PlenOctree

> PlenOctrees for Real-time Rendering of Neural Radiance Fields(2021)

The Spherical Harmonic Formula

First we need to derive the SH formula:

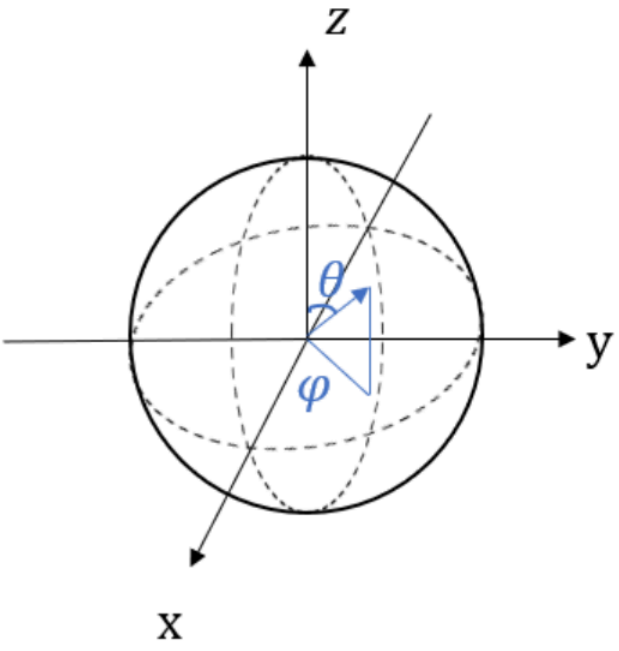
- The overall formula

Spherical harmonics are weighted sums of spherical harmonic bases using a set of spherical harmonic coefficients. The spherical harmonic base is the m^{th} component of the SH Basis of the spherical coordinate system (θ, ϕ), the y_l^m
represents the m^{th} component in the l order:

$$c(d;k) = Sigmoid(\sum_{l=0}^{l_{max}} \sum_{m=-l}^{l} K_l^m y_l^m(d))$$

- Step 1: $d \rightarrow (\theta, \varphi)$

d is the unit vector (x, y, z) in unit spherical coordinates:



$$|\sin\theta|\cos\varphi = x$$
$$|\sin\theta|\sin\varphi = y$$
$$\cos\theta = z$$

- Step 2: the reference blog

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi)P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m \sin(-m\varphi)P_l^{-m}(\cos\theta) & m < 0 \\ K_l^0 P_l^0(\cos\theta) & m = 0 \end{cases}$$

$$P_n(x) = \frac{1}{2^n \cdot n!}\frac{d^n}{dx^n}\left[(x^2-1)^n\right]$$

$$P_l^m = (-1)^m (1-x^2)^{\frac{m}{2}} \frac{d^m}{dx^m}(P_l(x))$$

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

- Step3: Start to substitute different l, m

$$\begin{cases} P_0(x) = 1 \\ P_0^0(x) = 1 \end{cases}$$

$$\begin{cases} P_1(x) = x \\ P_1^0(x) = x \\ P_1^1(x) = -(1-x^2)^{\frac{1}{2}} \end{cases}$$

$$\begin{cases} P_2(x) = \frac{3x^2-1}{2} \\ P_2^0(x) = \frac{2}{3x^2-1} \\ P_2^1(x) = -3x(1-x^2)^{\frac{1}{2}} \\ P_2^2(x) = 3(1-x^2) \end{cases}$$

- Step4: Solving for SH basis, the y_l^m

$$\begin{cases} y_0^0 = K_0^0 \end{cases}$$

$$\begin{cases} y_1^{-1} = -\sqrt{2}K_1^{-1}\sin(\varphi)\sin(\theta) = -\sqrt{2}K_1^{-1}y \\ y_1^0 = K_1^0\cos(\theta) = K_1^0 z \\ y_1^1 = -\sqrt{2}K_1^{-1}\cos(\varphi)\sin(\theta) = -\sqrt{2}K_1^{-1}x \end{cases}$$

$$\begin{cases} y_2^{-2} = 3\sqrt{2}K_2^{-2}\sin(2\varphi)\sin^2\theta = 6\sqrt{2}K_2^{-2}xy \\ y_2^{-1} = -3\sqrt{2}K_2^{-1}\sin(\varphi)\sin(\theta)\cos(\theta) = -3\sqrt{2}yz \\ y_2^0 = \frac{3\cos^2\theta - 1}{2} = K_2^0\frac{2z^2-x^2-y^2}{2} \\ y_2^1 = -3\sqrt{2}K_1^1\cos(\varphi)\sin(\theta)\cos(\theta) = -3\sqrt{2}K_1^1 xz \\ y_2^2 = 3\sqrt{2}K_2^2\cos(2\varphi)\sin^2\theta = 3\sqrt{2}K_2^2(x^2-y^2) \end{cases}$$

- Step5: Solving for K_l^m

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

$$K_0^0 = \frac{1}{\sqrt{4\pi}} = 0.282094$$

$$\sqrt{2}K_1^{-1} = \sqrt{2}K_1^1 = K_1^0 = \sqrt{\frac{3}{4\pi}} = 0.4886$$

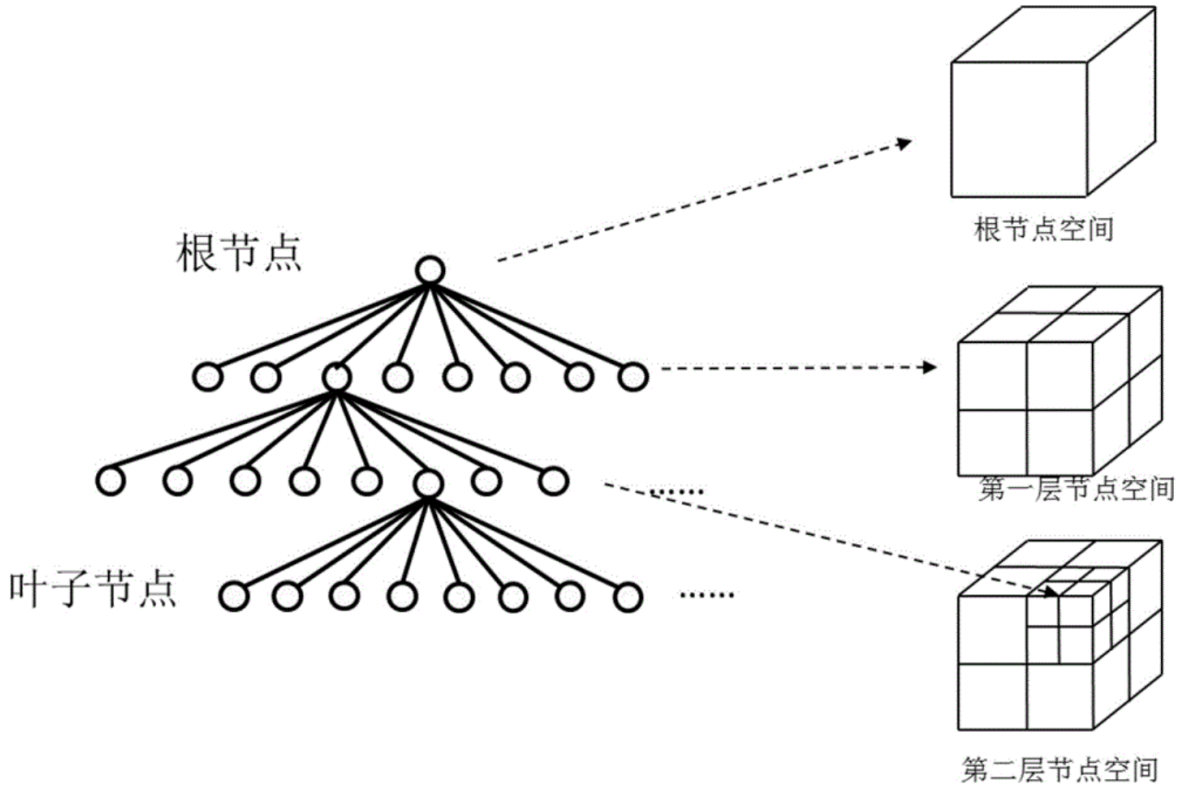
$$\begin{cases} 6\sqrt{2}K_2^{-2} = 1.0925 \\ -3\sqrt{2}K_2^{-1} = -1.0925 \\ K_2^0 = 0.31539 \\ -3\sqrt{2}K_1^2 = -1.0925 \\ 3\sqrt{2}K_2^2 = 0.54627 \end{cases}$$

SH Code in PlenOctree

```
C0 = 0.28209479177387814
C1 = 0.4886025119029199
C2 = [
    1.0925484305920792,
    -1.0925484305920792,
    0.31539156525252005,
    -1.0925484305920792,
    0.5462742152960396
]
C3 = [
    -0.5900435899266435,
    2.890611442640554,
    -0.4570457994644658,
    0.3731763325901154,
    -0.4570457994644658,
    1.445305721320277,
    -0.5900435899266435
]
C4 = [
    2.5033429417967046,
    -1.7701307697799304,
    0.9461746957575601,
    -0.6690465435572892,
    0.10578554691520431,
    -0.6690465435572892,
    0.47308734787878004,
    -1.7701307697799304,
    0.6258357354491761,
]

def eval_sh(deg, sh, dirs):
    """
    Evaluate spherical harmonics at unit directions
    using hardcoded SH polynomials.
    Works with torch/np/jnp.
    ... Can be 0 or more batch dimensions.

    Args:
        deg: int SH deg. Currently, 0-3 supported
        sh: jnp.ndarray SH coeffs [..., C, (deg + 1) ** 2]
        dirs: jnp.ndarray unit directions [..., 3]

    Returns:
        [..., C]
    """
    assert deg <= 4 and deg >= 0
    assert (deg + 1) ** 2 == sh.shape[-1]
    C = sh.shape[-2]

    result = C0 * sh[..., 0]
    if deg > 0:
        x, y, z = dirs[..., 0:1], dirs[..., 1:2], dirs[..., 2:3]
        result = (result -
```

```
                C1 * y * sh[..., 1] +
                C1 * z * sh[..., 2] -
                C1 * x * sh[..., 3])
        if deg > 1:
            xx, yy, zz = x * x, y * y, z * z
            xy, yz, xz = x * y, y * z, x * z
            result = (result +
                    C2[0] * xy * sh[..., 4] +
                    C2[1] * yz * sh[..., 5] +
                    C2[2] * (2.0 * zz - xx - yy) * sh[..., 6] +
                    C2[3] * xz * sh[..., 7] +
                    C2[4] * (xx - yy) * sh[..., 8])

            if deg > 2:
                result = (result +
                        C3[0] * y * (3 * xx - yy) * sh[..., 9] +
                        C3[1] * xy * z * sh[..., 10] +
                        C3[2] * y * (4 * zz - xx - yy)* sh[..., 11] +
                        C3[3] * z * (2 * zz - 3 * xx - 3 * yy) * sh[..., 12] +
                        C3[4] * x * (4 * zz - xx - yy) * sh[..., 13] +
                        C3[5] * z * (xx - yy) * sh[..., 14] +
                        C3[6] * x * (xx - 3 * yy) * sh[..., 15])
                if deg > 3:
                    result = (result + C4[0] * xy * (xx - yy) * sh[..., 16] +
                            C4[1] * yz * (3 * xx - yy) * sh[..., 17] +
                            C4[2] * xy * (7 * zz - 1) * sh[..., 18] +
                            C4[3] * yz * (7 * zz - 3) * sh[..., 19] +
                            C4[4] * (zz * (35 * zz - 30) + 3) * sh[..., 20] +
                            C4[5] * xz * (7 * zz - 3) * sh[..., 21] +
                            C4[6] * (xx - yy) * (7 * zz - 1) * sh[..., 22] +
                            C4[7] * xz * (xx - 3 * yy) * sh[..., 23] +
                            C4[8] * (xx * (xx - 3 * yy) - yy * (3 * xx - yy)) *
sh[..., 24])
    return result
```

What is octree?



Sparsity prior

Sparsity prior. Without any regularization, the model is free to generate arbitrary geometry in unobserved regions. While this does not directly worsen image quality, it would adversely impact our conversion process as the extra geometry occupies significant voxel space.

To solve this problem, we introduce an additional sparsity prior during NeRF training. Intuitively, this prior encourages NeRF to choose empty space when both space and solid colors are possible solutions. Formally,

$$\mathcal{L}_{\text{sparsity}} = \frac{1}{K}\sum_{k=1}^{K}|1-\exp(-\lambda\sigma_k)| \tag{6}$$

Here, $\{\sigma_k\}_{k=1}^{K}$ are the evaluated density values at K uniformly random points within the bounding box, and λ is a hyperparameter. The final training loss is then $\beta_{\text{sparsity}}\mathcal{L}_{\text{sparsity}} + \mathcal{L}_{\text{RGB}}$, where β_{sparsity} is a hyperparameter. Fig. 3 illustrates the effect of the prior.

I think that this sparsity prior is very interesting.

Define

The sparsity prior is defined as:

$$\mathcal{L}_{\text{sparsity}} = \frac{1}{K}\sum_{k=1}^{K}|1-\exp(-\lambda\sigma_k)|$$

- K represents the number of random points sampled uniformly within the bounding box.
- $\sigma_k$is the density value at the (k)-th point, which indicates whether that point is part of an object or not.
- $lambda$ is a hyperparameter controlling the strength of the prior.
- This loss encourages the model to reduce density in regions that don't need it, effectively promoting sparse density maps.

The term $1-\exp(-\lambda\sigma_k)$behaves as a sparsity-inducing function:

- When the density $sigma_k$ is small (close to 0), $\exp(-\lambda\sigma_k) \approx 1$, so $1-\exp(-\lambda\sigma_k) \approx 0$, meaning the point does not contribute much to the loss.

- When the density σ_k is large, $\exp(-\lambda\sigma_k)$ approaches 0, and $1 - \exp(-\lambda\sigma_k)$ approaches 1. This penalizes high-density areas, encouraging the model to reduce density where it's unnecessary.

Loss Function

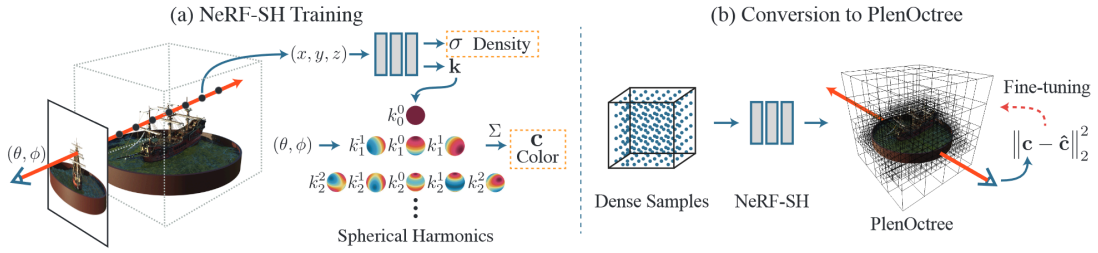
The total training loss combines the sparsity loss with the standard RGB loss:

$$\beta_{\text{sparsity}}\mathcal{L}_{\text{sparsity}} + \mathcal{L}_{\text{RGB}}$$

- $\beta_{\text{sparsity}}$is a hyperparameter that controls the weight of the sparsity prior in the overall loss function.
- $mathcal L_{\text{RGB}}$ is the standard color loss, ensuring that the model correctly predicts the colors in the scene.

By combining these two terms, the model is encouraged to choose sparse representations of the scene, correctly representing both the empty space and the solid objects.

Otree



1.模型转换为稀疏八叉树表示

训练好的 NeRF-SH 模型可以转换为稀疏八叉树（PlenOctree）结构，用于实时渲染。这个八叉树在每个叶子节点存储密度（density）和球谐系数（SH coefficients），以模拟视图相关的外观。

2. 渲染过程

- 渲染 PlenOctree 时，对于每条光线，首先确定光线与八叉树中体素（Voxel）的相交点。
- 这会生成一系列体素边界之间的片段（segments），这些片段有固定的密度和颜色，长度为 $\{\delta_i\}_{i=1}^{N}$。
- 然后，应用 NeRF 的体积渲染模型给光线赋予颜色。这样可以在跳过大体素的同时不遗漏小体素。

优化方法：

- 在测试时，可以通过早停（early-stopping）来加速渲染过程，当光线的透射率 T_i 小于 $\gamma = 0.01$ 时终止光线的处理。

3. 从 NeRF-SH 到 PlenOctree 的转换

转换过程分为三个步骤：

- 第一步：在高层次上，首先在网格上评估网络，仅保留密度值。
- 第二步：对体素应用阈值处理进行过滤。
- 第三步：在每个保留的体素中采样随机点，并将这些点的 SH 系数进行平均后存储到八叉树叶子节点中。

4. 评估过程

- 首先评估 NeRF-SH 模型以获得均匀分布在 3D 网格中的密度 σ值。
- 网格自动缩放以紧密适应场景内容。

Rendering

Rendering. To render the PlenOctree, for each ray, we first determine ray-voxel intersections in the octree structure. This produces a sequence of segments between voxel boundaries with constant density and color, of lengths $\{\delta_i\}_{i=1}^{N}$. NeRF's volume rendering model (1) is then applied to assign a color to the ray. This approach allows for skipping large voxels in one step while also not missing small voxels.

At test-time, we further accelerate this rendering process by applying early-stopping when the ray has accumulated transmittance T_i less than $\gamma = 0.01$.

渲染过程的具体解释如下：

1. 光线与体素相交（Ray-Voxel Intersections）

当我们对 PlenOctree 进行渲染时，需要处理每条光线与体素（Voxel）之间的相交情况。光线会穿过场景中的多个体素，每个体素表示该区域内的颜色和密度。

- 光线遍历八叉树：八叉树是一种递归细分空间的结构，体素是八叉树的节点。在渲染时，光线会与八叉树的体素发生相交，依次遍历体素。通过八叉树结构，我们可以快速确定光线与哪些体素相交，并计算光线在每个体素中穿过的距离。
- 体素分段：光线穿过的每个体素可以分为多个片段（segments），这些片段具有相同的密度和颜色。每个片段的长度为δ_i，表示光线在该片段中穿过的距离。有两种常见的分段策略：均匀分段：将光线在体素中的路径均匀分为若干个片段，这样可以保证体素中所有位置的密度和颜色都被考虑到。自适应分段：当体素密度或颜色较为均匀时，可以减少分段次数，直接用一个大片段来表示该体素。如果密度或颜色变化剧烈，则可以增加分段次数，以更精细地表示体素的变化情况。每个片段的密度和颜色由 NeRF 模型预测得到。对于每个体素的每个分段，体积渲染公式使用以下步骤计算其贡献。体积密度：每个片段的密度决定了该片段是否为透明或半透明区域。密度越高，表示光线越可能被吸收，颜色变化越大。颜色权重：根据光线穿过该片段的长度和密度，计算颜色的加权贡献。颜色的加权过程考虑到光线的累积透明度，越靠近光源的体素对颜色的贡献越大，而更远的体素由于光线的吸收，贡献会较小。

2. 应用 NeRF 的体积渲染模型

接下来，使用 NeRF 的体积渲染模型对这些相交片段进行处理，以生成最终的颜色。这个过程可以分为以下几个步骤：

- 体积密度的累积计算：NeRF 体积渲染公式会沿光线积累体积密度（density），并通过体积传输函数计算光线传递过程中的颜色变化。公式可以表示为：

  $$T_i = \exp\left(-\int_0^i \sigma(t)dt\right)$$

  其中 (T_i) 是透射率，表示光线从相机到当前体素的透明度；$\sigma(t)$ 是光线经过的体素的密度。密度越大，光线的透射率越低，意味着该体素可能是一个实体物体。

- 颜色累积：通过 NeRF 的体积渲染模型，将每个体素中的颜色按密度加权并累积，最终得到该条光线的颜色值。光线经过不同体素的颜色会累积起来，形成最终的图像。

3. 跳过大体素，精细处理小体素

PlenOctree 的优势在于，它能够根据体素的大小来灵活地跳过不必要的细节：

- 跳过大体素：如果光线与某个大体素相交，并且该体素内的密度和颜色一致，系统可以直接处理整个大体素的颜色，不需要进一步细分。这减少了计算量，加速了渲染过程。
- 处理小体素：对于较小的体素，系统则会逐个精细处理，以确保不会遗漏重要的细节。这样可以在保持高质量图像的同时，避免不必要的计算。

4. 测试时的早停机制

在测试过程中，为了进一步加快渲染速度，使用了早停机制（early-stopping）：

- 透射率终止条件：当光线在遍历体素时，如果累积的透射率 T_i 小于某个阈值$\gamma = 0.01$，就会提前停止该光线的处理。这是因为当透射率非常低时，表示光线已经被遮挡或吸收，后续的体素对图像颜色的影响很小，因此不再需要继续计算。

从 NeRF-SH 到 PlenOctree 的转换

Conversion from NeRF-SH. The conversion process can be divided into three steps. At a high level, we evaluate the network on a grid, retaining only density values, then filter the voxels via thresholding. Finally we sample random points within each remaining voxel and average them to obtain SH coefficients to store in the octree leaves. More details are given below:

Evaluation. We first evaluate the NeRF-SH model to obtain σ values on a uniformly spaced 3D grid. The grid is automatically scaled to tightly fit the scene content.[2]

Filtering. Next, we filter this grid to obtain a sparse set of voxels centered at the grid points sufficient for representing the scene. Specifically, we render alpha maps for all the training views using this voxel grid, keeping track of the maximum ray weight $1 - \exp(-\sigma_i \delta_i)$ at each voxel. We then eliminate the voxels whose weights are lower than a threshold τ_w. The octree is constructed to contain the re-

maining voxels as leaves at the deepest level while being empty elsewhere. Compared to naively thresholding by σ at each point, this method eliminates non-visible voxels.

Sampling. Finally, we sample a set of 256 random points in each remaining voxel and set the associated leaf of the octree to the mean of these values to reduce aliasing. Each leaf now contains the density σ and a vector of spherical harmonics coefficients for each of the RGB color channels.

This full extraction process takes about 15 minutes.[3]

从 NeRF-SH 模型到 PlenOctree的转换过程可以分为三个主要步骤，目的是将 NeRF-SH 模型转换为适合实时渲染的稀疏八叉树结构（PlenOctree）。每个叶子节点存储密度值和球谐函数（SH）系数，用于表示方向相关的外观。下面是具体的转换步骤：

1. 在网格上评估 NeRF-SH 模型

首先，使用 NeRF-SH 模型对场景进行评估。这里采用的是一个均匀分布的 3D 网格，该网格会遍历整个场景空间。对于网格中的每个点，NeRF-SH 模型会生成密度值σ：

- 密度值σ：这些密度值表示每个网格点处的体积密度，或说该点是否属于某个物体的一部分。通过这些密度值，可以初步识别场景中的空白区域和实物区域。
- 自适应缩放：在评估过程中，网格会根据场景的大小自动缩放，以确保模型能够捕捉到场景中的所有细节。

2. 通过阈值过滤体素

评估完成后，接下来对八叉树结构中的体素进行过滤：

- 阈值处理：使用预设的阈值对体素进行筛选。如果某个体素的密度值σ低于该阈值，则认为该体素代表的是空白区域，可以丢弃掉。如果密度值较高，则该体素会被保留，因为它可能包含场景中的实体部分。这个过程的目的是剔除无关的体素，以构建更加稀疏且高效的八叉树结构，减少不必要的计算量。

3. 采样并计算球谐系数

对于保留下来的体素，接下来要计算每个体素的球谐系数（SH coefficients），这些系数用于表示该体素内的方向性光照信息。

- 随机点采样：在每个体素中随机采样多个点。对于这些采样点，NeRF-SH 模型会生成相应的球谐系数，用来表示该点的方向相关外观（例如光照、颜色等）。
- 系数平均：在每个体素内，对采样点的球谐系数进行平均，并将结果存储到该体素的八叉树叶子节点中。这些系数帮助渲染时根据不同视角生成正确的颜色和亮度信息。