CHAPTER 41

Density and Differentiable Rendering

Point clouds have problems as a representation. For example, if you get too close to a point cloud, you can see holes in it (Figure **??**, for example). But the voxels of Section 36.3 aren't particularly attractive either. Voxels are always simultaneously too big (meaning the object has blocky structure) and too small (so you have unreasonably many voxels). This chapter explores a generalization of the idea of the voxel.

You should think of voxels as a function approximation procedure for 3D. You carve up 3D into voxels, then place a value in each voxel – the result is a piecewise constant function. But there are other ways to construct models of functions. For example, you could build some form of tree dividing 3D into cells (like the trees of Section 36.3) and use a constant value in each cell. As another example, you could use a neural network to accept the coordinates of a point and produce some function value.

Interpreting the function represented by simple voxels is straightforward. You cannot see through voxels that are full, and you can see through voxels that are empty. You could store a color in each voxel that is full (there is not much point in coloring empty voxels). It is then straightforward to produce a picture of – *render* – the voxel grid using a simple ray tracer (Section 36.3).

The rendering presents problems. If you have large voxels, the rendering may be blocky. Making it less blocky may involve a fine grid, and so a lot of voxels. A less obvious problem that you can't get a useful gradient of the picture with respect to volume occupancy, camera position and so on.

This chapter deals with function approximations that are more general than voxels. The big advantage of these approximations is that the rendered image they produce is differentiable with respect to the parameters of the function representation. This is a fantastic advantage. You can adjust the representation to look like an example picture using an optimization procedure in a straightforward way **exercises**. It is possible to optimize some non-differentiable functions, but doing so can be very difficult and for some cases is impossible.

41.1 PRELIMINARIES

41.1.1 The Plenoptic Function

The underlying object that a voxel representation is modeling is known as the *plenoptic function*.

Definition: 41.1 *The Plenoptic Function*

The function that maps a point $\mathbf{x}$ and a direction ω to the amount of light travelling through that point in that direction is known as the plenoptic function.

Voxels encode the plenoptic function because for any point and any direction, you can query the voxel representation and recover the amount of light travelling through that point and in that direction. Any system that will generate new images of an object or scene from a set of existing images is hiding a form of plenoptic function approximation somewhere. In some cases – for example, obtain the images using an SFM pipeline (Chapter 36.3) followed by a multiview stereo procedure (Chapter 36.3) – it can be hard to determine just where the plenoptic function approximation is happening. Recent procedures tend to produce fairly explicit approximations. There is a long history of explicit approximate representations of plenoptic functions. Until recently, it was difficult to obtain good images at high resolution with reasonably sized representations. Improvements are likely due to very significant advances in methods for function approximation.

41.1.2 Glowing Gas

Simple voxels – full or empty – are easy to render. To render a pixel, construct a ray from the focal point passing through the pixel. Find the first full voxel that the ray hits. Take the color of that voxel, and use it to compute the color represented by the ray. You might just use the color of the voxel for the color of the ray. Alternatively, you might engage in a complex shading calculation that uses the color of the voxel, information about lights and even a variety of other rays. Now average the color of all rays passing through the pixel, and place that value in the pixel. Full accounts of the possibilities presented by ray tracing appear in, for example, [**?**]. Interpreting the function represented by a more sophisticated function approximator requires some knowledge of volume rendering.

Now replace the voxels with a gas. This gas consists of a cloud of fine glowing particles. At each point $\mathbf{x}$ in 3D, you know the density of these particles $\sigma(\mathbf{x})$. Particles absorb light and also glow with color $\mathbf{c}(\mathbf{x})$. With a certain amount of work (below), you can determine the amount of light arriving at a pixel under this model. Because you can determine the amount of light arriving at a pixel, you can construct an image of this gas. Now model objects by adjusting the properties of the gass – its color and the density – to give the results you want.

41.1.3 The Effect of an Element of Gas

Figure 41.1 shows a ray passing into an arbitrarily thin slab of the gas. The ray is parametrized by t. It starts at the eye ($t = 0$), and t increases as the ray proceeds out into the scene. The amount of light at t is $I(t)$. The amount of light could be a function of wavelength. It is more usual to think of the amount of light as a three vector (R, G, and B), though doing so can lead to physical errors (below; **exercises**). The slab of gas absorbs some fraction of the light passing through

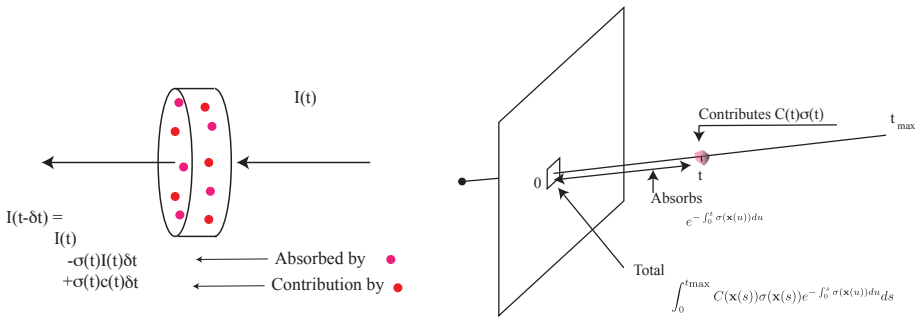


FIGURE 41.1: *On the* **left**, *a ray of light travels through a slab of gas with thickness* δt. *A fraction of the light arriving at* t *is absorbed. This fraction is proportional to the density of the gas. Furthermore light is added to the ray by the glowing material. In total,* $I(t-\delta t) = I(t) - \sigma(t)I(t)\delta t + \sigma(t)\mathbf{c}(t)\delta t$. *On the* **right**, *a sketch of how to render a pixel in an image of the gas. Construct a ray through the focal point and the pixel. Compute the brightness at* 0 *(the parameter value corresponding to the pixel on the ray). This brightness is obtained by summing contributions from every element along the ray; for each of these contributions you must account for the fraction absorbed along the section of ray between the element and the pixel.*

it and contributes some more because it is glowing. The amount of light absorbed is proportional to the density, because the probability of absorbtion depends on whether the light hits a particle or not. The amount of light generated in the slab is proportional to the density, because it depends on the number of particles in the slab.

Physical modelling yields

$$\begin{aligned} I(t-\delta t) &= I(t) - \text{Light absorbed by slab} + \text{Light generated in slab} \\ &= I(t) - \sigma(\mathbf{x}(t))\delta t I(t) + c(\mathbf{x}(t))\sigma(\mathbf{x}(t)). \end{aligned}$$

Some work with differential equations, etc. **exercises** yields that

$$\begin{aligned} I(0) &= \int_{s=0}^{\infty} c(\mathbf{x}(s))\sigma(\mathbf{x}(s)) e^{-\int_{u=0}^{s}\sigma(\mathbf{x}(u))du} ds. \\ &= \int_{s=0}^{\infty} [\text{amount generated at } s]\,[\text{absorption from } s \text{ to } 0]\, ds. \end{aligned}$$

This integral is differentiable as a function of density and color. If the density and color are differentiable, then the integral is differentiable as a function of ray position and direction, too **exercises**.

For some choices of glowing particle and absorbing particle, σ and $\mathbf{c}$ might depend on the direction in which the ray passes through the gas. As an extreme example, think about a glowing spherical particles that are (a) all oriented in space in the same way (perhaps there is a magnetic field) and (b) all glow red on the top half and blue on the bottom half. As you look at these particles from different viewpoints, you might see red (from on top), purple (from the side) or blue (from



FIGURE 41.2: *Notation for estimating the integral representing how much light a glowing gas produces at a pixel. Divide the ray into intervals. The first interval starts at $t = 0$. The far end of the i'th interval is $\mathbf{x}_i$, and the length of the i'th interval is d_i.*

below). In this case, write $\sigma(\mathbf{x}, \omega)$ and $\mathbf{c}(\mathbf{x}, \omega)$ for the density and color respectively, where ω is the direction in some angular coordinate system.

41.1.4 Rendering Hot Gas

Imagine you wish to render some of this glowing cloud. Choose a camera – focal point and image plane – from which you want to render. Construct a ray from the focal point passing through the pixel. Choose the ray's parametrization to have $t = 0$ at the image plane, and positive t in the direction toward the world. Assume that the cloud is bounded, so that the largest possible value of t along the ray is known (call this T_{max}). Now evaluate the integral.

Evaluating the integral can be quite hard, and might be very expensive depending on the particular gas. Here is one strategy that works well in principle. Divide the ray into intervals. The first interval starts at $t = 0$. The far end of the i'th interval is $\mathbf{x}_i$, and the length of the i'th interval is d_i (Figure 41.2). You can approximate the integral as

$$\sum_i \left[\text{term generated within } d_i\right] \left[\text{fraction absorbed between } \mathbf{x}_{i-1} \text{ and } 0\right].$$

Assume that $\mathbf{c}(\mathbf{x})$ is constant within the i'th interval. Write $\mathbf{c}_i$ for the value. The amount of light generated within that interval is

$$\int_0^{d_i} \mathbf{c}_i \sigma(u) e^{-\int_0^u \sigma(s) ds} du$$

assuming that $\sigma(u) = \sigma_i$ within the interval and a little work (**exercises**) you can show that this is

$$\mathbf{c}_i \left[1 - e^{-\sigma_i d_i}\right].$$

With some further work (**exercises**), you can estimate the integral as

$$\sum_i \left[\mathbf{c}_i \left(1 - e^{-\sigma_i d_i}\right)\right] [a_i]$$

where a is defined by a recurrence relation, giving $a_1 = 1$ and $a_{i+1} = a_i e^{-\sigma_{i-1} d_{i-1}}$. You can differentiate the estimate with respect to σ and $\mathbf{c}$ (pretty obviously **exercises**). The value you obtain will be the value of a different random variable. The expected value of this random variable will be the true value of the derivative. However, the variance could be fairly high.

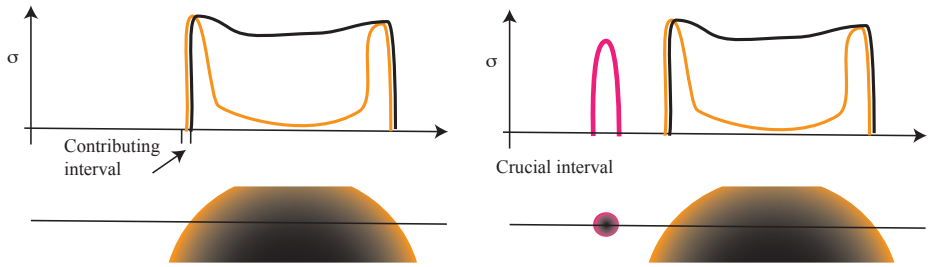


FIGURE 41.3: *The volume rendering integral can be very difficult to estimate from samples. The* **center** *shows two objects – represented as density functions.* **Top** *and* **bottom** *show the density plotted as a function along the ray. Notice that multiple different densities (the* **black** *and the* **orange** *on the top ray) will yield the same integral. This is because the small region of high density closest to the eye (here labelled the contributing interval) determines the value of the integral. If there is no sample in the interval, the estimate could be bad. In the* **bottom** *ray, not having a sample in the* **pink** *object will mean that the integral is orange, rather than pink. But getting a sample to the right place might be hard – the crucial interval is small.*

If you choose the $\mathbf{x}_i$ at random along the ray, the number you estimate is the value of a random variable (different choices yield different numbers). The weak law of large numbers guarantees that the mean of that random variable is the true value of the integral **exercises**. The variance is quite hard to determine, but is $O(1/N)$, meaning that increasing the number of samples improves the estimate annoyingly slowly.

The integral is particularly hard when the gas cloud is "clumpy", which is the most useful case for object modelling. Imagine σ is either very close to zero or large. Light passes through regions where σ is very close to zero with very little absorption, so such regions are like free space. An element of space where σ is large may generate a lot of light, but will also absorb light passing through it. Such elements are like elements of solids. Figure 41.3 shows how you might model an object as a "clumpy" gas. The integral is difficult in this case. Samples that land in free space contribute nothing to the integral, but consume computation. If samples miss some elements where σ is large, the estimate could be wildly wrong.

You can reduce the variance of the estimate further using an idea known as *importance sampling*. Assume you know a probability distribution $p(T)$ from which you can draw samples conveniently. Assume $p(T)$ tends to be large around t that contribute large values to the integral and small around t that contribute small values and is never zero. Then draw N samples t_i from $p(T)$ and sort them so if $i < j$, then $t_i < t_j$. Now compute

$$\sum_{i=1}^{N} \frac{1}{p(t_i)} c(\mathbf{x}(t_i))\sigma(\mathbf{x}(t_i)) e^{\sum_{j=1}^{i-1} \frac{\sigma(\mathbf{x}(t_j))}{p(t_j)}}.$$

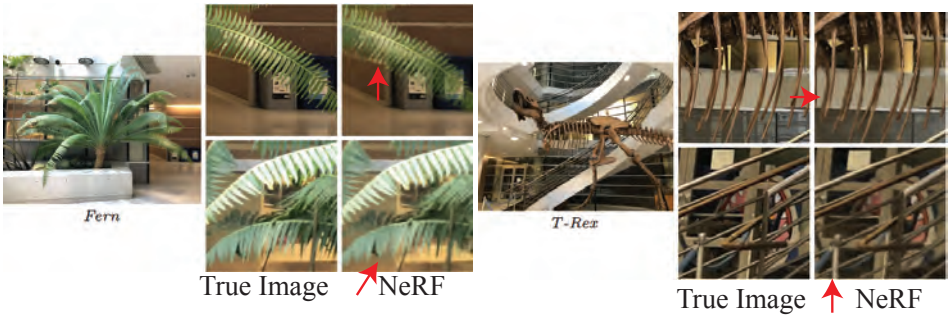


FIGURE 41.4: *NeRF generalizes very well, and images from new viewpoints are accurate. For each of these examples,* **left** *is the full scene,* **center** *shows two detail regions from the true image, and* **right** *shows the same regions from the NeRF reconstruction. Close examination reveals some smoothing of fine detail (the* **red** *arrows point to some cases) but the recostructions are very strong.* Image credit *Constructed using parts of Figure 6 of "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis, " Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, ECCV 2020*

This estimate can have very much lower variance.

Another method to control variance is *stratified sampling* (which is actually a form of importance sampling **exercises**). The difficulty with taking a set of uniform samples along the ray is that some samples will be quite close to their nearest neighbor. Others might be quite far from their nearest neighbor. This occurs because the samples are independent, so no sample "knows" where any other sample is. The most usual way to form a stratified sample for this integral is to break the ray up into N discrete cells of the same length, then draw a uniform sample inside each cell.

41.2 NERF

You have M ground truth images $\mathcal{G}_i$ of some object from a range of different viewpoints. You want to generate an image of that object in a new camera. In the simplest case, you know the camera parameters $\mathcal{P}_i$ for the i'th ground truth image, and there is no background in the image. NeRF models the object using the glowing gas model. There is some $\sigma(\mathbf{x}, \omega)$ and $\mathbf{c}(\mathbf{x}, \omega)$ that yielded the images. You now determine what they are, then render them from different viewpoints.

41.2.1 Basic NeRF

The density $\sigma(\mathbf{x}, \omega)$ and the color $\mathbf{c}(\mathbf{x}, \omega)$ are modelled using a differentiable function approximation procedure. The original paper used a *multi-layer perceptron* (a sequence of fully connected layers) that accepted $\mathbf{x}$ and ω, and produced density and color. Write θ for the parameters of this network. Write $\mathcal{P}$ for a set of camera parameters. For any choice of θ and $\mathcal{P}$, you can produce an image $\text{Render}(\mathcal{P}, \theta)$ by rendering using one of the procedures above. This image depends on camera

parameters in the obvious way and network parameters because the network parameters produce density and color for each sample point on each ray. The rendering is differentiable with respect to $\mathcal{P}$ and θ.

Procedure: 41.1 *Training a NeRF model*

Take your ground truth images $\mathcal{G}_i$ and their associated camera matrices $\mathcal{P}_i$. Choose parameters for the network by obtaining $\hat{\theta}$ so that rendering that network from the given viewpoints yields the images. To do this, choose a cost function $C(\mathcal{R}, \mathcal{G})$ that compares a rendered image $\mathcal{R}$ to a ground truth image $\mathcal{G}$. Now obtain $\hat{\theta}$ to minimize

$$\frac{1}{M}\sum_i C(\text{Render}(\mathcal{P}_i, \theta), \mathcal{G}_i).$$

Minimize using stochastic gradient descent (or one of its more sophisticated variants), where you choose a batch of pixels across multiple images, render those pixels, then compare to the ground truth pixels.

Once you have trained a NeRF model, you can generate new images of the object by simply rendering this density field in a new camera. You should think of a NeRF model as a regression model. At a high level, a trained NeRF accepts camera parameters and makes images. You can evaluating such a model on two points. First, it should make the ground truth images (or something similar) when give the ground truth camera parameters. Mostly, this is a just a check. If the model cannot do this, then it can't be much use. Second, and much more important, it should generalize. If you give it a new set of camera parameters, you should get a reasonable image.

Procedure: 41.2 *Evaluating a NeRF model*

Obtain a collection of sets of images of an object from different views, where you know the camera parameters for each image. You could simply render a 3D object; you could use the methods of Chapter 36.3 to obtain camera parameters for real images; or you could use one of the datasets listed below. Split each set into a train and test component. Train the NeRF using the training component, then render from the test viewpoints and compare to the true test images using one of the paired image metrics of Section 36.3.

The values returned by a NeRF MLP are not the value of the plenoptic function, but – as the rendering procedure showed – can be converted into a plenoptic function. Evaluate the amount of light travelling through a point $\mathbf{x}$ in direction ω using a NeRF by rendering the ray starting at $\mathbf{x}$ and proceeding in direction ω. In turn, NeRF is a plenoptic function approximation.

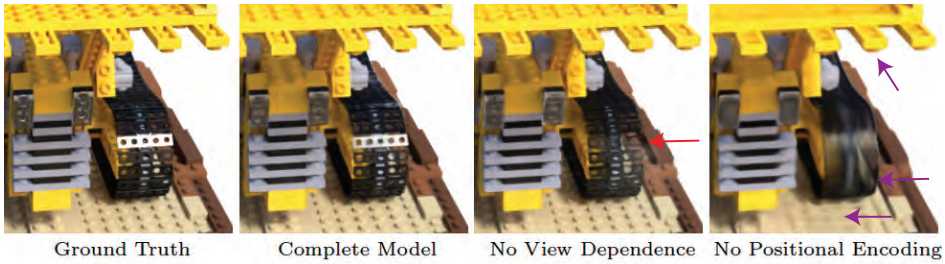


FIGURE 41.5: *A comparison of various different configurations of a NeRF model shows that both orientation dependence and positional encoding are helpful.* **Far left** *shows a detail from a ground truth image, obtained from a view not used in training the NeRF;* **center left** *shows the same detail, but now from a prediction of that view obtained using a NeRF model that has both orientation dependence and positional encoding;* **center right** *shows the same detail, but now the NeRF model omits orientation dependence; and* **far right** *shows the same detail, but now the NeRF model omits positional encoding. Omitting the orientation dependence means the specular reflection is not properly recovered (***red** *arrow). Omitting the positional encoding results in some unnecessary smoothing (***purple** *arrows).* Image credit *Constructed from Figure 4 of "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis, " Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, ECCV 2020*

41.2.2 A Working NeRF

Building a working NeRF requires care. In the original NeRF, the function approximation was a multi-layer perceptron or MLP. This is a stack of fully connected layers. The very simplest MLP would accept the position $\mathbf{x}$ and return a color and density. It turns out that omitting ω means the model does not correctly represent specular effects. In the fitted model, free space must have small $\sigma(\mathbf{x}, \omega)$. This means the same point $\mathbf{x}_0$ on the object viewed from different directions should have color about $\mathbf{c}(\mathbf{x}, \omega)$. If this color does not depend on ω, different views will have the same color. But specularities look different when you look at them from different directions, so leaving out the ω will result in problem images (Figure 41.5).

Another option is to build an MLP that accepts position and direction, and returns color and density. MLP's tend not to produce good approximations for functions that change very fast, so the simplest MLP produces somewhat smoothed predictions (Figure 41.5). Very significant improvements are possible using *positional encoding* (compare the positional embedding of Section 36.3). Take $\mathbf{x}$ and compute $\mathbf{v}(\mathbf{x})$, where $\mathbf{v}$ is a vector of functions that vary increasingly fast over space (the original paper used Fourier basis functions). Now build an MLP that accepts $\mathbf{v}(\mathbf{x})$ and ω and returns a color and density. A smooth function of a quickly varying Fourier basis element can vary quickly, and as a result the predictions are notably sharper.

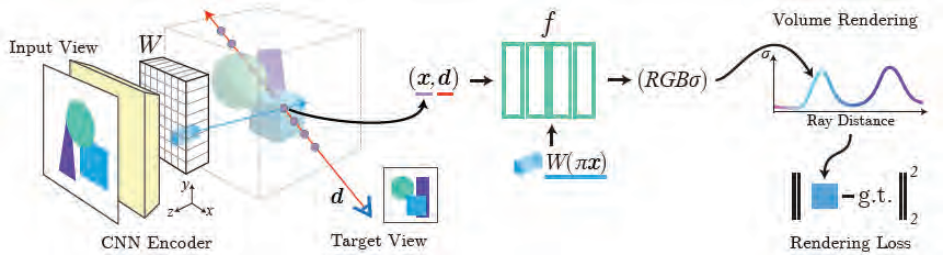


FIGURE 41.6: *pixelNeRF feeds features computed by an image encoder to the MLP that predicts density and color. By doing so, it encourages new images that are close to the original image to look more like it. When there are many training images from many viewpoints, this makes little difference. But as Figure 41.7 illustrates, when there are very few training images, there are real improvements in the predicted images.* Image credit *Constructed using Figure 2 of "pixelNeRF: Neural Radiance Fields from One or Few Images" Alex Yu, Vickie Ye, Matthew Tancik, Angjoo Kanazawa, CVPR 2021*

41.2.3 Some NeRF Variants

The original NeRF was relatively slow to train, relatively slow to render, and tended to require quite a large number of training images with quite accurate estimates of the camera parameters. More recent variants train fast, render fast, and are quite relaxed about camera estimates. I will describe some variants here, but the topic is far too broad for a comprehensive review (and there are some reviews in the reading list).

One natural source of improvement is speeding up the integration. Given a learned NeRF, use an octtree to encode the parts of the density field that are not free space. Once you have the octtree, you can speed up integration by intersecting a ray with the octtree and only drawing samples inside the occupied cells.

NeRF tends to produce poor estimates when there are too few training images (sometimes known as the *few-shot* case). pixelNeRF obtains improved behavior in this case by using image features to inform the MLP's prediction of density and color. Assume you have few training images. Find the training image closest to the image you wish to render. When you render a frame, you sample points along a ray. Since you know where the training images are in a global coordinate system, you know where a sample point projects to in a training image. Collect a set of image features at that point from a feature tensor computed using an encoder. Now supply these features as well as direction to the MLP that computes density and color (Figure 41.6). Of course, the MLP will need to be trained with these features as well. The pixelNeRF procedure effectively encourages views that are close to the training view to be consistent with the training view. If there are enough training images, doing so is unlikely to matter. But if there are very few training images, significant improvements result (Figure 41.7).

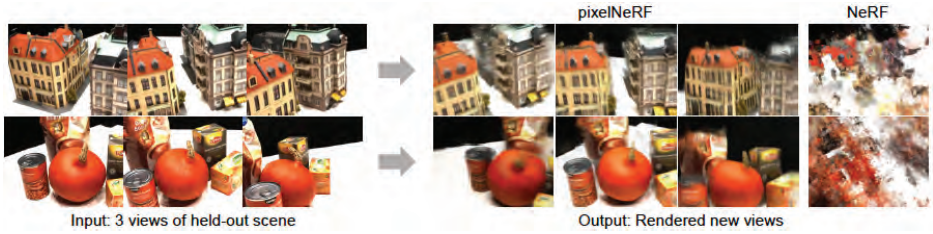


FIGURE 41.7: *A few-shot example comparing pixelNeRF and NeRF. For each scene, there are three training images (***left***). **Center** shows pixelNeRF reconstructions for three new viewpoints. Notice these viewpoints are somewhat close to the training images. **Right** shows the NeRF reconstruction from one of these new viewpoints. There is no reason to show more than one NeRF view, because the reconstruction is completely disorganized.* Image credit *Constructed using part of Figure 1 of "pixelNeRF: Neural Radiance Fields from One or Few Images" Alex Yu, Vickie Ye, Matthew Tancik, Angjoo Kanazawa, CVPR 2021*

Definition: 41.2 *Spherical harmonics*

Spherical harmonics are a set of orthonormal basis functions on the unit sphere. They can be constructed as the value of homogenous polynomials on the surface of the sphere. This means that they can be organized by the degree of the homogenous polynomial. For ω a point on the unit sphere – equivalently, a direction – write $\phi_d^i(\omega)$ for the i'th spherical harmonic derived from a polynomial of degree d. In vision and graphics applications, the first nine spherical harmonics are the most important. There is one degree 0 spherical harmonic – a constant function. There are three degree 1 spherical harmonics, which you can obtain by considering the values of the functions x, y, or z restricted to the unit sphere. There are five degree 2 spherical harmonics **exercises**. You might expect six, but on the surface of the sphere $x^2 + y^2 + z^2 - 1 = 0$, so there is a redundancy **exercises**. The main use of spherical harmonics is to represent functions on the unit sphere – equivalently, functions of direction. Such a function can be represented by its coefficients k_d^i, which yield the function

$$\sum_d \sum_i k_d^i \phi_d^i(\omega).$$

exercises

An alternative function approximation uses *spherical harmonics*. Rather than building an MLP that predicts color $\mathbf{c}$ and density α as a function of point $\mathbf{x}$ and direction ω, you could predict a set of spherical harmonic coefficients for $\mathbf{c}$. The MLP then accepts $\mathbf{x}$ and produces α and $\mathbf{k}$ – a set of spherical harmonic coefficients

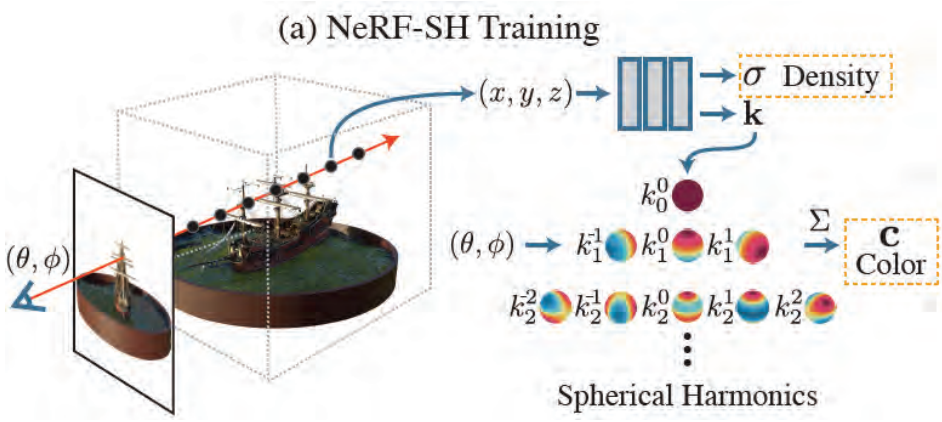


FIGURE 41.8: *Rather than predicting density and color for a point and direction, you can predict density and a set of spherical harmonic coefficients for each of R, G and B, then use the direction and the spherical harmonic coefficients to reconstruct the color. Doing this results in a somewhat simpler learning problem. As Figure* **??** *shows, there is no discernible loss in quality in the reconstructed images.* Image credit *Constructed using part of Figure 2 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*

for each of R, G and B (Figure 41.8). This means that you can determine the color at a point in any direction by evaluating the MLP, then using the spherical harmonics to obtain the color. Notice this means that this creates limits on how the color varies with directions (because you use a limited number of spherical harmonic coefficients **exercises**). This limit appears to have no important effect in practice. A model trained in this way is referred to as a *NeRF-SH*; there are significant speed improvements in training [**?**].

A further change in the function approximation yields an important speedup in rendering. Construct a dense set of samples of the NeRF-SH on some grid. Doing so is practical, because the NeRF-SH is a function of position (3D) rather than position and direction (5D). Each sample is a vector containing density and spherical harmonic coefficients. Use these samples to build an easily queried model. The Plenoctree's paper [**?**] built an octree, recursively splitting a cell if the samples in the cell vary enough, but you could build a kd-tree instead **exercises**. Computing the integral along a given ray involves evaluating the density and spherical harmonic coefficients at some set of sample points along on the ray. Do so by first finding the octree leaf in which the sample point lies **exercises**, then using the samples in that leaf to evaluate the function at the sample point **exercises**.

Plenoctrees goes further in modifying the representation of the function to obtain a Plenoctree []. Once the MLP has been trained, sample it at some points in 3D. For the moment, assume you use a voxel grid. At each vertex, obtain the α and $\mathbf{k}$ from the MLP. To render, compute the color and density at some

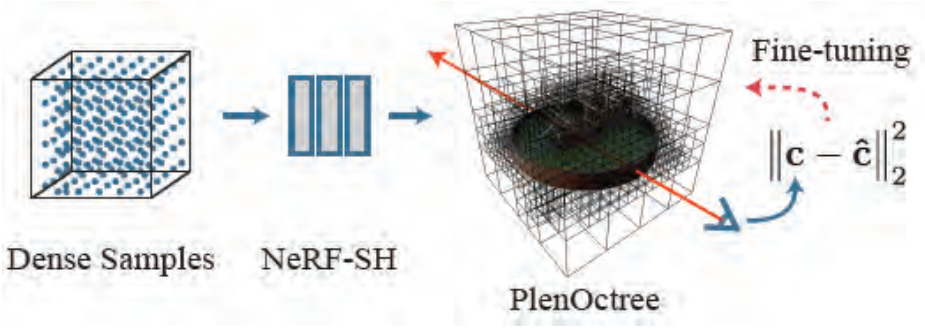


FIGURE 41.9: Image credit *Constructed using part of Figure 2 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*

point $\mathbf{x}$ on a ray by determining what voxel the point lies in, then using *trilinear interpolation* to interpolate spherical harmonic coefficients and density values at that point. Trilinear interpolation is a natural generalization of linear interpolation (like bilinear interpolation), and is explored in an exercise **exercises**. The result is very fast rendering. This rendering is differentiable in the values at each vertex of the grid. This means you can use stochastic gradient descent or a variant to adjust the values at the grid vertices to improve the renderings of training images.

You do not need to use a voxel grid to represent the function approximation. What is required is a set of samples in space, and some way to interpolate between them. Interpolation requires knowing which samples contribute to the interpolate. For example, if you use a voxel grid, you find the vertices of the voxel containing the sample you want to evaluate during rendering. You could, for example, use an octtree like plenOxels. In this case, samples live on each vertex; you find the samples to interpolate by finding the vertices of a cell; and you interpolate by trilinear interpolation. Alternatively, you could interpolate using some set of nearest neighbors. You would find the nearest neighbors by (say) querying a tree (as in Section 36.3). It is worth the trouble to use a more complex set of samples than a voxel grid. This is because you can concentrate samples where they do the most good, and have few samples in free space **exercises**.

An easy generalization over spherical harmonics is to have the approximation produce a vector of features $\mathbf{f}$. You could obtain a color by, for example, passing this vector into an MLP; alternatively, you could interpret it as a set of spherical harmonic coefficients; or as a set of coefficients on some other basis. Now assume you have values for density α and this feature vector $\mathbf{f}$ produced by an MLP at each vertex of a voxel grid, for each direction in a set of sample directions. If you interpret $\mathbf{f}$ as spherical harmonics, you do not need direction, but in other cases you might. If you have values on the grid points, you can then interpolate those values (as above) to render. You can obtain important speedups by compressing the set of values.

Think of the set of samples as a four dimensional tensor. Write $\mathbf{x}_{ijk}$ for the

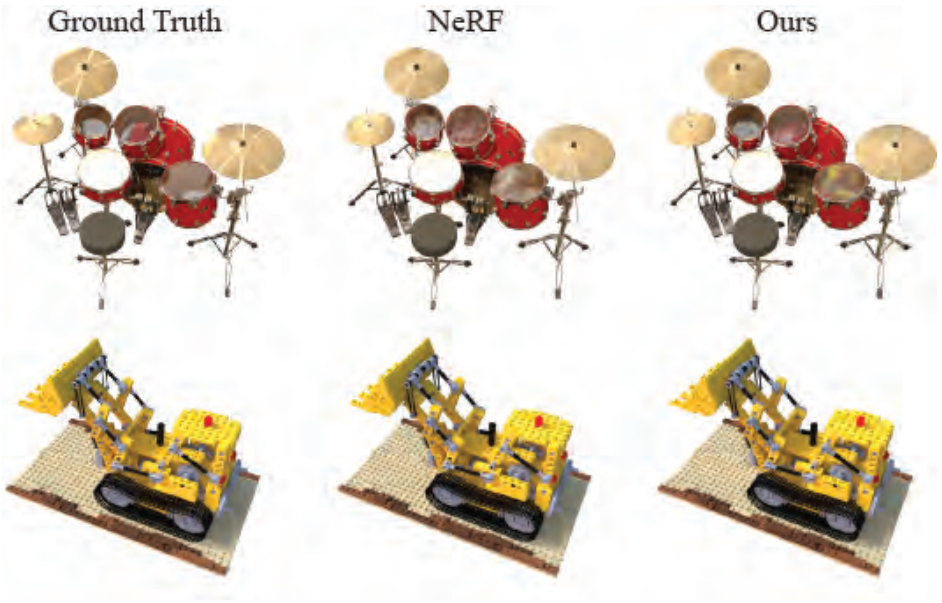


FIGURE 41.10: Image credit *Constructed using part of Figure 4 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*

grid points in 3D and ω_m for the sample directions. Then

$$\alpha_{ijk} = \alpha(\mathbf{x}_{ijk}) \text{ and } \mathbf{f}_{ijkm} = \mathbf{f}(\mathbf{x}_{ijk}, \omega_m).$$

Now use established procedures to compress the tensor representation. A successful compression is *vector-matrix decomposition*. In the case of α, choose some number of approximation components C_α and obtain $3C_\alpha$ vectors and $3C_\alpha$ matrices so that

$$\alpha_{ijk} \approx \sum_{r=1}^{C_\alpha} \left[u_i^{(r,1)} m_{jk}^{(r,1)} + u_j^{(r,2)} m_{ik}^{(r,2)} + u_k^{(r,3)} m_{ij}^{(r,3)} \right].$$

Decomposing $\mathbf{f}_{ijkm}$ requires a little more work. Experience shows that, in the case of $\mathbf{f}$, the following decomposition is successful. Deal with each component separately, so the n'th component of $\mathbf{f}_{ijkm}$ is f_{ijkm}. Write K for the number of components of $\mathbf{f}$. Obtain $6KC_\mathbf{f}$ vectors and $3KC_\mathbf{f}$ matrices so that

$$f_{ijkmn} \approx \sum_{r=1}^{C_\mathbf{f}} \left[u_i^{(n,r,1)} m_{jk}^{(n,r,1)} v_m^{(n,r,1)} + u_j^{(n,r,2)} m_{ik}^{(n,r,2)} v_m^{(n,r,2)} + u_k^{(n,r,3)} m_{ij}^{(n,r,3)} v_m^{(n,r,3)} \right].$$

Now use these compressed representations. TensorRF shows that these approximations yield fast and accurate renderings []. Notice that the renderings are again differentiable in the coefficients, so you can polish the vectors and matrices to produce the best versions of the training images.



FIGURE 41.11: Image credit *Constructed using part of Figure 6 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*

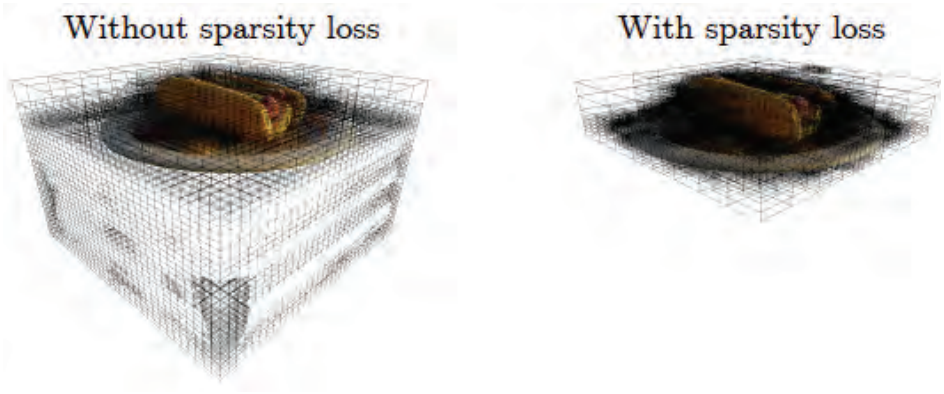


FIGURE 41.12: Image credit *Constructed using part of Figure 3 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*



FIGURE 41.13: Image credit *Constructed using part of Figure 8 of "Plenoxels: Radiance Fields without Neural Networks," Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, Angjoo Kanazawa, CVPR, 2022*



FIGURE 41.14: Image credit *Constructed using part of Figure 2 of "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021*

41.3 EXPLOITING FUNCTION APPROXIMATIONS

To ADD: ClimateNeRF

To ADD: SDF's and MonoSDF

To ADD: SDF from NeRF

41.3.1 X from NeRF

There are now established procedures for recovering other kinds of representation from a NeRF – or NeRF-like – representation. Imagine you wish to convert a NeRF into a surface or set of surfaces. A natural candidate is the collection of surfaces where the density is larger than some small constant, so $\alpha(\mathbf{x}) = \epsilon$. This is an implicit surface as in Section 39.3.6, and so you could use the marching cubes algorithm of that section to obtain a mesh.

It is usual in computer graphics circles to have normals at mesh vertices. This might strike you as odd – the vertices are locations where the surface is clearly not differentiable enough to have a normal – but makes sense when viewed from the perspective of rendering. The object represented by the mesh is smooth. If you know a normal at each mesh vertex, you can compute the shading value for each vertex of a triangle, then use bilinear interpolation to fill in the shading.

You can obtain a normal at each mesh vertex easily. At the vertex $\mathbf{v}_i$, compute

$$-\frac{\nabla_{\mathbf{x}}\alpha(\mathbf{x})}{\|\nabla_{\mathbf{x}}\alpha(\mathbf{x})\|}.$$

It is straightforward to show that this is an outward pointing unit normal **exercises**.

Rendering an RGB image from a NeRF is described in Section 41.1.4. You could obtain a depth map by passing the RGB image into a single image depth



FIGURE 41.15: Image credit *Constructed using part of Figure 11 of "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding", Thomas Müller, Alex Evans, Christoph Schied, Alexander Keller, ACM Trans. Graph., 2022*

predictor. Alternatively, you could render a depth map. To obtain the depth at any pixel, compute the

41.3.2 Other things to do with fnapprox

41.3.3 Instant NGP

One source of improvement is changes in the function approximation. Instant NGP – sometimes known as Instant Nerf – models the color and density with a less complex MLP applied to a much more complex learned positional encoding []. This strategy produces a function approximation that can be evaluated very quickly. In turn, both learning and rendering are fast. The positional encoding is obtained using a multi-resolution hash table. The location $\mathbf{x}$ is hashed to some cell in the table. In that cell is a learned vector that is used as a positional encoding by the MLP.

41.4 RESOURCES

41.5 YOU SHOULD

41.5.1 remember these facts:

41.5.2 remember these procedures:

Training a NeRF model 777
Evaluating a NeRF model 777

41.5.3 exploit these resources:

41.5.4 be able to:

- SOMETHIGN

41.5.5 read these materials:

- "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis, " Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, ECCV 2020
- "PlenOctrees for Real-time Rendering of Neural Radiance Fields," Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, Angjoo Kanazawa, ICCV, 2021
- "Plenoxels: Radiance Fields without Neural Networks," Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, Angjoo Kanazawa, CVPR, 2022
- "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding", Thomas Müller, Alex Evans, Christoph Schied, Alexander Keller, ACM Trans. Graph., 2022
- "pixelNeRF: Neural Radiance Fields from One or Few Images" Alex Yu, Vickie Ye, Matthew Tancik, Angjoo Kanazawa, CVPR 2021

EXERCISES

QUICK CHECKS

41.1.

LONGER PROBLEMS

41.2.