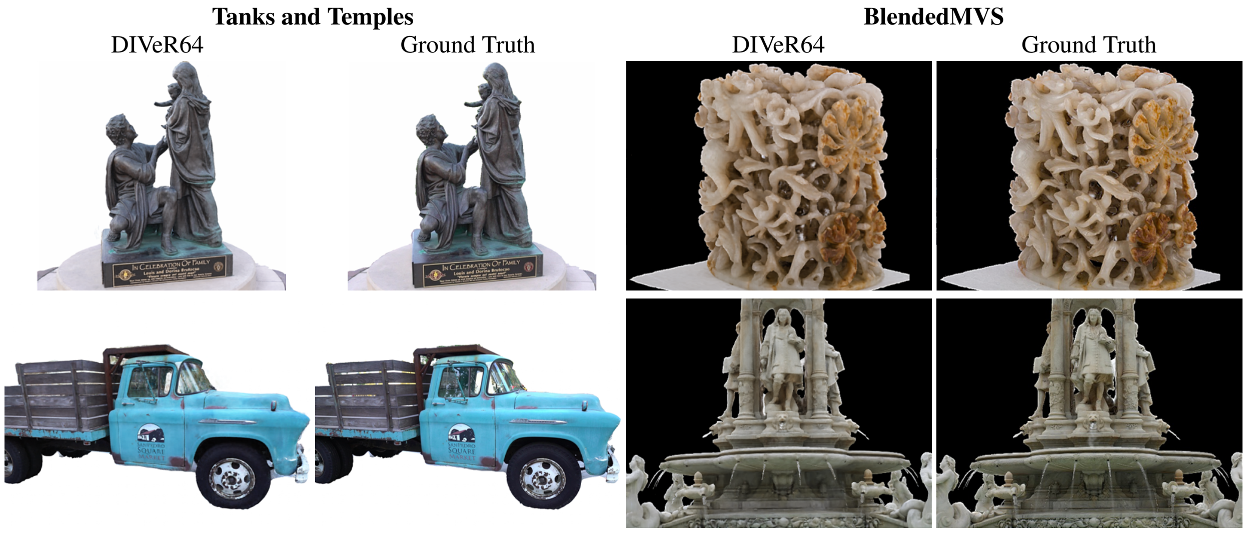


Figure 8. **Qualitative rendering results** of Tanks and Temples and BlendedMVS data show our method successfully models real-world fine structures.

variants based on quality-speed trade-off, we report their variants with the best rendering quality.

Baselines: For our real-time rendering baselines, we compare with PlenOctrees [55], SNeRG [13], FastNeRF [12], and KiloNeRF [33]. Since SNeRG did not provide their pre-trained models, we directly report the measurements from their paper. For the same reason, we report only the rendering quality for the FastNeRF as reported in their paper. For KiloNeRF, we measure the performance of those scenes for which their model was made available (chair, lego, and ship).

Results: Our rendering quality is either best (Tab. 3) or second best (Tab. 2), but our method achieves very high frame rates for very small models. All other methods must (1) convert to achieve a real-time form and then (2) fine-tune to recover precision loss after conversion. Fine-tuning is crucial for these models; for example, if SNeRG is not fine-tuned, its PSNR degrades dramatically to 26.68. In contrast, our model is evaluated as trained without conversion or fine-tuning. Early ray termination (Sec. 4.5) causes the mild degradation in quality observed in our real-time methods.

5.4. Ablation study

Architecture: We perform all our ablation studies on the NeRF-synthetic dataset. In Tab. 4, we show the performance trade-off between different network architectures. Without any real-time optimization, our model still runs faster than regular NeRF that takes minutes to run a single frame; if we use a smaller decoder for speed, there is a minor loss of quality but speed doubles (because DIVeR32 uses half as many registers as DIVeR64, allowing more threads to run in the CUDA kernel). Further economy with acceptable PSNR can be obtained by reducing the voxel grid size (compare DIVeR32 at 128 voxels yielding 30.42 PSNR for an about 12MB model to PlenOctree's variant with 30.7 PSNR, about 400MB). Fig. 9 compares different MLP sizes and voxel grid sizes qualitatively.

N	Decoder	RT	PSNR ↑	FPS ↑	MB ↓
256	DIVeR64	No	**32.32**	0.62	*62*
256	DIVeR32	No	32.16	0.62	68
128	DIVeR64	No	30.72	0.62	**12**
128	DIVeR32	No	30.53	0.62	**12**
256	DIVeR64	Yes	*32.30*	26±9	*62*
256	DIVeR32	Yes	32.12	*47±20*	68
128	DIVeR64	Yes	30.64	37±20	**12**
128	DIVeR32	Yes	30.52	**82±37**	**12**

Table 4. **Ablation study on network architecture** shows that real-time optimization (RT) and smaller MLPs (Decoder) involve minimal loss of rendering quality but huge speedups; going to a coarser grid (N) involves a larger loss of quality, for further very large speedup and improvement in model size.

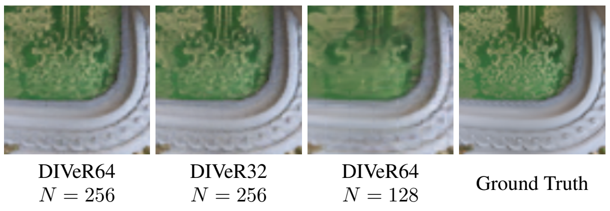
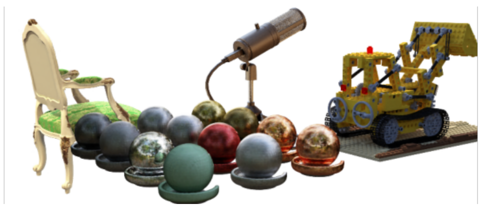


Figure 9. **Qualitative comparison of different architectures** shows larger voxel grid can better model the fine texture on the chair surface; switching to a smaller MLP does not affect the quality too much.



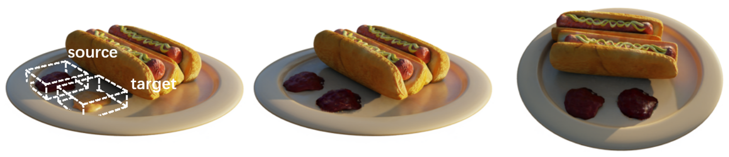


Figure 10. Our representation admits useful **editing** in a straightforward way. **Left:** shows a composite of pretrained models, obtained by blending voxel grids. **Right:** the mustard on the plate (first hot-dog) is replaced with ketchup (second hot-dog); this edit is preserved under view change (third hot-dog).

Integrator	Regularization	Data type	PSNR ↑	MB ↓
Det	Im-Ex	float32	**35.52**	<u>*64*</u>
Rand	Im-Ex	float32	33.89	67
Det	Im-Ex	uint8	<u>*35.44*</u>	**19**
Det	Im	float32	34.69	<u>*64*</u>
Det	Ex	float32	34.02	<u>*64*</u>

Table 5. **Ablation study on training strategy**. Implicit (Im) MLP initialization with explicit (Ex) training gives the best rendering quality. Mapping feature vectors with tanh allows features to be stored more efficiently (uint8) with acceptable loss of rendering qualities.

Training strategy: Tab. 5 shows the effect of different training strategies on the lego scene trained with our DIVeR64 model. The deterministic integrator is important: using a random integrator (implemented with sampling strategy of NSVF [21]) in train and test causes a notable loss of quality. The implicit-explicit training strategy is important: replacing it either with a pure implicit model or with no implicit MLP initialization (compare Fig. 3) results in a less significant loss of quality. A lower precision representation of the feature vectors (trained with a tanh mapping; converted to unit8) results in a minor loss of quality, but the model size is reduced by a factor of 3.

Editability: The voxel based representation allows us to perform some basic scene manipulations. We can composite scenes by blending their voxel grids and then using the corresponding decoder for rendering. Because feature vectors incorporate high level information of the local appearance, we can extract the segmentation of an object from a selected area by using k-mean clustering on the feature vectors, which allows us to swap objects without noticeable artifacts. Fig. 10 shows some examples.

6. Limitations

Training NeRF-like representations is expensive, and our method does not speed up training in any natural way. Aliasing error in a deterministic integrator tends to be patterned, whereas a stochastic integrator breaks it up [8]. In turn, rays near tangent to voxels or accumulated error in the intersection routine can cause problems (Fig. 11). A mixed stochastic-deterministic method (say, jittering voxel positions) may help. Our method, like NeRF, can fail to model view dependent effects correctly (Fig. 12); more physical modeling might help. Our method does not currently apply to unbounded scenes, and our editing abilities are currently quite limited.

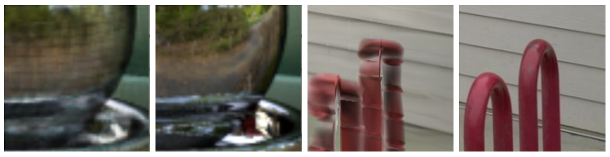


Figure 11. **Intersection errors** can cause aliasing problems, typically for structures near the scale of the voxel grid and rays near tangent to voxel faces. Worse, our current intersection routine, while efficient, can accumulate intersection errors, occasionally producing blocky aliasing artifacts at some view directions and scales (the scale of the blocky artifacts in each case is close to the scale of the voxel grid).

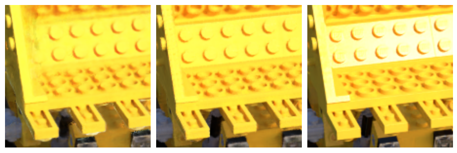


Figure 12. **NeRF variants fail to extrapolate view dependent effect** that is unseen from the training set. Neither ours nor other NeRF variants are able to model the reflection on the shovel correctly.

Acknowledgements: We thank Sara Aghajanzadeh, Derek Hoiem, Shenlong Wang, and Zhen Zhu for their detailed and valuable comments on our paper.

References

[1] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, 1991. 1

[2] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. Neural point-based graphics, 2020. 2

[3] Mark Boss, Raphael Braun, Varun Jampani, Jonathan T. Barron, Ce Liu, and Hendrik P.A. Lensch. Nerd: Neural reflectance decomposition from image collections. In *ICCV*, 2021. 2

[4] Phelim P. Boyle. Options: A monte carlo approach. *Journal of Financial Economics*, 4(3):323–338, 1977. 1, 2

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

[6] Anpei Chen, Zexiang Xu, Fuqiang Zhao, Xiaoshuai Zhang, Fanbo Xiang, Jingyi Yu, and Hao Su. Mvsnerf: Fast generalizable radiance field reconstruction from multi-view stereo, 2021. 2

[7] Julian Chibane, Aayush Bansal, Verica Lazova, and Gerard Pons-Moll. Stereo radiance fields (srf): Learning view synthesis from sparse views of novel scenes. In *CVPR*, 2021. 2

[8] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986. 8

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

[10] Yilun Du, Yinan Zhang, Hong-Xing Yu, Joshua B. Tenenbaum, and Jiajun Wu. Neural radiance flow for 4d view synthesis and video processing. In *ICCV*, 2021. 2

[11] Chen Gao, Ayush Saraf, Johannes Kopf, and Jia-Bin Huang. Dynamic view synthesis from dynamic monocular video. In *ICCV*, 2021. 2

[12] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. Fastnerf: High-fidelity neural rendering at 200fps, 2021. 3, 6, 7

[13] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis, 2021. 3, 4, 6, 7

[14] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, Jan. 1984. 2

[15] Petr Kellnhofer, Lars Jebe, Andrew Jones, Ryan Spicer, Kari Pulli, and Gordon Wetzstein. Neural lumigraph rendering. In *CVPR*, 2021. 2

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015. 5

[17] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Trans. Graph.*, 36(4), July 2017. 5

[18] Tianye Li, Mira Slavcheva, Michael Zollhoefer, Simon Green, Christoph Lassner, Changil Kim, Tanner Schmidt, Steven Lovegrove, Michael Goesele, and Zhaoyang Lv. Neural 3d video synthesis, 2021. 2

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

[20] David B. Lindell, Julien N. P. Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural volume rendering. In *CVPR*, 2021. 2, 5

[21] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. In *NeurIPS*, 2020. 1, 2, 3, 5, 8

[22] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph.*, 38(4):65:1–65:14, July 2019. 2

[23] 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. In *CVPR*, 2021. 2

[24] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion. *ACM Transactions on Graphics (TOG)*, 38:1 – 14, 2019. 2

[25] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 1, 2, 4, 5, 6, 8

[26] T. Neff, P. Stadlbauer, M. Parger, A. Kurz, J. H. Mueller, C. R. A. Chaitanya, A. Kaplanyan, and M. Steinberger. Donerf: Towards real-time rendering of compact neural radiance fields using depth oracle networks. *Computer Graphics Forum*, 40(4):45–59, 2021. 2

[27] Michael Niemeyer and Andreas Geiger. Giraffe: Representing scenes as compositional generative neural feature fields. In *CVPR*, 2021. 2

[28] Michael Niemeyer, Lars M. Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *CVPR*, pages 3501–3512, 2020. 2

[29] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. Nerfies: Deformable neural radiance fields. In *ICCV*, 2021. 2

[30] Keunhong Park, Utkarsh Sinha, Peter Hedman, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Ricardo Martin-Brualla, and Steven M. Seitz. Hypernerf: A higher-dimensional representation for topologically varying neural radiance fields. *arXiv preprint arXiv:2106.13228*, 2021. 2

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 5

[32] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. D-NeRF: Neural Radiance Fields for Dynamic Scenes. In *CVPR*, 2020. 2

[33] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps, 2021. 3, 6, 7

[34] Konstantinos Rematas and Vittorio Ferrari. Neural voxel renderer: Learning an accurate and controllable rendering tool. In *CVPR*, 2020. 2

[35] Konstantinos Rematas, Ricardo Martin-Brualla, and Vittorio Ferrari. Sharf: Shape-conditioned radiance fields from a single view. In *ICML*, 2021. 2

[36] Gernot Riegler and Vladlen Koltun. Free view synthesis. In *ECCV*, 2020. 2

[37] Gernot Riegler and Vladlen Koltun. Stable view synthesis. In *CVPR*, 2021. 2

[38] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa, and Hao Li. Pifu: Pixel-aligned implicit function for high-resolution clothed human digitization. In *ICCV*, pages 2304–2314, 2019. 2

[39] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *CVPR*, 2016. 2

[40] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. Graf: Generative radiance fields for 3d-aware image synthesis. In *NeurIPS*, 2020. 2

[41] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. Deepvoxels: Learning persistent 3d feature embeddings. In *CVPR*, 2019. 2

[42] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *NeurIPS*, 2019. 2

[43] 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. In *CVPR*, 2021. 2

[44] Pratul P. Srinivasan, Richard Tucker, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, and Noah Snavely. Pushing the boundaries of view extrapolation with multiplane images. In *CVPR*, pages 175–184, 2019. 2

[45] 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. In *CVPR*, 2021. 2

[46] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Trans. Graph.*, 38(4), July 2019. 2

[47] Alex Trevithick and Bo Yang. Grf: Learning a general radiance field for 3d scene representation and rendering. In *arXiv:2010.04595*, 2020. 2

[48] Cen Wang, Minye Wu, Ziyu Wang, Liao Wang, Hao Sheng, and Jingyi Yu. Neural opacity point cloud. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(7):1570–1581, 2020. 2

[49] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P. Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas A. Funkhouser. Ibrnet: Learning multi-view image-based rendering. In *CVPR*, 2021. 2

[50] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. 5

[51] Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. Nex: Real-time view synthesis with neural basis expansion. In *CVPR*, 2021. 2, 3

[52] Wenqi Xian, Jia-Bin Huang, Johannes Kopf, and Changil Kim. Space-time neural irradiance fields for free-viewpoint video. In *CVPR*, pages 9421–9431, 2021. 2

[53] Yao Yao, Zixin Luo, Shiwei Li, Jingyang Zhang, Yufan Ren, Lei Zhou, Tian Fang, and Long Quan. Blendedmvs: A large-scale dataset for generalized multi-view stereo networks. In *CVPR*, 2020. 5

[54] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. In *NeurIPS*, volume 33, 2020. 2

[55] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021. 1, 3, 5, 6, 7

[56] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelnerf: Neural radiance fields from one or few images, 2020. 2

[57] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018. 5

[58] Xiuming Zhang, Pratul P. Srinivasan, Boyang Deng, Paul E. Debevec, William T. Freeman, and Jonathan T. Barron. Nerfactor: Neural factorization of shape and reflectance under an unknown illumination. *ArXiv*, abs/2106.01970, 2021. 2

[59] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification. *ACM Transactions on Graphics (TOG)*, 37:1 – 12, 2018. 2