

# NeRF

---

## 一、NeRF基础

---

### 1.1 模型不具备泛化能力

---

NeRF的训练方式决定了它是**场景特定的**

- NeRF在训练时，需要用到**同一场景的多张图像**（带相机姿态），通过优化一个神经网络来隐式表示该场景的体积密度和辐射度函数。
- 训练完成后，NeRF网络的参数实际上只编码了**这个场景的信息**，无法直接迁移到另一个场景。  
👉 这意味着每一个新场景都要重新训练一个NeRF。

### 1.2 粒子的采集-光线代码

---

- `dirs`是归一化的相机坐标，是将图像像素坐标系转换为相机坐标系，也就是乘以一个内参矩阵`K`的逆

### 1.3 输入与输出

---

#### 粒子的空间位姿 (`x, y, z, theta, phi`)

- `(x, y, z)`：表示粒子在3D空间中的位置坐标。
- `(theta, phi)`：表示视角方向（即观察角度），通常用球坐标系表示：
  - `theta` 是方位角 (azimuth)，表示在水平面上的角度。
  - `phi` 是俯仰角 (elevation)，表示在垂直面上的角度。
- **位姿 (Pose)**：在这里指的是粒子在空间中的位置和观察方向的组合，描述了粒子的空间状态。

#### 输出是4D向量：颜色和密度 (`RGB, sigma`)

- `RGB`：表示粒子的颜色，是一个3D向量，分别对应红、绿、蓝三个通道。
- `sigma`：表示粒子的密度（或称为不透明度），是一个标量值：
  - 密度决定了光线在穿过该粒子时被吸收或散射的概率。
  - 密度越高，表示粒子越“实心”，光线越容易被阻挡。
  - 密度越低，表示粒子越“透明”，光线越容易穿过。
- **密度的物理意义：**在NeRF中，密度用于计算光线在空间中传播时的累积透射率，从而渲染出最终的图像。

## 二、输入粒子采样

---

### 2.1 粒子采样原理

---

对于图片上的某一个像素( $u, v$ )，沿着某一条射线上的无数个发光点的“和”， $O$ 为射线原点， $d$ 为方向， $t$ 为距离

射线表示为：  $r(t) = o + td$

方法：

- 设置 `near=2, far=6`
- 在 `near` 和 `far` 之间均匀采样 64 个点
  - $t_i = U[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)]$

### 2.2 前处理过程得到粒子

---

“从图片和相机位姿计算射线 从射线上采样粒子”

---

## 1. 基本原理

- **目标**: 从2D图像中的每个像素点出发，计算其在3D空间中对应的光线。
  - **方法**: 通过相机位姿（位置和方向）以及相机的内参矩阵，将2D像素点映射到3D空间中的光线。
- 

## 2. 步骤

### (1) 从像素坐标系到归一化相机坐标系

- **像素坐标系**: 图像中的像素点用  $(u, v)$  表示。
- **归一化相机坐标系**: 通过内参矩阵  $K$  将像素点  $(u, v)$  转换为归一化相机坐标系中的点  $(x_n, y_n)$ 。
- **归一化相机坐标系中的点**:  $(x_n, y_n, 1)$  表示光线在相机坐标系中的方向。

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

### (2) 从归一化相机坐标系到相机坐标系

- **相机坐标系**: 以相机为原点的3D坐标系。
- **光线方向**: 归一化相机坐标系中的点  $(x_n, y_n, 1)$  直接表示光线在相机坐标系中的方向向量  $(x_n, y_n, 1)$ 。

### (3) 从相机坐标系到世界坐标系

- **相机位姿**: 包括相机的位置  $t$  和旋转矩阵  $R$ 。
- **光线在世界坐标系中的表示**:
  - 光线起点: 相机在世界坐标系中的位置  $t$ 。
  - 光线方向: 将相机坐标系中的方向向量  $(x_n, y_n, 1)$  转换为世界坐标系中的方向向量:

$$[\mathbf{d} = R [x_n \ y_n \ 1]]$$

#### (4) 射线的参数化表示

- **射线方程：**射线可以用参数方程表示：

$$[\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}]$$

- $\mathbf{o}$ ：光线起点（相机位置）。
- $\mathbf{d}$ ：光线方向。
- $t$ ：参数，表示沿着光线的距离。

### 3. 具体应用：从射线上采样粒子

- **目标：**在射线上采样一系列点（粒子），用于后续的渲染或重建。
- **方法：**

- 在射线上均匀或非均匀地采样参数  $t$ ，得到一系列点。
- 例如，在  $t$  的范围内采样64个点，得到64个粒子。

### 4. 批量处理

- **一张图片的处理：**
  - 从图片中取1024个像素，计算1024条射线。
  - 每条射线上采样64个粒子，共 $1024 * 64$ 个粒子。
- **批量输入：**
  - 将这些粒子以batch形式输入模型，例如形状为  $(1024 * 64, 3)$  的张量。

### 5. 总结

- **根据图片和相机位姿计算射线**: 通过内参矩阵和相机位姿, 将2D像素点映射到3D空间中的光线。
- **射线的参数化表示**: 射线可以用参数方程  $r(t) = o + t * d$  表示。
- **采样粒子**: 在射线上采样一系列点(粒子), 用于后续的计算或渲染。
- **批量处理**: 将粒子以batch形式输入模型, 提高计算效率。

通过这种方法, 可以从2D图像中的像素点反推出其在3D空间中的对应光线, 并进一步用于渲染、重建或其他任务。

## 三、材质

在计算机图形学、计算机视觉和物理渲染中, **材质 (Material)** 是描述物体表面如何与光线交互的属性集合。材质决定了物体在光照下的外观, 包括颜色、反射、折射、粗糙度等特性。理解材质的概念对于模拟真实世界的光照效果至关重要。

### 3.1 材质 (Material)

- **定义**: 材质是物体表面的物理和光学属性, 决定了光线如何与物体表面交互。
- **关键属性**:
  - **颜色 (Color)** : 物体表面的基础颜色。
  - **反射率 (Reflectance)** : 物体表面反射光线的能力。
  - **粗糙度 (Roughness)** : 物体表面的光滑程度, 影响反射光线的散射。
  - **透明度 (Transparency)** : 物体允许光线通过的程度。
  - **折射率 (Refractive Index)** : 光线穿过物体时方向改变的程度。
  - **光泽度 (Glossiness)** : 物体表面的镜面反射强度。
  - **金属度 (Metallic)** : 物体是否具有金属特性, 影响反射光的颜色和强度。

### 3.2 物理材质 (Physically Based Material, PBR)

- **定义**: 物理材质是一种基于物理定律的材质模型, 旨在更真实地模拟光线与物体表面的交互。

- **核心思想：**

- 材质的属性应基于物理定律（如能量守恒、菲涅尔反射等）。
- 材质的外观在不同光照条件下应保持一致。

- **关键特性：**

- **能量守恒**：物体反射的光线能量不应超过入射光线的能量。
  - **菲涅尔反射 (Fresnel Reflection)**：反射率随观察角度的变化而变化。
  - **微表面理论 (Microsurface Theory)**：物体表面由许多微小的几何细节组成，影响光线的散射。
- 

### 3.3 材质的建模难度

- **复杂性**：材质的属性涉及多种物理现象（如反射、折射、散射等），建模时需要综合考虑这些现象。
  - **数据获取**：真实世界中的材质属性需要通过测量或实验获取，例如使用光谱仪测量反射率。
  - **计算开销**：基于物理的材质模型通常需要复杂的计算，尤其是在实时渲染中。
  - **多样性**：不同材质（如金属、塑料、木材等）具有不同的光学特性，需要针对性地建模。
- 

### 3.4 常见材质类型

- **漫反射材质 (Diffuse Material) :**

- 光线在表面均匀散射，无镜面反射。
- 例如：纸张、布料。

- **镜面反射材质 (Specular Material) :**

- 光线在表面发生镜面反射，形成高光。
- 例如：金属、镜子。

- **透明材质 (Transparent Material) :**

- 光线可以穿过物体，可能发生折射。

- 例如：玻璃、水。

- 次表面散射材质 (Subsurface Scattering Material) :

- 光线进入物体表面后在其内部散射，然后从其他位置射出。
  - 例如：皮肤、大理石。
- 

## 3.5 材质与光照的关系

---

- 材质的外观高度依赖于光照条件。例如：
    - 在强光下，金属材质会表现出明显的高光。
    - 在漫射光下，粗糙材质会显得更加柔和。
  - 渲染时，材质和光照模型需要紧密结合，以模拟真实的光照效果。
- 

## 总结

- **材质**：描述物体表面如何与光线交互的属性集合。
- **物理材质**：基于物理定律的材质模型，旨在更真实地模拟光线与物体表面的交互。
- **建模难度**：材质建模涉及复杂的物理现象、数据获取和计算开销。
- **应用**：材质建模在计算机图形学、计算机视觉和物理渲染中具有广泛的应用，是生成逼真图像和场景的关键技术。

# 四、图像处理和转换

---

## 4.1 相机标定

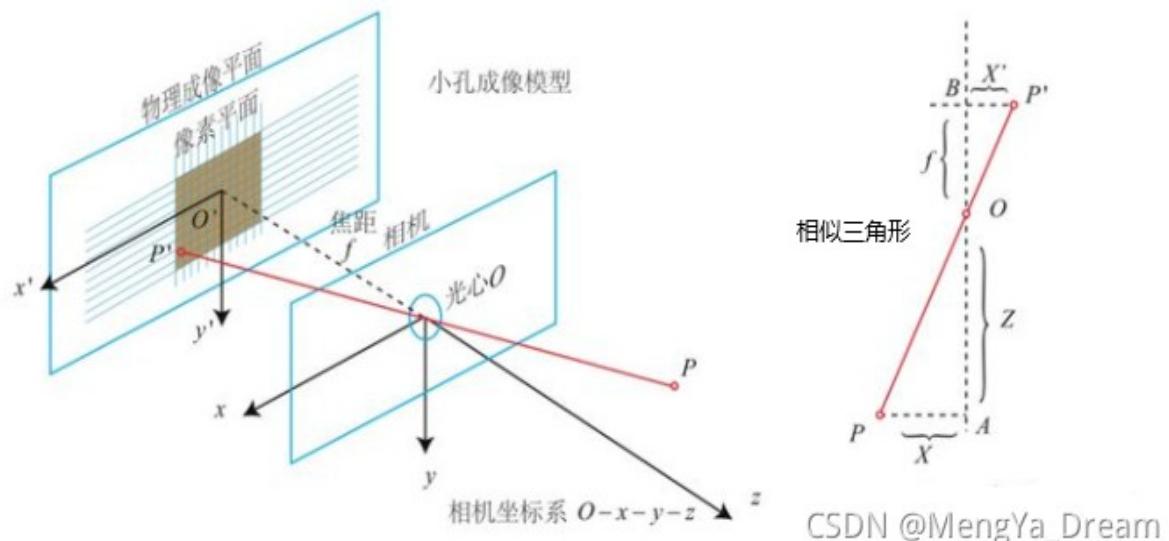
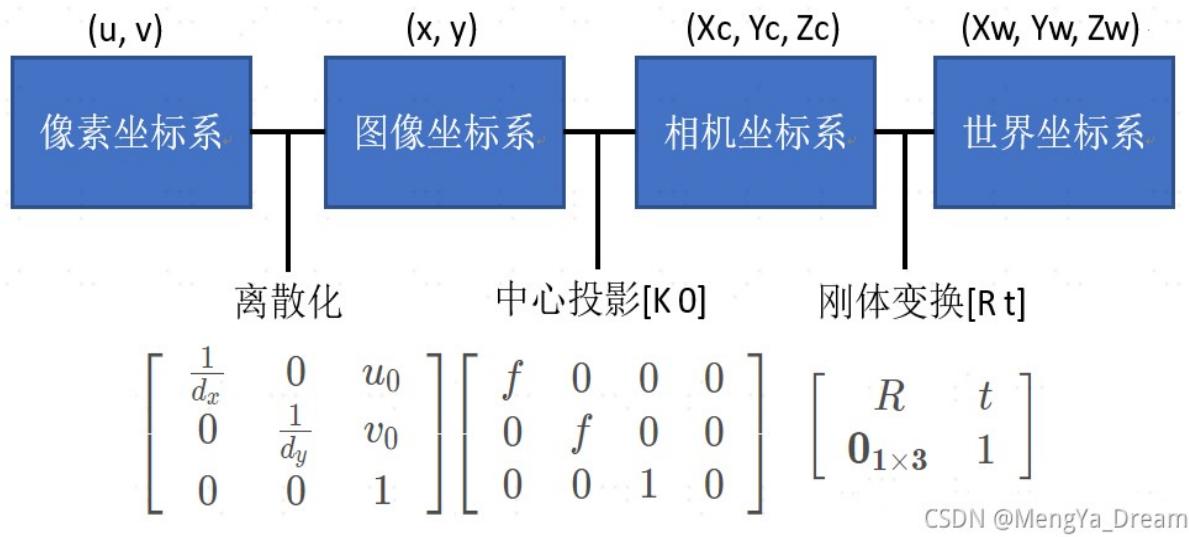
---

**相机标定**：求解参数的过程称为。因为描述相机的几何成像关系、需要进行**数学建模**，这些几何参数就是相机参数，包括内参和外参。

相机模型是计算机视觉中广泛使用的**针孔模型** (The basic pinhole model)。该模型在数学上是三维空间到二维平面 (**image plane or focal plane**) 的中心投影：

- 由一个 $3 \times 4$ 的投影矩阵  $P = K [R|t]^T$  来描述，
- $K$ 为相机内参 (internal camera parameters)， $[R|t]$ 为外参 (external

parameters)。

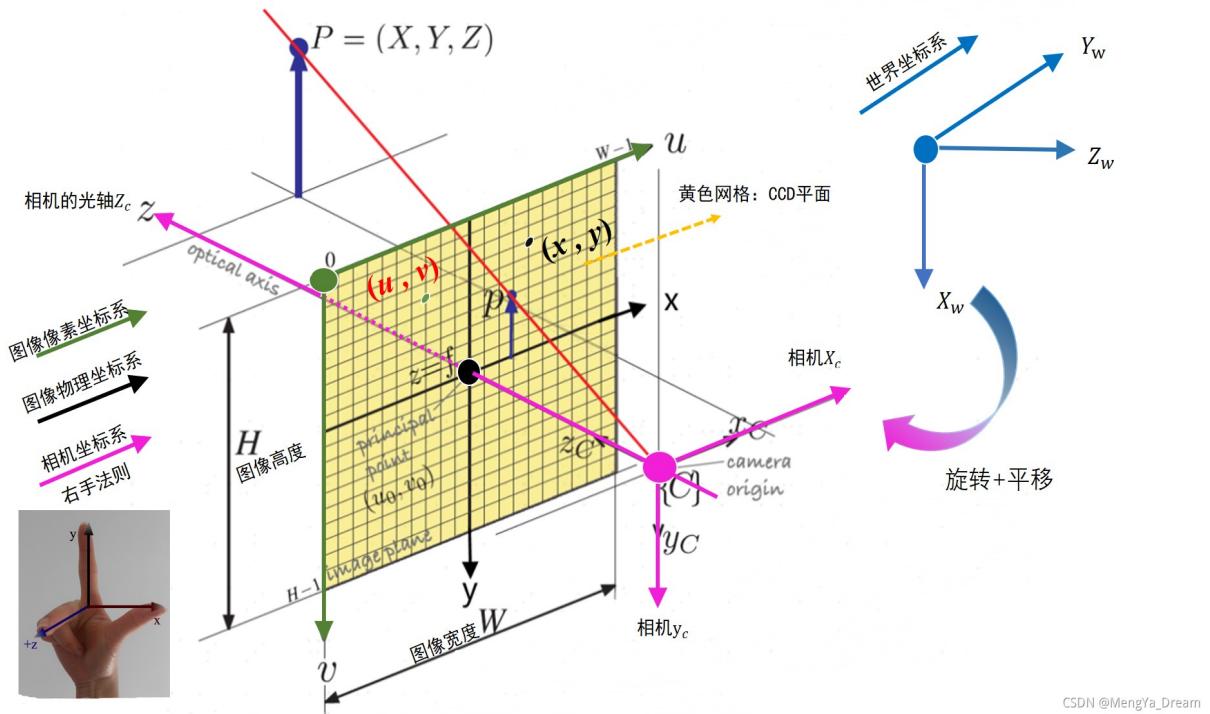


## 4.2 四个坐标系

- **图像像素坐标系:** 表示三维空间物体在图像平面上的投影, **像素是离散化的**, 其坐标原点在CCD图像平面的左上角,  $u$ 轴平行于CCD平面水平向右,  $v$ 轴垂直于 $u$ 轴向下, 坐标使用 $(u, v)$ 来表示。图像宽度W, 高度H。
- **图像物理坐标系:** 坐标原点在CCD图像平面的中心,  $x, y$ 轴分别平行于图像像素坐标系的 $(u, v)$ 轴, 坐标用 $(x, y)$ 表示。
- **相机坐标系:** 以相机的光心为坐标系原点,  $X_c, Y_c$ 轴平行于图像坐标系的 $x, y$ 轴, 相机的光轴为 $Z_c$ 轴, 坐标系满足右手法则。相机的光心可理解为相机透镜的几何中

心。

- **世界坐标系**: 用于表示空间物体的绝对坐标, 使用 $(X_w, Y_w, Z_w)$ 表示, 世界坐标系可通过旋转和平移得到相机坐标系。



CSDN @MengYa\_Dream

## 4.3 世界坐标系-相机坐标系

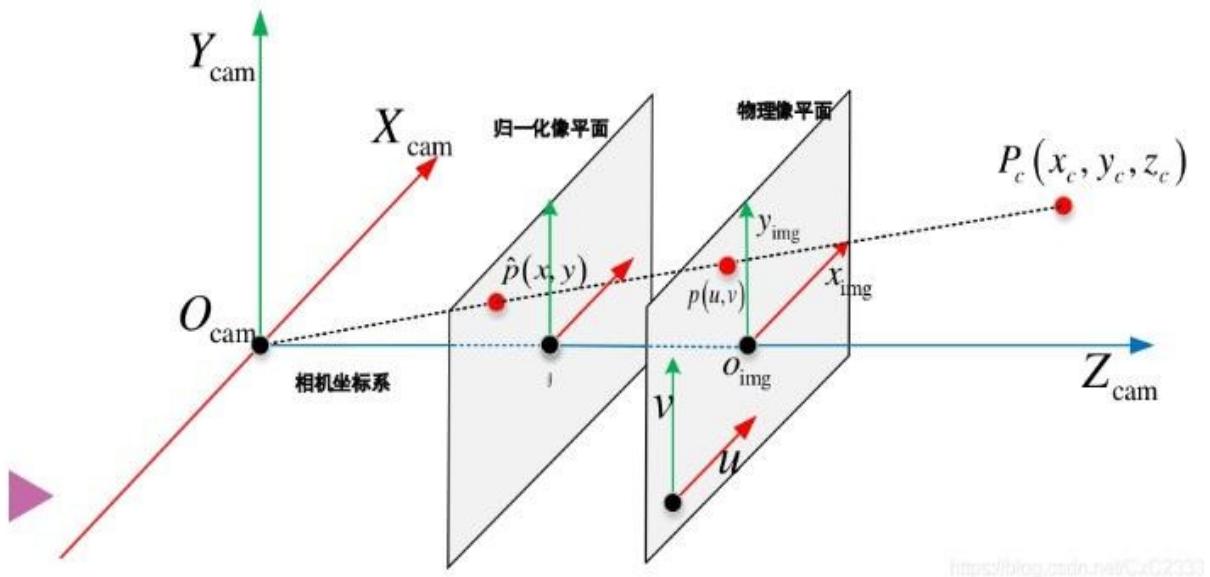
平移比较好理解, 世界坐标系原点移动到相机坐标系原点;

旋转一共有3个自由度, 及绕x,y,z旋转, 根据旋转角度可以分别得到三个方向上的旋转矩阵 $R_x, R_y, R_z$ , 而旋转矩阵即为他们的乘积,  $R = R_x \times R_y \times R_z$

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

## 4.4 相机坐标系-归一化坐标系

## 相机坐标系——>归一化平面



<https://blog.csdn.net/CxG2333>

归一化平面与相机坐标只相差了一个 $Z_c$ :

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{z_c} \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix}$$

知乎 @骆驼

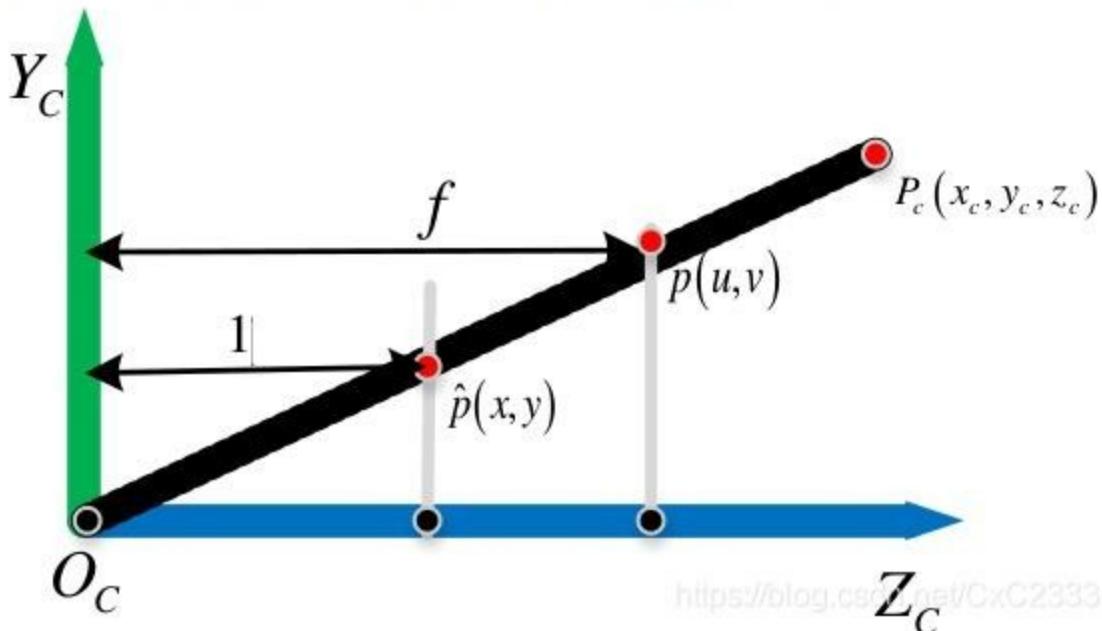
## 4.5 归一化坐标系-图像坐标系

物理平面实际上在光心后面 ( $Z$ 轴的负半轴方向上)，但实际中一般可以将其放到光心前面来考虑 ( $Z$ 轴的正半轴方向上)。归一化平面与物理平面有什么不同呢？

首先，他们都是一个平面，且都是三维点投影过来成像。

唯一的不同是：归一化平面位于 $Z=1$ 的位置；物理平面位于 $Z=f$ 的位置。

显而易见，他们仅仅相差一个倍数  $f$ 。于是直接上相似三角形：



<https://blog.csdn.net/CxC2333>

很显然，很直接的一个倍数关系而已，举个例子：

对于归一化平面上的坐标 $(x, y)$ 、物理平面上的坐标 $(u, v)$

$u = f * x$  不能直接将坐标轴的Z坐标丢掉

$v = f * y$

写成矩阵形式：

$[u]$

$[v] =$

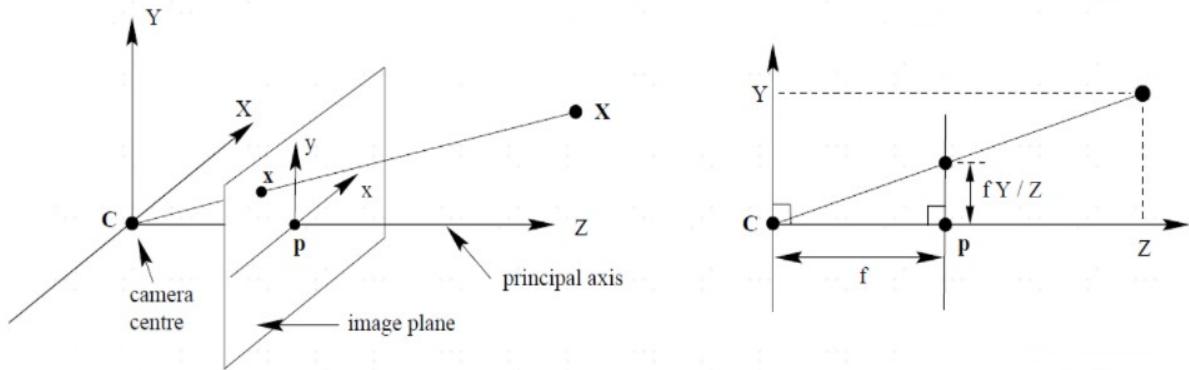
$[1]$

$[f \ 0 \ 0][x]$

$[0 \ f \ 0][y]$

$[0 \ 0 \ 1][1]$

知乎 @骆驼



$$\begin{cases} \frac{x}{f} = \frac{X_C}{Z_C} \\ \frac{y}{f} = \frac{Y_C}{Z_C} \end{cases}$$

写成齐次坐标形式的矩阵相乘为,

$$Z_C \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = [K|0] \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \quad (2)$$

其中  $K$  称为相机内参数矩阵。

CSDN @MengYa\_Dream

## 4.5 图像坐标系-像素坐标系

像素坐标系是图像坐标系的离散化表示，实际CCD相机每个像素对应一个感光点，是个矩形，假设其物理尺寸为  $dx$  宽， $dy$  高。以CCD传感器的左上角为坐标原点建立的坐标系与以成像平面中心建立的坐标系的转换关系

二者有一个重要的区别：

- 1、像素平面的基本单位是pixel；
- 2、物理平面的基本单位是mm。

因此二者的转换需要解决两个点：

- (1)基本单位的转换 **使用dx和dy实现**
- (2)原点坐标的平移 **使用u0和v0实现**

转换其实也很简单。

其实就是要知道，**像素平面上的一个像素，对应了实际物理尺寸（物理平面）的多少毫米。**

物理尺寸，举个栗子：

$dx=2\text{mm}/\text{像素}$

$dy=3\text{mm}/\text{像素}$  ( $dx$ ,  $dy$ 就是转换的度量)

对于物理平面上的某个点 $(X, Y)=(10\text{mm}, 33\text{mm})$

$10\text{mm} / (2\text{mm}/\text{像素}) = 5\text{像素}$

$33\text{mm} / (3\text{mm}/\text{像素}) = 11\text{像素}$

得： $(u, v) = (5+u_0, 10+v_0)$

设物理平面上的点坐标为  $(X, Y)$ ；像素平面上点的坐标是  $(u, v)$

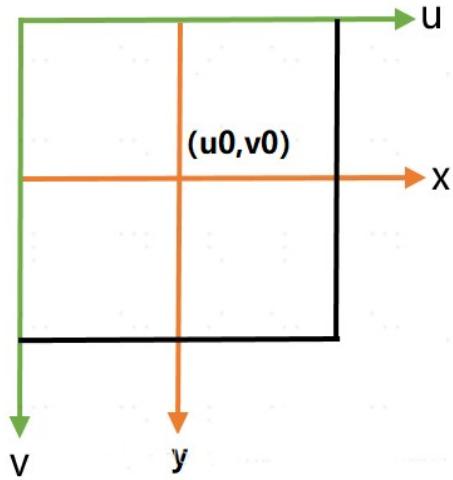
写成公式：

$$\begin{cases} u = \frac{X}{dX} + u_0 \\ v = \frac{Y}{dY} + v_0 \end{cases}$$

写成矩阵形式：

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{dX} & 0 & u_0 \\ 0 & \frac{1}{dY} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

知乎 @骆驼



$$\begin{cases} u = \frac{x}{d_x} + u_0 \\ v = \frac{y}{d_y} + v_0 \end{cases}$$

齐次坐标矩阵形式为,

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{d_x} & 0 & u_0 \\ 0 & \frac{1}{d_y} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (3)$$

CSDN @MengYa\_Dream

## 4.6 最终公式总结

---

.从点P的世界坐标系(Xw, Yw, Zw)到像素坐标(u, v)的转化流程。

1) 世界坐标Pw; 2) 相机坐标Pc; 3) 归一化平面坐标; 4) 物理平面坐标; 5) 像素平面坐标p

- $f_x$  和  $f_y$  分别是相机在x和y方向上的焦距 (以像素为单位)

根据公式(1)(2)(3)可得,

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{d_x} & 0 & u_0 \\ 0 & \frac{1}{d_y} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

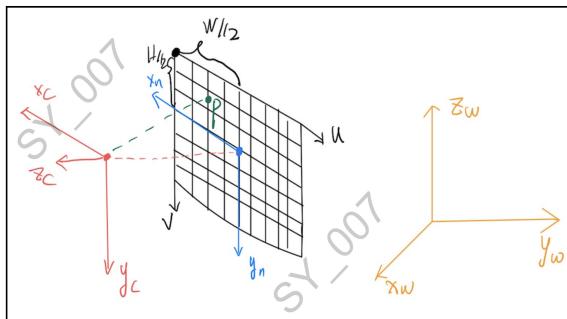
其中,

$\begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$  为相机内参矩阵,  $\begin{bmatrix} R & t \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix}$  为外参矩阵。相机标定就是为了求解这两个矩阵的参数。

CSDN @MengYa\_Dream

## 4.7 像素点反推射线

5



从图片获得射线

由像素点 \$P(u, v)\$ 反推射线:

- 像素平面 \$\rightarrow\$ 物理成像平面 \$(x\_n, y\_n) = (- (u - w/2), v - h/2)\$
- 物理成像平面 \$\rightarrow\$ 相机坐标系 \$(x\_c, y\_c, z\_c) = (x\_n, y\_n, -f)\$
- 归一化 \$(x\_c, y\_c, z\_c) = (\frac{x\_c}{f}, \frac{y\_c}{f}, -1)\$
- 相机坐标系 \$\rightarrow\$ 世界坐标系 \$(x\_w, y\_w, z\_w) = c2w \* (x\_c, y\_c, z\_c)\$

## 4.8 相机位姿

### (1) 定义

- 相机位姿描述了相机在 3D 世界坐标系中的位置和方向。
- 通常用一个 旋转矩阵 \$R\$ 和一个 平移向量 \$t\$ 表示。

## (2) 数学表示

- **旋转矩阵 R**: 一个  $3 \times 3$  的正交矩阵，描述相机的方向（即相机坐标系相对于世界坐标系的旋转）。
- **平移向量 t**: 一个  $3 \times 1$  的向量，描述相机的位置（即相机坐标系的原点在世界坐标系中的坐标）。
- 相机位姿可以用一个  $4 \times 4$  的变换矩阵 T 表示：

$$T = [R \ t \ 0]$$

## (3) 物理意义

- 相机位姿定义了相机的视角 (Viewpoint)，即相机从哪个位置、以什么方向观察场景。
- 通过相机位姿，可以将世界坐标系中的点转换到相机坐标系中。

# 五、具体的数据形状变化

## 5.1 输入阶段

- **单张图像**:  $H \times W$  个像素
- **随机采样**: 从  $H \times W$  个像素中随机选择  $N_{rays} = 4096$  个像素， $4096$  是设置每张图像采样的射线数量
- **批次处理**: 将  $batch\_size$  张图像的数据组合在一起

# 批次处理后的数据形状

points: [batch\_size, N\_rays, n\_samples, 3] # [1, 4096, 64, 3]

rays\_d: [batch\_size, N\_rays, 3] # [1, 4096, 3]

rays\_o: [batch\_size, N\_rays, 3] # [1, 4096, 3]

rgb\_gt: [batch\_size, N\_rays, 3] # [1, 4096, 3]

z\_vals: [batch\_size, N\_rays, n\_samples] # [1, 4096, 64]

# 批次处理后的数据形状，如果选择batch\_size为1024

```
points: [1024, 4096, 64, 3] # 1024张图像, 每张4096条射线
```

```
# 张量形状表示法
```

```
[1024, 4096, 64, 3] # 4维张量
```

```
↑      ↑      ↑      ↑
```

```
|      |      |      └ 第4维: 3个分量 (x,y,z 或 R,G,B)
```

```
|      |      └—— 第3维: 64个采样点
```

```
|      └—— 第2维: 4096条射线
```

```
└———— 第1维: 1024张图像
```

```
rays_d: [1024, 4096, 3] # 1024张图像, 每张4096条射线方向
```

```
rays_o: [1024, 4096, 3] # 1024张图像, 每张4096条射线原点
```

```
rgb_gt: [1024, 4096, 3] # 1024张图像, 每张4096个像素的RGB值
```

## 5.2 模型调用

```
# 调用NeRF模型
```

```
raw = network_query_fn(pts, viewdirs, network_fn)
```

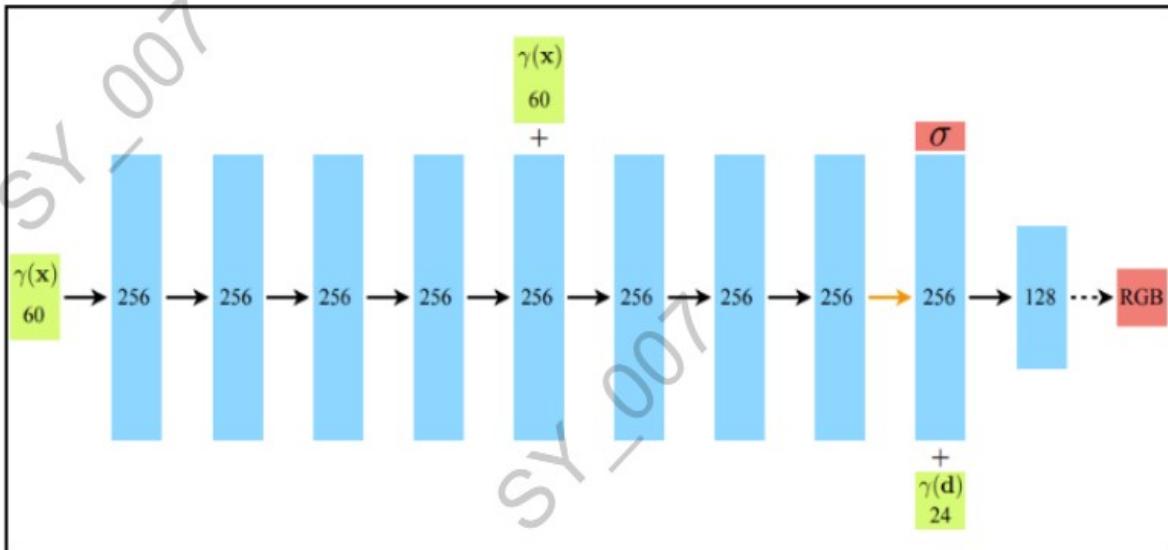
```
# pts: [1024, 64, 3] - 3D采样点, [1024条射线, 64个采样点, 3个坐标]
```

```
# viewdirs: [1024, 3] - 单位化的射线方向, [1024条射线, 3个方向分量]
```

```
# 返回: raw - 模型预测的密度和颜色
```

# 六、模型调用

## 6.1 模型架构



- 8层全连接层
- 半路再次输入位置坐标
- 后半路输出密度
- 后半路输入方向视角
- 最后输出颜色 RGB

## 6.2 位置编码

当只输入(3D位置, 3D视角)时, 建模结果 细节丢失。缺乏高频信息, 引入位置编码进行改善

**位置编码的作用:** 通过将低维输入映射到高维空间, 增强模型对高频信息的捕捉能力, 从而改善建模结果

$$\gamma(p) = (\sin(2^0 \pi p), \cos(2^0 \pi p), \dots, \sin(2^{L-1} \pi p), \cos(2^{L-1} \pi p))$$

- $p$ 需要归一化到[-1, 1]
- 对于空间坐标,  $L = 10$  (频率级别), 每个坐标编码后的维度为  $2L = 20$ , 3个坐标的总维度为  $3 * 20 = 60$ , 加上原始的3D坐标, 总维度为  $60 + 3 = 63$
- 对于3D视角  $(\theta, \phi)$ , 通常用单位向量表示,  $L = 4$ , 每个视角坐标编码后的维度为  $2L = 8$ , 3个视角坐标的总维度为  $3 * 8 = 24$ , 加上原始的3D视角, 总维度为  $24$

+ 3 = 27

• 在代码中,加上初始值: 是63D, 是27D

- $L$  : 编码的层数 (频率级别) 。
- 每个频率级别生成两个值 ( $\sin$  和  $\cos$ ) , 因此编码后的维度为  $2L$  。

## 6.3 3D视角

### 1. 视角向量的几种表示方式

一个 **视角 / viewing direction**, 本质上就是一条 **单位向量**, 表示相机射线的方向。它确实 **自由度是 2** (因为长度被约束为 1) , 但在代码实现和网络输入时, 常用 **三维坐标**表示:

$$\mathbf{d} = (d_x, d_y, d_z), \quad \|\mathbf{d}\| = 1$$

这样就避免了奇点问题 (比如球坐标在两极点会出问题) , 也方便直接做位置编码。

### 2. 为什么用 3D 向量而不是 2D 参数?

- **球坐标 (2 参数)** :

可以写成  $(\theta, \phi)$  表示方位角和仰角。

$$d_x = \cos \theta \sin \phi, \quad d_y = \sin \theta \sin \phi, \quad d_z = \cos \phi$$

→ 虽然只需 2 个参数, 但涉及三角函数变换, 还要小心周期性和奇点。

- **笛卡尔坐标 (3 参数)** :

直接用  $(d_x, d_y, d_z)$  , 虽然冗余 1 个自由度, 但简单、平滑, 没有极点问题。  
训练时只要确保  $\|\mathbf{d}\| = 1$  即可。

所以 NeRF 及大多数神经渲染方法, 都是 **用 3D 向量表示方向**, 再做位置编码。

### 3. 编码时的维度计算

- 输入:  $\mathbf{d} = (d_x, d_y, d_z)$ , 3 维。
- 频率  $L = 4$ 。
- 每个维度生成  $2L = 8$  个 sin/cos 通道:

$$[\sin(2^0\pi d_x), \cos(2^0\pi d_x), \dots, \sin(2^3\pi d_x), \cos(2^3\pi d_x)]$$

同理对  $d_y, d_z$ 。

- 所以:
  - 仅编码部分:  $3 \times 8 = 24$ D
  - 含原始向量:  $24 + 3 = 27$ D

### 4. 直观理解

- 本质上: 视角只有 2 个自由度。
- 实现上: 用 3 个分量存储, 方便网络处理。
- 位置编码只是把这 3 个分量展开成高维特征向量 (比如 27D), 让网络更容易拟合高频细节。

要不要我给你画一张小图 (球面上的方向 → 三维单位向量 → 位置编码展开成 27 维) ? 这样更直观地看到 “自由度=2, 但表示维度=3”的区别。

## 6.4 损失函数

在基于射线 (Ray) 的渲染或重建任务中, 损失函数 (Loss) 的计算是通过比较预测像素颜色和真实像素颜色来实现的。

### 1. 基本思路

- 目标: 对于图片上的某一个像素  $(u, v)$ , 计算其预测颜色与真实颜色之间的差

异。

- **方法：**

- 沿着该像素对应的射线上的多个采样点（粒子）的颜色进行加权求和，得到预测颜色。
- 将预测颜色与真实颜色进行比较，计算损失。

---

## 2. 具体步骤

### (1) 射线上的采样点

- **射线：**对于像素  $(u, v)$ ，通过相机位姿和内参矩阵计算其对应的射线。
- **采样点：**在射线上离散采样多个点（粒子），假设采样了  $N$  个点，每个点的位置为  $t_i$ ，颜色为  $c_i$ ，密度为  $\sigma_i$ 。

### (2) 粒子颜色的加权求和

- **透明度：**每个采样点的透明度  $T_i$  表示光线到达该点之前未被阻挡的概率：

$$[T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \Delta t_j\right)]$$

其中， $\Delta t_j$  是相邻采样点之间的距离。

- **权重：**每个采样点的权重  $w_i$  表示该点对最终颜色的贡献：

$$[w_i = T_i(1 - \exp(-\sigma_i \Delta t_i))]$$

- **预测颜色：**将所有采样点的颜色加权求和，得到预测颜色：

$$[\hat{C}(u, v) = \sum_{i=1}^N w_i c_i]$$

### (3) 损失函数

- **真实颜色**: 图片上像素  $(u, v)$  的真实颜色  $C(u, v)$ 。
- **MSE 损失**: 计算预测颜色与真实颜色之间的均方误差 (MSE) :

$$[\text{Loss}(u, v) = |\hat{C}(u, v) - C(u, v)|^2]$$

- **批量损失**: 对于一个 batch 中的  $R$  条射线 (例如 1024 条), 计算所有射线的损失均值:

$$[\text{Loss} = \frac{1}{R} \sum_{r=1}^R |\hat{C}_r - C_r|^2]$$

## 3. 自监督学习

- **GT (Ground Truth)** : 真实颜色  $C(u, v)$  来自输入图片的像素值。
- **预测值**: 通过射线上的粒子颜色加权求和得到预测颜色
- **自监督**: 不需要额外的标注数据, 直接使用输入图片的像素值作为监督信号。

## 4. 总结

- **射线上的采样点**: 在射线上离散采样多个点, 计算每个点的颜色和密度。
- **粒子颜色的加权求和**: 通过透明度和权重计算预测颜色。
- **损失函数**: 计算预测颜色与真实颜色之间的 MSE 损失。
- **批量损失**: 对一个 batch 中的所有射线计算损失均值。
- **自监督**: 使用输入图片的像素值作为监督信号。

通过这种方法, 可以有效地训练模型, 使其能够从射线上的采样点中重建出高质量的图像。

## 6.5 模型结构2-改进

问题:

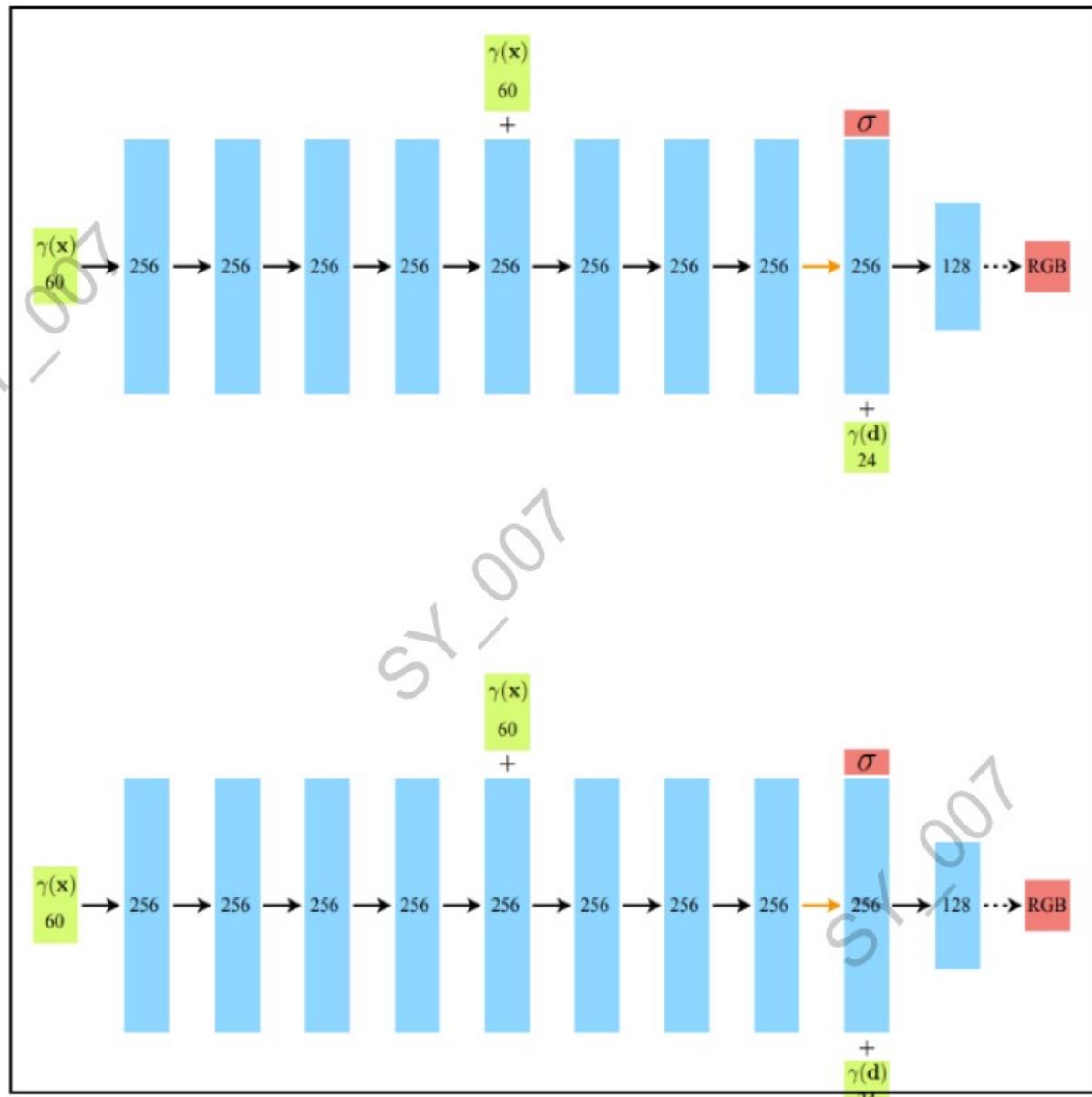
- 无效区域(空白区域和遮挡区域)均匀采样

- 我们希望,有效区域多采样

- 无效区域少采样/不采样

解决方法:

- 可以根据概率密度进行再次采样



- 由2个模型组成

- 粗模型:输入均匀采样粒子,输出密度

- 细模型:根据密度,二次采样

- 最后输出:采用模型2的输出
- 粗模型和细模型结构相同

## 6.6 二次采样-逆变换采样

在基于射线 (Ray) 的渲染或重建任务中, **二次采样** (Secondary Sampling) 是一种通过**逆变换采样** (Inverse Transform Sampling) 技术来优化采样点分布的方法。

### 1. 问题背景

- **粗模型**: 在初始阶段, 每条射线上采样了 64 个粒子, 用于计算颜色和密度。
- **细模型**: 为了捕捉更精细的细节, 需要根据粗模型的结果, 在密度较高的区域进行二次采样。
- **目标**: 通过二次采样, 优化采样点的分布, 使其更符合密度分布。

### 2. 解决方法: 逆变换采样

#### (1) 粗模型的结果

- **粗模型的输出**: 对于每条射线, 粗模型生成了 64 个粒子的密度  $\sigma_i$  和权重  $w_i$ 。
- **权重归一化**: 将权重  $w_i$  进行归一化, 使其和为 1, 即:

$$[\hat{w}_i = \frac{w_i}{\sum j = 1^{64} w_j}]$$

归一化后的权重

$$\hat{w}_i$$

可以看作概率密度函数 (PDF)。

## (2) 生成累积分布函数 (CDF)

- **CDF 的计算**: 根据归一化权重  $\hat{w}_i$ , 生成累积分布函数 (CDF) :

$$[\text{CDF}(i) = \sum_{j=1}^i \hat{w}_j]$$

CDF 是一个单调递增的函数, 范围在  $[0, 1]$  之间。

## (3) 逆变换采样

- **生成随机数**: 使用均匀分布生成一个随机数  $u$ , 例如  $u = \text{drand48}()$ 。
- **反函数计算**: 找到 CDF 的反函数, 即:

$$[r = \text{CDF}^{-1}(u)]$$

得到的  $r$  就是符合 PDF 分布的随机数。

- **采样点**: 根据  $r$  的值, 在射线上确定新的采样点。

## (4) 二次采样

- **新采样点**: 通过逆变换采样, 生成 128 个新的采样点。
- **总采样点**: 将新的 128 个采样点与粗模型的 64 个采样点合并, 每条射线上共有 192 个采样点。

---

### 3. 具体实现步骤

#### (1) 权重归一化

- 计算粗模型的权重  $w_i$  的和, 并归一化:

$$[\hat{w}_i = \frac{w_i}{\sum j = 1^{64} w_j}]$$

## (2) 生成 CDF

- 计算 CDF:

$$[\text{CDF}(i) = \sum_{j=1}^i \hat{w}_j]$$

## (3) 逆变换采样

- 生成均匀随机数 `u = drand48()`。
- 找到 `r` 使得 `CDF(r) = u`，即 `r = CDF{-1}(u)`。
- 根据 `r` 的值，确定新的采样点。
- 这里 `r` 是一个比例因子，用于在区间内确定新的采样点。

$$\bullet \quad *t * new = *t * near + *r * .(*t * far - *t * near)$$

## (4) 合并采样点

- 将新的 128 个采样点与粗模型的 64 个采样点合并，得到 192 个采样点。

## 4. 总结

- 逆变换采样：**通过归一化权重生成 PDF，计算 CDF，再通过反函数生成符合 PDF 分布的随机数。
- 二次采样：**根据密度分布，在射线上生成新的采样点，优化采样点分布。
- 总采样点：**将粗模型的 64 个采样点与细模型的 128 个采样点合并，每条射线上共有 192 个采样点。

通过这种方法，可以有效地优化采样点分布，使模型能够捕捉到更精细的细节。

# 七、体渲染

**体渲染 (Volume Rendering)** 是一种用于渲染 3D 体积数据的技术，其核心思想是通过对光线在体积中的传播过程进行建模，计算每个像素的颜色。在体渲染中，光线上粒子颜色的求和是通过 **连续积分** 实现的。

## 1. 体渲染的基本概念

### (1) 光线传播模型

- 假设一条光线从相机出发，穿过 3D 场景，沿途会遇到多个粒子。
- 每个粒子对光线的贡献包括：
  - 吸收**：粒子吸收光线，导致光线强度减弱。
  - 发射**：粒子自身发光，为光线贡献颜色。

### (2) 连续积分

- 光线的最终颜色是通过对沿途所有粒子的贡献进行积分得到的。
- 积分公式：

$$[\hat{C} = \int_0^{+\infty} T(s)\sigma(s)C(s)ds]$$

其中：

- ( $T(s)$ )：在点 ( $s$ ) 之前，光线没有被阻碍的概率（透明度）。
- ( $\sigma(s)$ )：在点 ( $s$ ) 处，光线碰击粒子的概率密度（密度函数）。
- ( $C(s)$ )：在点 ( $s$ ) 处，粒子发出的颜色（颜色函数）。

## 2. 各部分的详细解释

### (1) 透明度 ( $T(s)$ )

- **定义:** 在点 (  $s$  ) 之前, 光线没有被阻碍的概率。

- **公式:**

$$[T(s) = e^{- \int_0^s \sigma(t) dt}]$$

- **解释:**

- ( $\sigma(t)$ ) 是点 (  $t$  ) 处的密度函数, 表示光线撞击粒子的概率密度。
- 指数函数

$$(e^{- \int_0^s \sigma(t) dt})$$

表示光线在到达点 (  $s$  ) 之前没有被吸收的概率。

### (2) 密度函数 ( $\sigma(s)$ )

- **定义:** 在点 (  $s$  ) 处, 光线撞击粒子的概率密度。

- **解释:**

- ( $\sigma(s)$ ) 描述了粒子在点 (  $s$  ) 处的分布密度。
- 密度越高, 光线撞击粒子的概率越大, 光线被粒子阻碍的概率密度越大

### (3) 颜色函数 ( $C(s)$ )

- **定义:** 在点 (  $s$  ) 处, 粒子发出的颜色。

- **解释:**

- ( $C(s)$ ) 描述了粒子在点 (  $s$  ) 处发出的颜色值 (例如 RGB 值) 。

---

## 3. 总结

概念	描述	公式
透明度 ( $T(s)$ )	在点 ( $s$ ) 之前, 光线没有被阻碍的概率。	( $T(s) = e^{-\int_0^s \sigma_{\text{ma}}(t) dt}$ )
密度函数 ( $\sigma(s)$ )	在点 ( $s$ ) 处, 光线撞击粒子的概率密度。	由场景中的粒子分布决定。
颜色函数 ( $C(s)$ )	在点 ( $s$ ) 处, 粒子发出的颜色。	由场景中的粒子属性决定。
最终颜色 ( $\hat{C}$ )	光线的最终颜色, 通过对沿途所有粒子的贡献进行积分得到。	( $\hat{C} = \int_0^{+\infty} T(s) \sigma(s) C(s) ds$ )

通过连续积分, 体渲染能够准确地模拟光线在 3D 场景中的传播过程, 生成高质量的渲染结果。离散化实现使得计算更加高效, 适用于实际应用。

下面把你图里的“体渲染—离散积分”公式从物理含义 → 连续形式 → 离散化 → 代码可用形式完整解释一遍。

---

## 4. 各符号的物理/概率含义

以一条射线  $\mathbf{r}(s) = \mathbf{o} + s \mathbf{d}$  (从相机原点  $\mathbf{o}$  沿方向  $\mathbf{d}$ ) 为例:

- $\sigma(s)$ : **体密度 / 消光系数** (per meter)。越大表示在此处“被阻挡/散射/吸收”的概率越高。
  - $C(s)$ : **发射的颜色/辐射亮度** (NeRF 网络输出的 RGB)。
  - $T(s)$ : **透过率**, 即“光从射线起点走到  $s$  处之前都没被阻挡的概率”。
- 由 Beer–Lambert 定律:

$$T(s) = \exp\left(-\int_0^s \sigma(u) du\right)$$

直观: 到达位置  $s$  的光要先活着 (透过), 再在  $s$  附近被体积元素吸收并发出我们看到的颜色。

---

## 5. 连续体渲染公式

像素颜色 (沿射线积分) :

$$\mathbf{C}(\mathbf{r}) = \int_0^{\infty} T(s) \sigma(s) C(s) ds$$

其中  $T(s)$  负责“活着走到  $s$ ”， $\sigma(s)ds$  相当于“在这一小段里发生相互作用的概率”， $C(s)$  是这次相互作用 (发射/散射) 产生的颜色。

## 6. 离散化 (把连续积分变成求和)

计算机只能处理离散点。把射线在  $[s_{\text{near}}, s_{\text{far}}]$  上切成  $N$  段：

- 采样点  $s_1, \dots, s_N$ , 相邻间隔  $\delta_i = s_{i+1} - s_i$ 。
- 近似假设：每个小段内  $\sigma(s), C(s)$  **近似常数**，记为  $\sigma_i, C_i$ 。

用 Riemann 和近似：

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^N T_i \underbrace{\left(1 - e^{-\sigma_i \delta_i}\right)}_{\alpha_i} C_i, \quad T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)$$

这正是你图里的公式：

$$\hat{C}(r) = \sum_{i=1}^N T_i (1 - e^{-\sigma_i \delta_i}) C_i, \quad T_i = e^{-\sum_{j=1}^{i-1} \sigma_j \delta_j}.$$

- $\alpha_i = 1 - e^{-\sigma_i \delta_i}$ : 第  $i$  段被“吸收/碰撞”的**条件概率** (把这一小段当作一个“体素”的不透明度)，可以看作第  $n$  个点的不透明度
- $T_i$ : 到达第  $i$  段**之前**从未被遮挡的概率 (前缀的透过率)。

等价形式：

$$Ti = \prod j < i (1 - \alpha_j)$$

。

因为  $1 - \alpha_j = e^{-\sigma_j \delta_j}$ , 连乘就是指数和。

## 7. 这些权重的直觉

- 第  $i$  个采样点对像素颜色的**贡献权重**:

$$w_i = T_i \alpha_i \in [0, 1]$$

- 权重具有**前缀衰减** (越靠近相机的点越可能抢走权重)。
- 权重和不超过 1:  $\sum_i w_i \leq 1$ ; 剩余的  $T_{N+1}$  是**背景透过率**, 可乘上背景颜色  $C_{bg}$ :  
$$\hat{C} \leftarrow \hat{C} + T_{N+1} C_{bg}, \quad T_{N+1} = \exp\left(-\sum_{j=1}^N \sigma_j \delta_j\right).$$

## 8. 与网络输出的对应

- 网络对每个粒子 (采样点) 输出:  $\sigma_i$  (密度) 与  $C_i$  (RGB)。
- $\delta_i$  来自相邻深度样本差 (可均匀采样或分段随机/重要性重采样)。
- 沿射线按上式聚合得到像素预测  $\hat{C}$ , 用 MSE 等损失与真值像素  $C_{gt}$  监督。

## 8. 代码落地 (常见实现要点)

- 先算  $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$ , 可以看作第n个点的不透明度
- 用前缀乘积算  $T_i$ :

```
# 假设 alpha: [N], 按射线顺序
T = torch.cumprod(torch.cat([torch.ones(1), 1 - alpha + 1e-10], 0), 0)[-1]
w = T * alpha          # 权重
C_hat = (w[..., None] * rgb).sum(dim=-2)
```

- 数值稳定性: 给  $1 - \alpha$  或  $\exp$  里加  $\text{eps}$ , 防止 underflow/NaN。
- 期望深度 (可选) :  $\hat{D} = \sum_i w_i s_i$ ; 不透明度:  $\sum_i w_i$ 。

## 9. 和你图上语句一一对应

---

- “如何将光线上粒子颜色进行求和? ”  
→ 就是上面的离散和:  $\sum_i T_i (1 - e^{-\sigma_i \delta_i}) C_i$ 。
- “ $T(s)$ : 在  $s$  点之前光线未被阻碍的概率”  
→ 连续式  $T(s) = \exp(-\int_0^s \sigma)$ , 离散式

$$Ti = \exp(-\sum j < i \sigma j \delta j)$$

- “ $\sigma(s)$ : 在  $s$  处阻碍的概率密度;  $C(s)$ : 在  $s$  处的颜色\*\*”  
→ 对应网络逐点输出  $\sigma_i, C_i$ 。
- “离散化”: 把  $[0, s]$  划分为  $N$  个等间隔/分箱区间, 区间长度为  $\delta_n$ , 并假设区间内  $\sigma_n, C_n$  固定”  
→ 就得到公式里的  $\alpha_i, T_i$  与加权求和。

---

## 一句话记住

---

Beer-Lambert 透过率  $T \times$  段内相互作用概率  $\alpha \times$  该处颜色  $C$ , 沿射线把每段的贡献加权求和, 就是 NeRF 的体渲染离散公式。

$$\begin{aligned}\hat{C} &= \sum_{n=0}^N I(T_n \rightarrow T_{n+1}) \\ &= \sum_{n=0}^N C_n e^{-\sum_{i=0}^{n-1} \sigma_i \delta_i} (1 - e^{\sigma_n \delta_n})\end{aligned}$$

设  $\alpha_n = 1 - e^{-\sigma_n \delta_n}$ ，可以看作第n个点的不透明度。

$$\begin{aligned}\hat{C} &= \sum_{n=0}^N C_n \alpha_n (1 - \alpha_0)(1 - \alpha_1) \dots (1 - \alpha_{n-1}) \\ &= C_0 \alpha_0 + C_1 \alpha_1 (1 - \alpha_0) + C_2 \alpha_2 (1 - \alpha_0)(1 - \alpha_1) \\ &\quad + \dots + C_n \alpha_n (1 - \alpha_0)(1 - \alpha_1) \dots (1 - \alpha_{n-1})\end{aligned}$$

## 八、总结

### 1. 前处理

#### (1) 将像素映射到射线

- **输入**: 图片中的每个像素  $(u, v)$ 。
- **相机模型**: 通过相机位姿和内参矩阵, 将每个像素  $(u, v)$  映射到对应的射线。
- **输出**:  $400 * 400$  条射线 (假设图片分辨率为  $400 \times 400$ )。

#### (2) 在每条射线上采样

- **采样**: 在每条射线上均匀采样 64 个粒子。
- **输出**:  $batch\_size * 64$  个粒子。

#### (3) 位置编码

- **位置编码**: 对每个粒子的位置坐标和方向向量进行编码。

- 位置坐标编码为 63D。
- 方向向量编码为 27D。

- **输出**:

- 位置编码:  $(batch\_size, 64, 63)$ 。
- 方向编码:  $(batch\_size, 64, 27)$ 。

### 2. 模型1

#### (1) 模型结构

- **模型**: 8 层 MLP (多层感知机)。
- **输入**:
  - 位置编码: `(batch_size, 64, 63)`。
  - 方向编码: `(batch_size, 64, 27)`。
- **输出**: `(batch_size, 64, 4)`, 其中 `4` 表示每个粒子的 `(r, g, b, σ)` (颜色和密度)。

## (2) 作用

- 模型1 用于预测初始采样点的颜色和密度。
- 

## 3. 后处理1

### (1) 二次采样

- **目标**: 根据模型1 的输出, 在每条射线上进行二次采样, 增加采样点数量。
- **方法**:
  - 使用逆变换采样 (Inverse Transform Sampling), 在密度较高的区域增加采样点。
  - 每条射线上新增 `128` 个采样点。
- **总采样点**: 每条射线上共有 `64 + 128 = 192` 个采样点。

## (2) 输出

- **输出**:
    - 位置编码: `(batch_size, 192, 63)`。
    - 方向编码: `(batch_size, 192, 27)`。
- 

## 4. 模型2

### (1) 模型结构

- **模型**: 8 层 MLP (多层感知机)。

- **输入：**

- 位置编码：(batch\_size, 192, 63)。
- 方向编码：(batch\_size, 192, 27)。

- **输出：**(batch\_size, 192, 4)，其中 4 表示每个粒子的 (r, g, b, σ) (颜色和密度)。

## (2) 作用

- 模型2 用于预测二次采样后的粒子的颜色和密度。
- 

## 5. 后处理2

### (1) 体渲染

- **透明度：**对于每个采样点，计算其透明度  $T_i$ ：

$$[T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \Delta t_j\right)]$$

- **权重：**对于每个采样点，计算其权重  $w_i$ ：

$$[w_i = T_i(1 - \exp(-\sigma_i \Delta t_i))]$$

- **预测颜色：**对于每条射线，将所有采样点的颜色加权求和，得到预测颜色：

$$[\hat{C} = \sum_{i=1}^{192} w_i c_i]$$

### (2) 生成渲染结果

- **输出：**400x400 的渲染图像。
-

## 6. 总结

步骤	输入	输出	描述
前处理	图片像素 $(u, v)$	射线、采样点、位置编码	将像素映射到射线，采样 64 个粒子，对位置和方向进行编码。
模型1	$(batch\_size, 64, 63)$ 和 $(batch\_size, 64, 27)$	$(batch\_size, 64, 4)$	预测初始采样点的颜色和密度。
后处理1	模型1 的输出	$(batch\_size, 192, 63)$ 和 $(batch\_size, 192, 27)$	进行二次采样，每条射线上共有 192 个采样点。
模型2	$(batch\_size, 192, 63)$ 和 $(batch\_size, 192, 27)$	$(batch\_size, 192, 4)$	预测二次采样后的粒子的颜色和密度。
后处理2	模型2 的输出	400x400 渲染图像	通过体渲染，将采样点的颜色和密度转换为像素值，生成渲染图像。

通过以上步骤，可以高效地完成基于射线的渲染任务，生成高质量的渲染结果。

## 7. 缺点

### (1) 很慢，训练/推理都慢

- 原因：

- 每条射线上需要采样大量点（例如 192 个点）。
- 需要对每个采样点进行复杂的计算（包括 MLP 前向传播、透明度计算等）。
- 体渲染过程本身计算量较大。

- 影响：训练和推理时间较长，难以实时应用。

## (2) 只能表达静态场景

- **原因：**模型假设场景是静态的，无法处理动态物体或场景变化。
- **影响：**无法用于动态场景的渲染（例如视频、动画）。

## (3) 对光照处理的一般

- **原因：**模型仅学习颜色和密度的分布，未显式建模光照条件（例如光源位置、光照强度等）。
- **影响：**渲染结果对光照变化的适应性较差，难以生成复杂光照条件下的图像。

## (4) 没有泛化能力

- **原因：**模型是针对特定场景训练的，无法直接泛化到其他场景。
- **影响：**每渲染一个新场景都需要重新训练模型，增加了时间和计算成本。

## 8. 核心技术与缺点总结

核心内容	描述	优点
体渲染	通过模拟光线在体积中的传播过程，计算每个像素的颜色。	能够渲染复杂的体积数据，生成高质量的图像。
位置编码	将位置坐标和方向向量映射到高维空间，增强模型对高频细节的捕捉能力。	显著提升模型对细节的表现能力。
层级采样	在每条射线上进行两次采样，第一次均匀采样，第二次根据密度分布进行重要性采样。	更高效地捕捉密度分布，提升渲染质量。

缺点	描述	影响
很慢	每条射线上需要采样大量点，计算量大。	训练和推理时间较长，难以实时应用。
只能表达静态场景	模型假设场景是静态的，无法处理动态物体或场景变化。	无法用于动态场景的渲染。

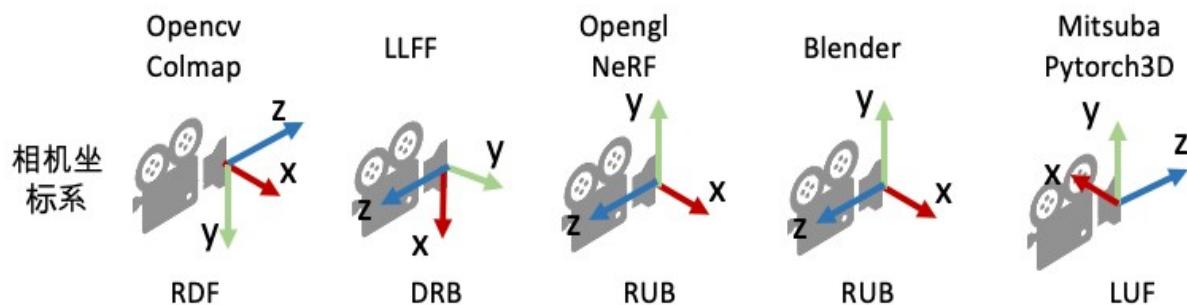
对光照处理的一般	模型未显式建模光照条件，仅学习颜色和密度的分布。	渲染结果对光照变化的适应性较差。
没有泛化能力	模型是针对特定场景训练的，无法直接泛化到其他场景。	每渲染一个新场景都需要重新训练模型。

通过理解核心内容和缺点，可以更好地评估基于射线的渲染任务的适用性和局限性。

## 九、部分补充

### 9.1 坐标系定义

**坐标系定义：**为了唯一地描述每一个空间点的坐标，以及相机的位置和朝向，我们需要先定义一个世界坐标系。一个坐标系其实就是由原点的位置与XYZ轴的方向决定。接着，为了建立3D空间点到相机平面的映射关系以及多个相机之间的相对关系，我们会对每一个相机定义一个局部的相机坐标系。下图为常见的坐标系定义习惯。



### 9.2 6DOF

- 平移运动：刚体可以在3个自由度中平移：向前/后，向上/下，向左/右
- 旋转运动：刚体在3个自由度中旋转：纵摇(Pitch)、横摇(Roll)、垂摇(Yaw)
- 因此，3种类型的平移自由度+3种类型的旋转自由度 = 6自由度

### 9.3 相机外参矩阵与逆矩阵

刚体的任何可能性运动都可以通过6自由度的组合进行表达，对相机在这6个自由度的数值描述即相机外参。

相机外参是一个 $4 \times 4$ 的矩阵，其作用是计算世界坐标系的某个点 $[xw, yw, zw]^T$ 在相机坐标

系下的坐标 $[xp, yp, zp]^T$ 。

我们也把相机外参叫做 **world-to-camera (w2c)矩阵**。

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = [\mathbf{R}|\mathbf{t}] \cdot \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \cdot \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \cdot \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \mathbf{R} \cdot \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + \mathbf{t}$$

其实  $\mathbf{R}$  和  $\mathbf{t}$  分别对应坐标系的 **旋转和平移**。乘上  $\mathbf{R}$  可以看作是坐标系原点不动，各坐标轴旋转到和相机坐标轴对齐时的该点的坐标 $[xw, yw, zw]^T$  应该怎么变，

$\mathbf{t}$  是世界坐标系在相机坐标系下的位置，可以看做相机坐标系的位置相对于世界坐标系中是 $=\mathbf{t}$ 。 $\mathbf{t}$  可以看作是接着将坐标系原点移动到旋转后的坐标系的 $-\mathbf{t}$  位置时的该点坐标应该怎么变， $\mathbf{t}$  值也同时是 **世界坐标系原点在相机坐标系下的位置**。其计算结果 $[xc, yc, zc]^T$  即是目标点相对于相机坐标系的位置：

相机外参的逆矩阵被称为 **camera-to-world (c2w)矩阵**，其作用是把相机坐标系的点变换到世界坐标系

$$[\mathbf{R}|\mathbf{t}]^{-1} \cdot \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}$$

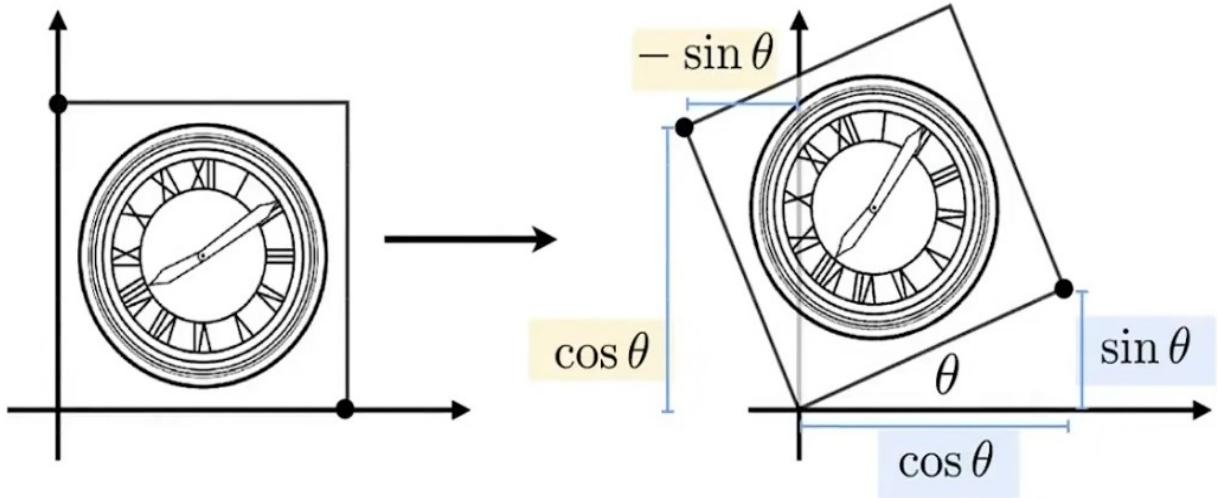
## 9.4 旋转矩阵与欧拉角

---

二维的旋转矩阵可以看作是把单位向量拉到指定的两个夹角 $90^\circ$ 的单位向量上：

给定三维中简单的旋转，可以组合出任意的三维空间中的旋转，对应的 $\alpha, \beta, \gamma$  称为欧拉角

# Rotation Matrix



$$\mathbf{R}_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

拓展到三维：

绕x轴旋转 $\alpha$ ，相当于x轴单位向量 $[1, 0, 0]^T$ 不变，y轴和z轴单位向量 $[0, 1, 0]^T$ 和 $[0, 0, 1]^T$ 分别拉到 $[0, \cos \alpha, \sin \alpha]^T$ 和 $[0, -\sin \alpha, \cos \alpha]^T$ ：

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix}$$

绕y轴旋转 $\alpha$ ，相当于y轴单位向量 $[0, 1, 0]^T$ 不变，x轴和z轴单位向量 $[1, 0, 0]^T$ 和 $[0, 0, 1]^T$ 分别拉到 $[\cos \alpha, 0, -\sin \alpha]^T$ 和 $[\sin \alpha, 0, \cos \alpha]^T$ ：

$$\mathbf{R}_y(\alpha) = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha \\ 0 & 1 & 0 \\ -\sin \alpha & 0 & \cos \alpha \end{bmatrix}$$

绕z轴旋转 $\alpha$ ，相当于z轴单位向量 $[0, 0, 1]^T$ 不变，x轴和y轴单位向量 $[1, 0, 0]^T$ 和 $[0, 1, 0]^T$ 分别拉到 $[\cos \alpha, \sin \alpha, 0]^T$ 和 $[-\sin \alpha, \cos \alpha, 0]^T$ ：

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

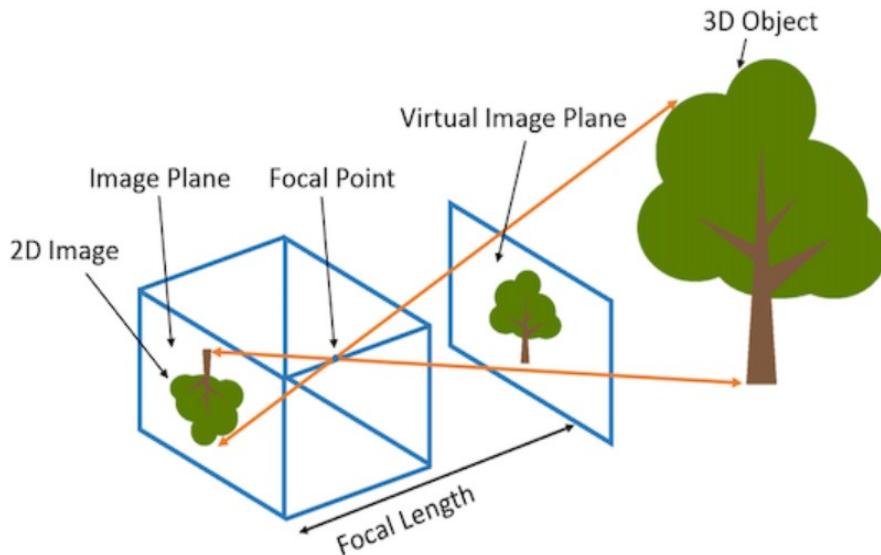
进一步，空间中绕轴 $n$ 旋转 $\alpha$ 的旋转也都可以表示为一个矩阵 $R(n,\alpha)$ ，称为Rodrigues旋转公式：

$$R(n, \alpha) = \cos(\alpha)I + (1 - \cos \alpha)n n^T + \sin \alpha \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix}$$

## 9.5 相机内参

相机的内参矩阵将相机坐标系下的3D坐标映射到2D的图像平面，这里以针孔相机(Pinhole camera)为例介绍相机的内参矩阵K：

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$



内参矩阵K包含4个值，其中 $f_x$ 和 $f_y$ 是相机的水平和垂直焦距（对于理想的针孔相机， $f_x = f_y$ ）。焦距的物理含义是相机中心到成像平面的距离，长度以像素为单位。 $c_x$ 和 $c_y$ 是图像原点相对于相机光心的水平和垂直偏移量（图像传感器中心并不一定与镜头中轴线对齐）。 $c_x$ ,  $c_y$ 有时候可以用图像宽和高的1/2近似：

这里， $zp$ 变成了表示“成像距离”，而 $zcu=xp$ 和 $zcv=yp$ 变成了表示“若在成像距离 $zp$ 处放置成像平面，坐标 $[xc, yc, zc]^T$ 会被投影到该成像平面上的何处”。此即相机内参的真正含义，它不是将一个坐标系变换到另一个坐标系，而是根据目标点给出成像距离与目标点在成像平面上的位置关系。其中的 $zc$ 和 $zp$ 仅仅是最后用于计算归一化成像平面的量，所以其未进行任何变换，直接是 $zp=zc$ 。这个 $zp=zc$ 也可以理解成“小孔成像的物距等于像

距”。

$$z_p \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = z_c \begin{bmatrix} f_x \frac{x_c}{z_c} + c_x \\ f_y \frac{y_c}{z_c} + c_y \\ 1 \end{bmatrix}$$

$$\begin{aligned} \frac{x_p}{z_p} &= f_x \frac{x_c}{z_c} + c_x \\ \frac{y_p}{z_p} &= f_y \frac{y_c}{z_c} + c_y \end{aligned}$$

所以相机内参的真正表现形式应该是

## 9.6 如何获得相机参数

---

### 合成数据

对于合成数据集，我们需要通过指定相机参数来渲染图像，所以得到图像的时候已经知道对应的相机参数，比如像NeRF用到的Blender Lego数据集。常用的渲染软件还有Mitsuba、OpenGL、PyTorch3D、Pyrender等。渲染数据比较简单，但是把得到的相机数据转到NeRF代码坐标系牵扯到坐标系之间的变换，有时候会比较麻烦。

### 直接线性变换(DLT)

DLT的目标是求解出矩阵  $P = \{p_{ij}\}_{3 \times 4P}$

由于  $[p_{11}, p_{21}, p_{31}]$  一共有 12 个值，因此最少通过六对匹配点（12条方程），即可实现参数矩阵  $[R|t]$  的线性求解，这种方法(也)称为直接线性变换(Direct Linear Transform, DLT)。当匹配点大于六对时，可以使用 SVD 等方法对超定方程求最小二乘解。

$$P = K[R|t] = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$$

## P3P算法

PnP(Perspective-n-Point)是求解 3D 到 2D 点对运动的方法。它描述了当我们知道  $n$  个 3D 空间点以及它们的投影位置时,如何估计相机所在的位姿

通俗的讲, PnP问题就是在已知世界坐标系下 $n$ 个空间点的真实坐标以及这些空间点在图像上的投影, 如何计算相机所在的位姿。换句话说, 就是已知量为空间中的真实坐标和图像坐标, 求解相机的位姿 (未知量) 。

P3P 算法只利用三个点的信息进行求解, 其缺点是不能利用更多点的信息, 优点是不需要SVD所以计算速度快。

P3P 顾名思义  $n=3$ , 即利用给定的3个点的几何关系。它的输入数据为三对 3D-2D 匹配点。记 3D 点为  $A, B, C$ , 2D 点为  $a, b, c$ , 其中小写字母代表的点为大写字母在相机成像平面上的投影, 如上图所示。此外, P3P 还需要使用一对验证点, 以从可能的解中选出正确的那一个(类似于对极几何情形)。记验证点对为  $D-d$ , 相机光心为  $O$ 。请注意, 我们知道的是  $A, B, C$  在 **世界坐标系中的坐标**, 而不是在 **相机坐标系中的坐标**。

缺点:

- P3P 只利用三个点的信息。当给定的配对点多于 3 组时, 难以利用更多的信息。
- 如果 3D 点或 2D 点受噪声影响, 或者存在误匹配, 则算法失效。所以后续人们还提出了许多别的方法, 如 EPnP、UPnP 等。它们利用更多的信息, 而且用迭代的方式对相机位姿进行优化, 以尽可能地消除噪声的影响。

## 9.7 3D空间射线怎么构造

给定一张图像的一个像素点, 我们的目标是构造以相机中心为起始点, 经过相机中心和像素点的射线。

首先, 明确两件事:

1. 一条射线包括一个起始点和一个方向, 起点的话就是相机中心。对于射线方向, 我们都应该知道两点确定一条直线, 所以除了相机中心我们还需另一个点, 而这个点就是成像平面的像素点。
2. NeRF 代码是在相机坐标系下构建射线, 然后再通过 camera-to-world (c2w) 矩阵将射线变换到世界坐标系。

首先3D像素点的x和y坐标是2D的图像坐标  $(i, j)$  减去光心坐标  $(c_x, c_y)$ ，然后z坐标其实就是焦距  $f$ （因为图像平面距离相机中心的距离就是焦距  $f$ ）。

所以我们可以得到射线的方向向量是  $[i - c_x, j - c_y, f] - [0, 0, 0] = [i - c_x, j - c_y, f]$ 。因为是向量，我们可以把整个向量除以焦距  $f$  归一化z坐标，得到  $(\frac{i-c_x}{f}, \frac{j-c_y}{f}, 1)$ 。

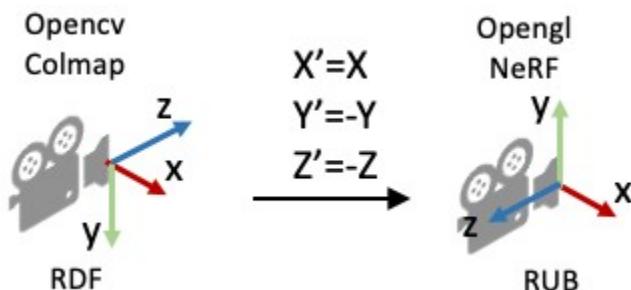
接着只需要用c2w矩阵把相机坐标系下的相机中心和射线方向变换到世界坐标系就搞定了。

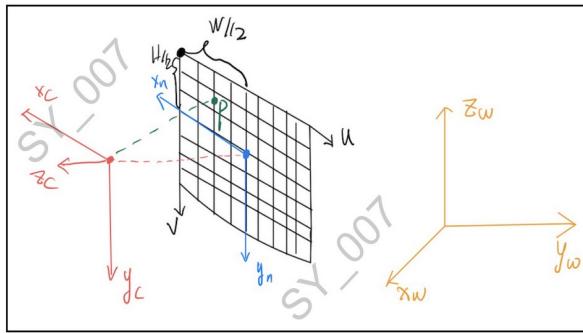
```
def get_rays_np(H, W, K, c2w):
    i, j = np.meshgrid(np.arange(W, dtype=np.float32), np.arange(H, dtype=np.float32))
    dirs = np.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1][1], -np.ones_like(i)],
    # Rotate ray directions from camera frame to the world frame
    rays_d = np.sum(dirs[...], np.newaxis, :) * c2w[:3,:3], -1) # dot product, equal
    # Translate camera frame's origin to the world frame. It is the origin of all r
    rays_o = np.broadcast_to(c2w[:3,-1], np.shape(rays_d))
    return rays_o, rays_d
```

为什么函数的第二行中dirs的y和z的方向值需要乘以负号，和我们刚刚推导的的  $(fi-cx, fj-cy, 1)$  不太一样呢？

这是因为OpenCV/Colmap的相机坐标系里相机的Up/Y朝下，相机光心朝向+Z轴，

而NeRF/OpenGL相机坐标系里相机的Up/Y朝上，相机光心朝向-Z轴，所以这里代码在方向向量dir的第二和第三项乘了个负号。





从图片获得射线

由像素点 $P(u, v)$ 反推射线：

- 像素平面  $\rightarrow$  物理成像平面  $(x_n, y_n) = (-\frac{u}{w}, \frac{v}{h})$
- 物理成像平面  $\rightarrow$  相机坐标系  $(x_c, y_c, z_c) = (x_n, y_n, -f)$
- 归一化  $(x_c, y_c, z_c) = (\frac{x_c}{f}, \frac{y_c}{f}, -1)$
- 相机坐标系  $\rightarrow$  世界坐标系  $(x_w, y_w, z_w) = c2w * (x_c, y_c, z_c)$

## 十、代码详解

### 10.1 dataloaders.py文件

#### 10.1.1 class LoadSyntheticDataset(Dataset):

- 作为NeRF合成数据集的加载类，继承自Dataset类
- `def __init__(self, path_to_images, path_to_labels):`: 输入是图像文件路径，标签文件路径
- `def get_origins_and_directions(self, frame, width, height):`

1. 根据相机的外参矩阵，计算射线的原点与方向
2. 输入为包含相机变换矩阵的帧数据，图像宽，图像高
3. 输出为射线的原点 `origins [width*height, 3]`，射线方向 `directions [width*height, 3]`

"""\norigins示例:

形状: `[width*height, 3] = [10000, 3]`

```
origins = torch.tensor([
    [1.0, 2.0, 3.0], # 像素(0,0)的射线起点
    [1.0, 2.0, 3.0], # 像素(0,1)的射线起点
    [1.0, 2.0, 3.0], # 像素(0,2)的射线起点
    ...
    [1.0, 2.0, 3.0], # 像素(99,99)的射线起点
])
```

所有射线起点都相同，都是相机位置 """"

"""" directions示例：

形状：[width\*height, 3] = [10000, 3]

```
directions = torch.tensor([
    [-0.5, -0.5, -1.0], # 像素(0,0)的射线方向
    [-0.49, -0.5, -1.0], # 像素(0,1)的射线方向
    [-0.48, -0.5, -1.0], # 像素(0,2)的射线方向
    ...
    [0.5, 0.5, -1.0], # 像素(99,99)的射线方向
])
```

每个像素的射线方向都不同 """"

- `def sample_random_rays(self, rays_o, rays_d, N_rays):` 从所有射线中随机采样一定数量的射线

Args: rays\_o: 射线原点 [total\_rays, 3], rays\_d: 射线方向 [total\_rays, 3], N\_rays: 要采样的射线数量

Returns: rays\_o\_sampled: 采样后的射线原点 [N\_rays, 3], rays\_d\_sampled: 采样后的射线方向 [N\_rays, 3]

- `def get_rays_sampling(self, origins, directions, near, far, samples):`沿着射线进行采样，生成3D点

## 1. Args:

`origins`: 射线原点 [N\_rays, 3]

`directions`: 射线方向 [N\_rays, 3]

`near`: 近平面距离

`far`: 远平面距离

`samples`: 采样点数量

## Returns:

`points`: 采样点坐标 [N\_rays, samples, 3]

`z_vals`: 深度值 [samples]

## 2. 计算沿射线的采样点：原点 + 方向 \* 深度值

`points = origins + directions * z_vals # [N_rays, samples, 3]`

- `def __getitem__(self, idx):`获取指定索引的数据项，这个函数完成了前几个函数所有的功能，前几个没用上

1. `idx`: 整数索引，表示要获取第几张图像，然后获取对应帧的标签信息，检查图像并转换为RGB格式
2. 设置每张图像采样的射线数量，`N_rays = 4096`，得到H, W，焦距后计算射线方向，并将射线方向转换到世界坐标系，并得到射线原点
3. 设置`near, far = 2.0, 6.0`，应用随机扰动进行分层采样采样64个点，计算沿射线的3D采样点， [N\_rays, 64, 3]
4. 返回包含所有必要信息的字典

```

return {
    'points': points,      # 3D采样点坐标
    'rays_d': rays_d,     # 射线方向
    'rgb_gt': rgb_gt,     # 真实RGB值
}
```

```
'z_vals': z_vals      # 深度值
```

```
}
```

### 5. 输出1: 3D采样点坐标 [N\_rays, 64, 3]

```
'points': torch.tensor([
```

```
# 形状: [4096, 64, 3]
```

```
# 4096条射线, 每条射线64个采样点, 每个点3D坐标
```

```
[
```

```
[[1.0, 2.0, 3.0], [1.1, 2.1, 3.1], ...], # 第1条射线的64个点
```

```
[[1.0, 2.0, 3.0], [1.2, 2.2, 3.2], ...], # 第2条射线的64个点
```

```
...
```

```
]
```

```
]),
```

### 6. 输出2: 射线方向rays\_d [N\_rays, 3]

```
'rays_d': torch.tensor([
```

```
# 形状: [4096, 3]
```

```
# 4096条射线的方向向量
```

```
[0.1, 0.2, -1.0], # 第1条射线方向
```

```
[0.15, 0.25, -1.0], # 第2条射线方向
```

```
...
```

```
])
```

### 7. 输出3 rgb\_gt: 真实RGB值[N\_rays, 3]

```
'rgb_gt': torch.tensor([
```

```
# 形状: [4096, 3]
```

```
# 4096个像素的真实RGB值  
[0.8, 0.6, 0.4],      # 第1个像素的RGB  
[0.9, 0.7, 0.5],      # 第2个像素的RGB  
...  
]),
```

#### 8. 输出4: 深度值 z\_vals [N\_rays, 64]

```
'z_vals': torch.tensor([  
    # 形状: [4096, 64]  
    # 每条射线的64个深度值  
    [2.0, 2.06, 2.12, ..., 6.0],  # 第1条射线的深度  
    [2.0, 2.06, 2.12, ..., 6.0],  # 第2条射线的深度  
    ...  
])
```

#### 9. 如果调用其他函数的形式:

```
def __getitem__(self, idx):  
    # 使用其他方法组合功能  
    origins, directions = self.get_origins_and_directions(...)  
    rays_o, rays_d = self.sample_random_rays(origins, directions, N_rays)  
    points, z_vals = self.get_rays_sampling(rays_o, rays_d, near, far,  
    samples)  
  
    return {'points': points, 'rays_d': rays_d, 'rgb_gt': rgb_gt,  
    'z_vals': z_vals}
```

### 10.1.2 class CustomDataloader

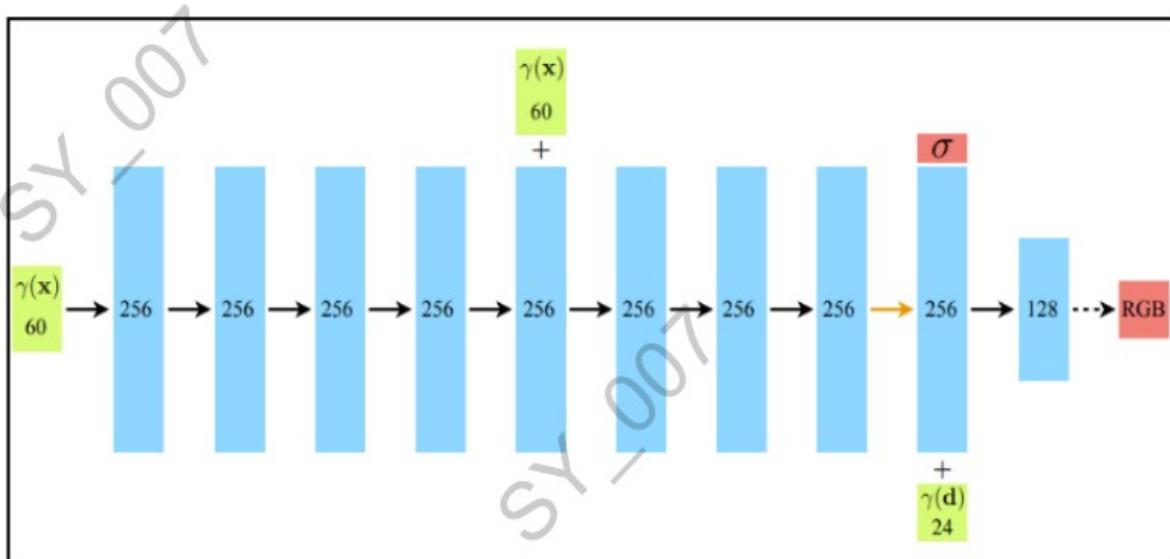
- 自定义的数据集加载器类

- `def __init__(self, batch_size, path_to_images=None, path_to_labels=None):`: 初始化自定义数据集加载器类，包含并使用之前的 `LoadSyntheticDataset`，参数包括批次大小，图片数据路径，标签路径

## 10.2 Model.py文件

### 10.2.1 class NeRF(nn.Module):

- 核心模型，用于学习3D场景的密度和颜色表示
- `def __init__(self, pos_in_dims=63, dir_in_dims=27, hidden_size=256):`: 初始化 NeRF网络
- `def forward(self, x, d):`: 前向传播函数，`x`: 位置编码后的3D坐标 `[N, pos_in_dims]`, `d`: 方向编码后的视角方向 `[N, dir_in_dims]`



1. `pos_in_dims`: 位置编码后的输入维度，默认63  $(3*(1+2*10))$ ，`dir_in_dims`: 方向编码后的输入维度，默认27  $(3*(1+2*4))$ ，`hidden_size`: 隐藏层大小，默认256

2. 第一层: `h = F.relu(self.layer1(x))`

```
# 输入: x [4096, 63]
```

```
# 权重: [63, 256]
```

```
# 输出: h [4096, 256]
```

第二层: `h = F.relu(self.layer2(h))`

```
# 输入: h [4096, 256]
```

```
# 权重: [256, 256]
```

```
# 输出: h [4096, 256]
```

第三层: 跳跃连接 `h = F.relu(self.layer3(torch.cat([h, x], dim=-1)))`

```
# 输入: torch.cat([h, x], dim=-1) [4096, 256+63] = [4096, 319]
```

```
# 权重: [319, 256]
```

```
# 输出: h [4096, 256]
```

第四层: `h = F.relu(self.layer4(h))`

```
# 输入: h [4096, 256]
```

```
# 权重: [256, 256]
```

```
# 输出: h [4096, 256]
```

第五层: 跳跃连接 `h = F.relu(self.layer5(torch.cat([h, x], dim=-1)))`

```
# 输入: torch.cat([h, x], dim=-1) [4096, 256+63] = [4096, 319]
```

```
# 权重: [319, 256]
```

```
# 输出: h [4096, 256]
```

第六层: `# 输入: h [4096, 256]`

```
# 权重: [256, 256]
```

```
# 输出: h [4096, 256]
```

第七层: 跳跃连接 `h = F.relu(self.layer7(torch.cat([h, x], dim=-1)))`

```
# 输入: torch.cat([h, x], dim=-1) [4096, 256+63] = [4096, 319]
```

```
# 权重: [319, 256]
```

```
# 输出: h [4096, 256]
```

第八层: `# 输入: h [4096, 256]`

```
# 权重: [256, 256]
```

```
# 输出: h [4096, 256]
```

### 3. 密度计算阶段

```
sigma = self.sigma_layer(h)
```

```
# 输入: h [4096, 256]
```

```
# 权重: [256, 1]
```

```
# 输出: sigma [4096, 1]
```

### 4. 方向处理阶段

准备方向输入 `dir_input = torch.cat([h, d], dim=-1)`

```
# h: [4096, 256] (位置特征)
```

```
# d: [4096, 27] (方向编码)
```

```
# 输出: dir_input [4096, 256+27] = [4096, 283]
```

方向处理第一层 `h = F.relu(self.dir_layer1(dir_input))`

```
# 输入: dir_input [4096, 283]
```

```
# 权重: [283, 128] (hidden_size//2 = 256//2 = 128)
```

```
# 输出: h [4096, 128]
```

方向处理第二层: `h = F.relu(self.dir_layer2(h))`

```
# 输入: h [4096, 128]
```

```
# 权重: [128, 128]
```

```
# 输出: h [4096, 128]
```

### 5. RGB颜色输出

```
rgb = torch.sigmoid(self.rgb_layer(h))
```

```
# 输入: h [4096, 128]
```

```
# 权重: [128, 3]  
  
# 输出: rgb [4096, 3] (R, G, B值在[0,1]范围内)
```

sigmoid激活函数：确保RGB值在[0,1]范围内，符合颜色表示要求。

## 6. 最终输出

```
return torch.cat([rgb, sigma], dim=-1)  
  
# rgb: [4096, 3]  
  
# sigma: [4096, 1]  
  
# 输出: [4096, 4] (R, G, B, sigma)
```

### 10.2.2 class PositionalEncoding(nn.Module)

- 位置编码模块
- `def __init__(self, num_freqs=10, include_input=True):`: 初始化位置编码器，`num_freqs`: 频率数量，默认10，`include_input`: 是否包含原始输入，默认True
- `def forward(self, x):`: 前向传播，对输入位置进行编码，

`x`: 输入坐标 [N, 3]

Returns:

`encoded`: 编码后的特征 [N, 3\*(1+2\*num\_freqs)]

### 10.2.3 class NeRFModel(nn.Module)

- 完整的模型架构，结合位置编码和方向编码的端到端神经网络
- `def __init__(self, pos_freqs=10, dir_freqs=4, hidden_size=256):`  
`pos_freqs`: 位置编码的频率数量，默认10  
`dir_freqs`: 方向编码的频率数量，默认4  
`hidden_size`: 隐藏层大小，默认256
- `def forward(self, points, view_dirs):`

points: 3D坐标点 [N, 3]

view\_dirs: 视角方向 [N, 3]

Returns:

output: RGB和密度值 [N, 4]

## 10.3 Train.py文件

---

### 10.3.1 class TrainModel

- NeRF模型训练类，负责管理训练过程、数据加载、优化器和损失函数

- def \_\_init\_\_(self): 初始化训练模型

i. 检测并设置计算设备（优先使用GPU）

ii. 初始化模型 self.model = NeRFModel(pos\_freqs=10, dir\_freqs=4, hidden\_size=256).to(self.device)

iii. 设置Adam优化器 lr=1e-4, # 较低的学习率，有助于稳定训练

betas=(0.9, 0.999), # Adam优化器的动量参数

eps=1e-8 # 数值稳定性参数

iv. 设置批次大小为1

v. 创建数据加载器

vi. 设置epoch为200

vii. 使用均方误差损失函数 self.mse\_loss = nn.MSELoss()

viii. 设置学习率调度器（指数衰减） self.scheduler = optim.lr\_scheduler.ExponentialLR(self.optimizer, gamma=0.995)

ix. # 训练状态变量

```
    self.start_epoch = 0 # 起始轮数
```

```
    self.best_loss = float('inf') # 最佳损失值，记录训练过程中遇到的最小损失值
```

```
    self.running_loss = [] # 运行中的损失列表，存储最近几个批次的损失
```

值

```
self.window_size = 100 # 损失窗口大小, running_loss 列表的最大长度
```

- `def train(self):` 包含完整的训练循环、体积渲染和模型保存

每一个epoch都要遍历数据加载器中的每个批次，过程如下：

- i. 设置每次处理2048条射线, `chunk_size = 2048 # 分块大小 (减少内存使用)`
- ii. 清零梯度`self.optimizer.zero_grad()`

- iii. 分块进行体积渲染：

- `points: [1, 4096, 64, 3] → [2048, 64, 3]` (处理2048条射线), `rays_d: [1, 4096, 3] → [2048, 3]`, `z_vals: [1, 4096, 64] → [2048, 64]`
- 展平数据并扩展方向: `points_flat: [2048, 64, 3] → [131072, 3]` ( $2048 \times 64 = 131072$ 个点), `dirs_flat: [2048, 3] → [131072, 3]` (每条射线的方向复制64次), 表示每条射线的64个采样点都使用相同的方向向量
- 通过模型获取RGB和密度 `outputs = self.model(points_flat, dirs_flat)`

```
outputs = outputs.reshape(-1, n_samples, 4) # [N, S, 4]
```

输入：

`points_flat: [131072, 3]` - 3D坐标

`dirs_flat: [131072, 3]` - 方向向量

输出：

`outputs: [131072, 4] → [2048, 64, 4]` (RGB + 密度)

- 分离RGB和密度: `rgb: [2048, 64, 3]` - 每个采样点的RGB颜色, `sigma: [2048, 64]` - 每个采样点的密度值
- 计算相邻采样点之间的距离

```
dists = z_vals_chunk[..., 1:] - z_vals_chunk[..., :-1] # [N, S-1]
```

为最后一个采样点添加大距离值

```
dists = torch.cat([dists, torch.ones_like(dists[..., :1]) * 1e10], dim=-1)
```

```
z_vals = [2.0, 2.06, 2.12, 2.18, ..., 6.0] # 深度值  
dists = [0.06, 0.06, 0.06, ..., 1e10] # 相邻点距离
```

- 计算alpha值（不透明度） 不透明度表示光线在经过该体积后的累积吸收效果，而体密度反映了光线在体积内的吸收和散射程度

```
alpha = 1 - torch.exp(-F.relu(sigma) * dists) # [N, S]
```

公式:  $\alpha = 1 - \exp(-\sigma \times \delta)$

- $\sigma$ : 密度值
- $\delta$ : 采样间隔
- $\alpha$ : 不透明度 (0-1)

- 计算透射率T（累积乘积）

```
T = torch.cumprod(torch.cat([torch.ones_like(alpha[..., :1]), # 第一个透射率为1 (1 - alpha + 1e-10)[..., :-1] # 后续透射率 ], dim=-1), dim=-1)
```

光线到达该点前未被吸收的概率，计算:  $T_i = \prod(1 - \alpha_j) \text{ for } j < i$

- 计算权重（体积渲染的核心）

```
weights = alpha * T # [N, S]
```

每个采样点对最终颜色的贡献权重，公式:  $w_i = \alpha_i \times T_i$

- 计算最终RGB值（加权平均）

```
rgb_chunk = (weights.unsqueeze(-1) * rgb).sum(dim=1) # [N, 3]
```

沿射线的加权颜色积分

公式:  $C = \sum(w_i \times c_i)$

iv. 计算损失（预测RGB与真实RGB的均方误差）

v. 反向传播 loss.backward()

vi. 梯度裁剪, `torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=0.1)`

vii. 更新模型参数`self.optimizer.step()`, 更新统计信息和损失列表

现在遍历批次结束，开始统计这个epoch的信息

- i. 更新学习率self.scheduler.step()
- ii. 计算当前轮的平均损失 $\text{avg\_loss} = \text{epoch\_loss} / \text{batch\_count}$
- iii. 保存最佳模型
- iv. 每50轮一个检查点

## 10.4 Inference.py文件

---

### 10.4.1 `def load_checkpoint(model, checkpoint_path, device='cpu')`

- 加载最新的模型检查点，model: 要加载权重的NeRF模型，checkpoint\_path: 检查点文件路径，device: 目标设备，默认为CPU

### 10.4.2 `def get_rays(H, W, focal, c2w, device='cuda')`

- 根据给定的相机参数生成射线
- 输入 H: 图像高度 (如400)  
W: 图像宽度 (如400)  
focal: 相机焦距 (如277.13)  
c2w: 相机到世界的变换矩阵 [4, 4]

- device: 计算设备
- 返回 rays\_o: 射线原点 [H, W, 3] - 每条射线在3D空间中的起始点  
rays\_d: 射线方向 [H, W, 3] - 每条射线的方向向量
- 和数据加载时候的函数相似，但有不同：推理用的是来处理任意相机姿态，生成完整图像的射线，进行坐标系转换：相机坐标系 → 世界坐标系

### 10.4.3 `def render_novel_view(model, c2w, H=400, W=400, focal=None, device='cuda')`

- 从给定的相机视角渲染新视角
- 输入 model: 训练好的NeRF模型

c2w: 相机到世界的变换矩阵

H: 图像高度, 默认400

W: 图像宽度, 默认400

focal: 相机焦距, 如果为None则使用W/2

device: 计算设备

- 输出: rgb\_final: 渲染的RGB图像 [H, W, 3]
- 将模型设置为评估模式 model.eval()
- 生成射线并把维度展平, 分块处理, 每次处理4096条射线,

```
rays_o: [400, 400, 3] → [160000, 3] # 展平后
```

```
rays_d: [400, 400, 3] → [160000, 3] # 展平后
```

- 深度采样, 使用与训练相同的采样策略, 分层采样, 有上边界和下边界, 并添加随机扰动, 得到 z\_vals = lower + (upper - lower) \* t\_rand
- 计算沿射线的采样点, 并扩展所有视角的采样点, 展平点和方向以便模型处理

### 射线参数

```
rays_o_chunk: [4096, 3] # 射线起点, rays_d_chunk: [4096, 3], # 射线方向  
, z_vals: [4096, 64] # 深度值
```

### 计算3D点

```
points: [4096, 64, 3] # 每条射线64个3D点 view_dirs: [4096, 64, 3] # 每条射线64个  
方向 (相同)
```

### 展平

```
points_flat: [262144, 3] # 4096*64=262144个3D点 dirs_flat: [262144, 3] # 262144  
个方向
```

- 在更小的子块8192中处理以进一步节省内存

子块处理

```
points_flat: [262144, 3] # 总点数 sub_chunk_size = 8192 # 每子块8192个点
```

第1子块

```
points_sub: [8192, 3] # 8192个3D点 , dirs_sub: [8192, 3] # 8192个方向,  
outputs_sub: [8192, 4] # 8192个输出(RGB+密度)
```

第2子块

```
points_sub: [8192, 3] outputs_sub: [8192, 4]
```

最后子块

```
points_sub: [8192, 3] # 剩余点数 outputs_sub: [8192, 4]
```

合并输出

```
outputs: [262144, 4] # 所有输出 outputs: [4096, 64, 4] # 重塑为射线形状
```

- 分离输出，并缩放密度0.1

```
outputs: [4096, 64, 4] # 模型输出
```

```
rgb: [4096, 64, 3] # RGB颜色
```

```
sigma: [4096, 64] # 密度值
```

```
sigma: [4096, 64] # 缩放后的密度
```

- 体积渲染计算

- i. 计算相邻采样点之间的距离，为最后一个采样点添加一个很大的距离值

- ii. 计算alpha合成（不透明度）

- iii. 计算T和权重（体积渲染的核心），计算最终的RGB值（加权平均）

- iv. dists: [4096, 63] # 相邻点距离

```
dists: [4096, 64] # 添加最后一个距离
```

```
# Alpha计算
```

```
sigma: [4096, 64] # 密度值
```

```
alpha: [4096, 64]          # 不透明度  
  
# 权重计算  
  
T: [4096, 1]                # 透射率  
  
weights: [4096, 64]          # 最终权重  
  
# RGB合成  
  
rgb: [4096, 64, 3]          # RGB颜色  
  
weights: [4096, 64]          # 权重  
  
rgb_chunk: [4096, 3]         # 最终RGB  
  
# 存储结果  
  
rgb_final: [160000, 3]       # 最终图像
```

#### v. 逐层计算权重的意思：

当前层权重 = 更新后的透射率 × 当前层不透明度

更新透射率 = 透射率 × (1 - 当前层不透明度)

输入数据

```
alpha = [0.1, 0.3, 0.5, 0.2] # 各层不透明度 T = [1.0] # 初始透射率
```

逐层计算

层0

```
weights[0] = 1.0 * 0.1 = 0.1 # 第0层权重 T = 1.0 * (1 - 0.1) = 0.9 # 更新透射率
```

层1

```
weights[1] = 0.9 * 0.3 = 0.27 # 第1层权重 T = 0.9 * (1 - 0.3) = 0.63 # 更新透射率
```

层2

```
weights[2] = 0.63 * 0.5 = 0.315 # 第2层权重 T = 0.63 * (1 - 0.5) =
```

```
0.315 # 更新透射率
```

层3

```
weights[3] = 0.315 * 0.2 = 0.063 # 第3层权重 T = 0.315 * (1 - 0.2) =  
0.252 # 更新透射率
```

最终权重

```
weights = [0.1, 0.27, 0.315, 0.063] # 各层贡献权重
```

- 重塑处理

```
rgb_final: [160000, 3] # 展平的图像
```

```
rgb_final: [400, 400, 3] # 重塑为图像形状
```

```
rgb_final: [400, 400, 3] # 限制在[0,1]范围内
```

#### 10.4.4 def create\_360\_degree\_poses(num\_frames=120, radius=4.0, h=0.5)

- 创建围绕物体的360度旋转相机姿态
- 输入： num\_frames: 总帧数， 默认120， radius: 相机轨道半径， 默认4.0， h: 相机高度偏移， 默认0.5
- 输出： poses: 包含变换矩阵的相机姿态列表
- 在0到360度之间均匀分布角度， 创建相机到世界的变换矩阵， 输出包含120个相机外参矩阵的列表
- 生成相机轨迹， 调用下面的look\_at构建每个姿态

#### 10.4.5 def look\_at(eye, target, up)

- 作用：根据相机位置、注视点和上方向构建单个相机的外参矩阵
- 输入： eye: [3] # 相机位置  
target: [3] # 注视点  
up: [3] # 上方向向量
- 输出： c2w: 相机到世界的变换矩阵 [3, 4]

#### 10.4.6 `def create_gif(image_folder, output_path, duration=0.1)`

- 从图像文件夹创建GIF动画

---

- 参数：

- image\_folder: 包含图像的文件夹路径

- output\_path: 输出GIF文件路径

- duration: 每帧持续时间 (秒) , 默认0.1

- 遍历所有png文件

#### 10.4.7 `def load_test_poses(transforms_path)`

- 从transforms.json文件加载测试姿态
- 输入： transforms\_path: transforms.json文件路径
- 输出： frames: 包含变换矩阵和文件路径的帧列表, camera\_angle\_x: 相机水平视角

#### 10.4.8 `def main():主函数`

- 初始化NeRF模型

- model = NeRFModel(pos\_freqs=10, dir\_freqs=4).to(device)

- 加载训练好的模型检查点，首先尝试加载最佳检查点
- 创建输出目录, render\_dir = 'rendered\_views'
- 加载测试变换数据, transforms\_test.json
- 根据相机视角计算焦距

- H = W = 400 # 匹配训练时的分辨率

- focal = W / (2 \* np.tan(camera\_angle\_x / 2))

- 开始渲染所有测试视角

- 使用正确的焦距进行渲染

- rgb\_map = render\_novel\_view(model, c2w, H=H, W=W, focal=focal,

```
device=device)
```

调用render\_novel\_view函数渲染当前视角

传入相机外参矩阵、图像尺寸、焦距等参数

- 从渲染的图像创建GIF动画

## 10.5 Renderer.py文件

```
10.5.1 def save_rendered_image(rendered_colors,  
image_width, image_height, output_path)
```

- 将渲染的图像保存到文件
- 输入： rendered\_colors (torch.Tensor): 渲染的颜色值 (N\_rays, 3)
  - image\_width (int): 输出图像的宽度
  - image\_height (int): 输出图像的高度
  - output\_path (str): 保存输出图像的路径
- 将渲染的颜色重塑为图像尺寸
- 将浮点数值转换为8位颜色值 (0-255范围)

# 十一、解决问题和优缺点总结

## 11.1 NeRF解决的核心问题

### 1. 3D场景表示问题

传统方法的问题：

- **点云**: 离散，难以渲染连续表面
- **体素网格**: 内存消耗巨大，分辨率有限
- **网格**: 难以表示复杂几何和材质
- **隐式函数**: 只能表示几何，无法表示外观

NeRF的解决方案：

- 用**连续神经网络**表示3D场景
- 同时学习**几何**（密度）和**外观**（颜色）
- 支持**任意分辨率**的渲染

## 2. 新视角合成问题

**传统方法的问题：**

- 需要大量多视角图像
- 视角间插值效果差
- 无法处理复杂光照和反射

**NeRF的解决方案：**

- 从**少量输入图像**学习完整3D场景
- 生成**任意新视角**的高质量图像
- 保持**视角一致性和光照连续性**

## NeRF的核心思想

---

### 1. 连续场景表示

```
# 传统方法：离散表示
voxel_grid = np.zeros((64, 64, 64, 4)) # 体素网格

# NeRF：连续表示
def nerf(x, y, z, theta, phi):
    # 输入：3D坐标 + 视角方向
    # 输出：密度 + RGB颜色
    return density, rgb
```

### 2. 体积渲染

```
# 沿射线积分计算像素颜色
def render_pixel(ray_origin, ray_direction):
    # 1. 沿射线采样3D点
    points = sample_points_along_ray(ray_origin, ray_direction)

    # 2. 查询每个点的密度和颜色
```

```
densities, colors = nerf(points)

# 3. 体积渲染积分
final_color = volume_rendering_integral(densities, colors)

return final_color
```

## NeRF的优点

---

### 1. 高质量渲染

- **照片级真实感**: 生成非常逼真的图像
- **细节丰富**: 能捕捉复杂的几何和材质细节
- **视角连续**: 新视角之间过渡平滑

### 2. 内存效率

```
# 传统体素网格
voxel_grid = np.zeros((512, 512, 512, 4)) # 512MB

# NeRF网络
neural_network = NeRF() # 几MB参数
```

### 3. 连续表示

- **任意分辨率**: 可以渲染任意分辨率的图像
- **平滑插值**: 支持连续的空间和视角插值
- **可微分**: 支持梯度优化

### 4. 统一框架

- **几何 + 外观**: 同时学习形状和材质
- **光照建模**: 考虑视角相关的光照效果
- **端到端训练**: 从图像直接学习3D表示

### 5. 泛化能力强

- **少样本学习**: 从少量图像学习完整场景

- **新视角合成**: 生成训练时未见过的视角
- **场景插值**: 支持场景间的平滑过渡

## NeRF的缺点

### 1. 训练速度慢

- # 训练时间
- # 单个场景: 几小时到几天
- # 高分辨率: 数天到数周
- # 原因: 需要大量射线采样和网络查询

#### 原因分析:

- 每条射线需要采样64-128个点
- 每个点需要网络前向传播
- 训练时需要渲染整个图像

### 2. 推理速度慢

- # 渲染时间
- # 400x400图像: 几分钟
- # 800x800图像: 十几分钟
- # 原因: 每个像素都需要体积渲染

#### 瓶颈:

- 网络推理次数多
- 体积渲染计算复杂
- 无法并行化所有计算

### 3. 内存消耗大

- # 内存使用
- # 训练时: 8-16GB GPU内存
- # 推理时: 4-8GB GPU内存
- # 原因: 需要存储大量中间结果

## 4. 泛化能力有限

- **场景特定**: 每个场景需要单独训练
- **无法泛化**: 不能直接用于新场景
- **数据依赖**: 需要高质量的多视角图像

## 5. 几何表示模糊

- **密度场**: 几何边界不够清晰
- **表面提取**: 需要后处理才能得到网格
- **碰撞检测**: 难以用于物理仿真

## 6. 光照限制

- **静态光照**: 无法处理动态光照
- **材质简化**: 无法表示复杂材质属性
- **反射限制**: 难以处理镜面反射

# 应用场景

## 1. 适合的应用

- **虚拟现实**: 高质量3D场景展示
- **电影制作**: 特效和虚拟场景
- **文化遗产**: 文物数字化保护
- **建筑可视化**: 建筑场景展示
- **游戏开发**: 高质量环境渲染

## 2. 不适合的应用

- **实时渲染**: 速度太慢
- **物理仿真**: 几何表示不够精确
- **大规模场景**: 内存和计算限制
- **动态场景**: 无法处理运动物体

# 改进方向

## 1. 速度优化

- Instant-NGP：使用哈希编码加速
- Plenoxels：使用稀疏体素表示
- TensoRF：使用张量分解

## 2. 泛化能力

- NeRF-W：处理光照变化
- Mip-NeRF：多尺度表示
- NeRF++：处理无界场景

## 3. 动态场景

- D-NeRF：处理动态物体
- NeRF-DT：处理时间变化
- 4D-NeRF：时空表示

## 总结

---

### NeRF的核心价值：

- 用神经网络表示3D场景
- 实现高质量的新视角合成
- 统一几何和外观学习

### 主要优势：

- 渲染质量极高
- 内存效率好
- 表示连续

### 主要劣势：

- 训练和推理慢
- 泛化能力有限
- 几何表示模糊

NeRF代表了3D场景表示的一个重要突破，虽然存在速度限制，但其高质量渲染能力使

其在多个领域有重要应用价值。