应用数学理论与方法
NeRF & 3D Gaussian Splatting

齐林 244050089

1 Introduction

ICCV 的最佳论文奖是马尔奖(the Marr Prize)，以英国神经科学家 David Marr 的名字命名。Marr 作为计算机视觉的开创者曾经提出过一个系统的视觉理论：计算机视觉的**终极问题**是输入二维图像，输出是由二维图像“重建”出来的三维物体的位置与形状；而其他的一些诸如识别、检测等任务，只能被称为**“模式识别”(Pattern Recognition)**问题，而不能被称为**“计算机视觉”(Computer Vision)**问题。这其中的差别在于，Marr 证明如果终极问题能被解决，那么其他的问题都能够被解决。所以从整个计算机视觉的领域来讲，**NeRF**[1] 所解决的问题就是计算机视觉最根本的问题，它所展示的效果是计算机视觉领域最根本的进步。

而后续的 **3D Gaussian Splatting(3DGS)**[2] 则是计算机视觉领域上三维重建的又一大迈步，它通过引入高斯分布作为基本的渲染单元，对三维空间中的点进行更加高效、连续的表示和渲染。与传统的点云或网格表示方法相比，3DGS 利用空间中高斯核的连续分布来表示物体的形状和纹理，使得渲染过程能够更自然地捕捉细节，同时减少计算复杂度。这种方法通过优化高斯核的参数（包括位置、方差和颜色等），能够快速生成逼真的视图合成效果，并且在实时渲染和交互场景中表现出优异的性能。它不仅显著提升了三维重建的效率，还为动态场景建模和高保真图像生成提供了全新的可能性，成为计算机视觉领域在三维重建研究中的重要进展之一。

2 NeRF

NeRF 的核心思想是：人眼或者相机观察三维场景的过程是，给定一个相机的位置和（朝向）方向，根据三维场景的参数，可以渲染得到一张投影的图片。NeRF 将三维场景用 MLP 表示，前向的网络计算就和人眼或者相机观察三维场景的过程一致，当整个计算过程都可微的时候，通过渲染图片的监督，就可以对 MLP 进行优化，“学出”三维场景的“隐式”参数，如 Figure 1 所示。

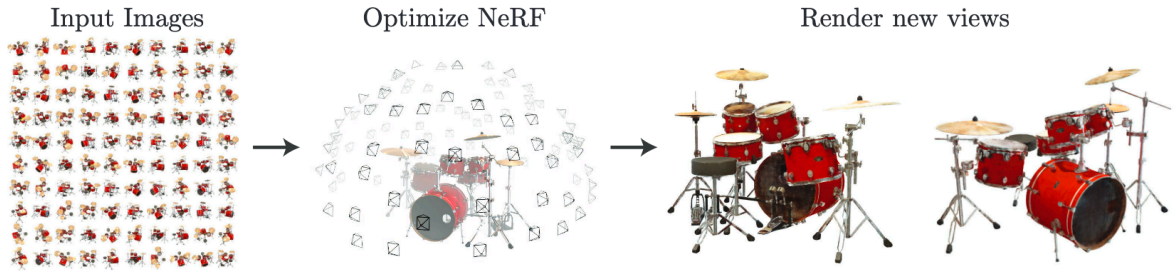

Figure 1: NeRF Forward.

2.1 Volume Rendering

若给定相机的姿态（Camera Pose），从 NeRF 的模型中获得一张输出的图片，关键就是获得每一个图片每一个像素坐标 (x, y) 的像素值。而该点的像素值的计算方法为：从相机光心发出一条射线（Camera Ray）经过该像素坐标，途径三维场景很多点，这些“途径点”或称作“采样点”的某种累加决定了该像素的最终颜色。这个过程也被称作**“体渲染”**。

像素的颜色由 (2.1) 体渲染公式计算出，其中 $\mathbf{c}$ 表示颜色，σ 表示密度，$\mathbf{r}, \mathbf{d}$ 分别表示 Camera Ray 上的距离和方向，t 表示在 Camera Ray 上采样点离相机光心的距离。

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt, \text{where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right). \quad (2.1)$$

这个公式的得出涉及到一定的光学知识，接下来会从物理的角度进行解释。首先，因为光路可逆，为了计算方便，在计算像素平面某点的像素值时，一般通过相机发射出的射线来采样空间中的点并计算它们颜色的某种累加。如 Figure 2 所示，红色的箭头表示从相机光心发射出去的光线，黑色的圆柱体为粒子碰撞、光发生吸收、反射的区域，多种颜色的小球为粒子，橙色的箭头为实际的光线的路径（碰到小球的光线即停下）。

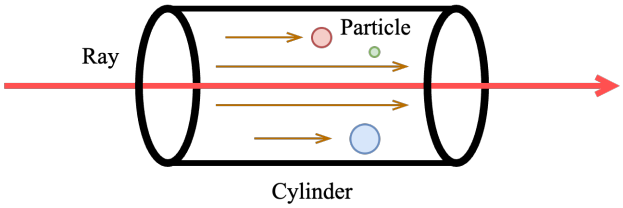

Figure 2: The rendering of NeRF.

考虑一条光线，这条光线上的光照强度为 $I(s)$，其中 s 表示光线上的位置。在光线上极小的一段路径，即 Figure 2 中的圆柱体，其初始位置为 s，长度为 $\Delta s \to 0$，圆柱体的底面面积为 E，内部粒子密度为 $\rho(s)$，其中的粒子都是半径为 r 的小球。

所有碰到粒子的光线都会被吸收或者反射，所以计算光被吸收的概率就可以知道有多少光能够通过该段路径。因为 $\Delta s \to 0$，粒子可以被近似当作平铺在圆柱体内部，粒子所占用的面积为 $E\Delta s\rho(s)\pi r^2$，即粒子数量与粒子截面面积的乘积。粒子所占用的面积与整个圆柱体地面积的比值为：

$$\frac{E\Delta s\rho(s)\pi r^2}{E} = \Delta s\rho(s)\pi r^2$$

所以光通过这段路径之后光照强度变为：

$$I(s+\Delta s) = (1 - \Delta s \rho(s) \pi r^2) I(s)$$

光强度的变化量：

$$\Delta I = I(s+\Delta s) - I(s) = -\Delta s \rho(s) \pi r^2 I(s)$$

也即：

$$\frac{dI(s)}{ds} = -\rho(s) \pi r^2 I(s) \tag{2.2}$$

记 $-\rho(s)\pi r^2$ 为 $\sigma(s)$，则 (2.2) 的解为：

$$I(s) = I(0) e^{\int_0^s -\sigma(t)dt} \tag{2.3}$$

在式 (2.3) 中，记 $T(s) = e^{\int_0^s -\sigma(t)dt}$，则有：

$$I(s) = I(0) T(s)$$

其中 $T(s)$ 的物理意义是透明度，表示当光线达到 s 处时，光强保留的幅度。实际上像素的颜色由那些反射光的粒子决定，而与透过光的部分无关。记 $F(s) = 1 - T(s)$ 表示不透明度，表示当光线达到 s 处的时候，光强反射的幅度，可以近似认为 s 处的粒子颜色为 $\mathbf{c}(s)$，则最终的颜色输出可以表示为：

$$\begin{aligned} E(c) &= \int_0^\infty F'(s) \mathbf{c}(s) ds \\ &= \int_0^\infty T(s) \sigma(s) \mathbf{c}(s) ds \end{aligned} \tag{2.4}$$

式 (2.4) 和 NeRF 原文中的式 (2.1) 已经有非常的相似性了。一方面，NeRF 将 $\sigma(s)$ 密度建模为一个仅和采样点三维坐标相关的量，将 $\mathbf{c}(s)$ 颜色建模成一个采样点三维坐标以及相机光线方向都有关的量；另一方面，在实际计算的时候，往往会选择 Camera Ray 上一个最近的点和一个最远的点，只计算两点之间的粒子对最终颜色的贡献。最终便可得到 NeRF 体渲染的最终公式。

2.2 黎曼和形式的推导

在实际计算的时候，如 (2.4) 的积分形式是无法直接用代码实现的，需要转换成黎曼求和的形式。假设把采样的最近端和最远端之间划分为 N 个小区间，第 i 个区间在 Camera Ray 上的位置就是 $\left[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)\right]$。该小区间的积分可以表示为：

$$\begin{aligned} C(\mathbf{r})_i &= \int_{t_i}^{t_{i+1}} T(t) \sigma(\mathbf{r}(t)) \mathbf{c}(\mathbf{r}(\mathbf{t}), \mathbf{d}) dt \\ &= \int_{t_i}^{t_{i+1}} \exp\left(- \int_{t_n}^{t} \sigma(s) ds \right) \sigma_i \mathbf{c}_i dt \end{aligned} \tag{2.5}$$

在 (2.5) 函数中，密度 $\sigma(\mathbf{r}(\mathrm{t}))$ 和颜色 $\mathbf{c}(\mathbf{r}(\mathbf{t}), \mathbf{d})$ 都可以在这个小区间内近似为常数，直接由 MLP 的输出来确定。而 $T(t)$ 的值相对于小区间不能忽略，所以进一步化简 (2.5) 可以得到：

$$
\begin{aligned}
C(\mathbf{r})_i &= \sigma_i \mathbf{c}_i \int_{t_i}^{t_i+1} \exp\left(-\int_{t_n}^{t} \sigma(s) ds\right) dt \\
&= \sigma_i \mathbf{c}_i \int_{t_i}^{t_i+1} \exp\left(-\int_{t_n}^{t_i} \sigma(s) ds\right) \exp\left(-\int_{t_i}^{t} \sigma(s) ds\right) dt \\
&= \sigma_i \mathbf{c}_i T_i \int_{t_i}^{t_i+1} \exp\left(-\int_{t_i}^{t} \sigma(s) ds\right) dt \\
T_i &= \exp\left(-\int_{t_n}^{t_i} \sigma(s)\right) ds
\end{aligned}
$$

其中嵌套积分项可以直接求解出结果，得到：

$$
\exp\left(-\int_{t_i}^{t} \sigma(s)\right) ds = \exp(-\sigma_i(t - t_i))
$$

从而可以得到：

$$
\begin{aligned}
C(\mathbf{r})_i &= \sigma_i \mathbf{c}_i T_i \cdot \frac{e^{-\sigma_i(t-t_i)}}{-\sigma_i} \Big|_{t_i}^{t_{i+1}} \\
&= \mathbf{c}_i T_i \big(1 - e^{-\sigma_i \delta_i}\big)
\end{aligned}
$$

求和后：

$$
C(\mathbf{r}) = \sum_{i=1}^{N} \mathbf{c}_i T_i \big(1 - e^{-\sigma_i \delta_i}\big), \text{where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_i \delta_i\right) \tag{2.6}
$$

式 (2.6) 就是式 (2.1) 的离散形式，其中求和对应积分符号，T_i 对应 $T(t)$，$\mathbf{c}_i$ 对应 $\mathbf{c}(\mathbf{r}(t), \mathbf{d})$，$\sigma_i \delta_i$ 对应 $\sigma(\mathbf{r}(t))$。

2.3 位置编码

NeRF 的采样点是一个五维向量，其中坐标三维，视角方向二维，经过 MLP 输出颜色与密度。设想辐射场中距离很近即采样向量很相似的两点，输出具有相似的颜色和密度直觉上是合理的。对于两个很近的点，导数决定了这两点的输出能够有多大的差异，如果相近的点的输出难以拉开差距，那么渲染结果就倾向于模糊、平滑，也就是图像处理常提到的低频信号，反之如果想实现结果高频信息足够清晰，则需要模型能对相近点的输出差异较大，对应较大导数。回到乘 weight 和加 bias 的本质，MLP 输出的其中一维大概是这个形式的:

$$(w+\mathrm{ww}+\cdots)x_0+(w+\mathrm{ww}+\cdots)x_1+\cdots+(w+\mathrm{ww}+\cdots)x_n \\ +(b+\mathrm{wb}+\mathrm{wbw}+\cdots)$$

对输入向量求导会发现导数是 weight 多项式，多项式的值先不提，只考虑一个输入维度会发现偏导数是恒定的，即如果自变量在线性空间朝着某个直线方向改变，只能产生等比例的输出变化。

所以理论上来说网络的输出会变成渐变色一块的样子，但是激活函数的加入让 MLP 变为非线性函数，让倒数可以有更丰富的变化来描述不同位置的辐射场，但是直观感觉上只能对线性空间产生一定变形的效果。从结果来看也是，平滑的低频信息依然占据主要地位。

MLP 的导数形式导致相似输入容易产生相似输出，进而难以产生高频信号的问题。一共就五个分量，基于线性函数的 MLP 难以较好地用导数区分开输出。既然导数难以动手，那么就可以引入 **Positional Encoding** 的核心动机：让原本相似的输入不再那么相似。回到最开始的输入：三维坐标，二维视角方向，一共五维。所谓不相似就是距离远一些，五维空间是满秩的，而 Positional Encoding 的做法就是，映射到更高维的空间。

NeRF 在进行 Positional Encoding 时，首先会进行一次频率扩展：对输入的每个坐标分量进行变换，生成一组具有不同频率的正弦和余弦值。然后对于每个坐标 $p=(x,y,z)$，将每个坐标分量 p_i 转换为不同频率的正弦和余弦函数，具体公式如下：

$$\mathrm{PE}(p_i)=(\sin(2^0\pi p_i),\cos(2^0\pi p_i),\cdots,\sin(2^L\pi p_i),\cos(2^L\pi p_i))$$

其中，p_i 表示空间坐标中的某一个维度，L 是一个超参数，表示编码的频率的最大层数，2^L 控制了频率的增长。每个坐标分量会被映射到一个较高维的空间，其中包含了不同频率的正弦和余弦值。原文中 NeRF 对坐标三维取 $L=10$，视角方向取 $L=4$，使用 Positional Encoding 之后 MLP 的输入维度从五维变成了 $(3*10*2+2*4*2)=76$ 维，维度大了很多，虽然没有办法保证满秩或尽量分散，但是相较于原来的五维来说效果也会更好。

3 Gaussian Splatting

在 NeRF 中，沿着一个像素，发出一条射线，然后这条射线“射向体数据”（在 NeRF 里就是沿着光线进行采样，然后查询采样点的属性）的过程。这个过程可以归结为一种 Backward mapping。那么这就会引发一定的思考，是否有一种 Forward mapping 的办法。形式上，就是将整个“体数据”投影到此时位姿所对应的图像平面。这种办法的前提就不能是用 NeRF 那样的隐式表达了，需要一些显式的表达才能支持这样直接的投影。例如以三个顶点长成的三角面基元(primitive)，然后将这些许多的三角面直接投影到成像平面上，判断哪些像素是什么颜色，当有多个三角形投影时，根据它们的“深度”来判断前后顺序，然后进行 Alpha Compositing。

无论是 Backward mapping 还是 Forward mapping，这个过程都涉及到将连续的表示变成离散的。在 Backward mapping 里，是对场进行采样；在 Forward mapping 里，是需要直接生成出基元，这也是一种连续化为离散。为了理解在这个过程中，高斯分布为什么重要，需要涉及到信号与系统中的概念，要清楚此时引入信号与系统里的工具的目的是

什么。回想刚才三角面基元的情景，在实际情境中，接触不到“连续”的表达，比如三角面，只会记录它的三个顶点。当投影完成后，只能做一些有限的操作来阻止“锯齿”，例如对结果进行一个模糊操作，这些操作一般都是局部的。而这样做的目的，本质是“希望用离散的表达来重建原来的信号，进一步在重建好的信号上进行 **resampling**。如果视觉上对处理后的结果看起来没什么混叠或者锯齿上的问题，那就说明 resampling 是成功的。

3.1 Derivation of Sampling

考虑采样率 $f_s = 1/T$，那么考虑一个连续信号 $x_a(t)$ 通过周期性采样得到离散化采样信号 $\hat{x}_a(t)$：

$$\hat{x}_a(t) = \sum_{n=-\infty}^{\infty} x_a(t)\delta(t - nT)$$

假设连续时间信号 $x_a(t)$ 的频谱为 $\hat{X}_a(j\Omega)$，其中 Ω 是频率变量，表示连续信号 $x_a(t)$ 在频域的分布。根据连续时间信号 $x_a(t)$ 的傅立叶变换：

$$X_a(j\Omega) = \int_{\infty}^{\infty} x_a(t)e^{-j\Omega t}dt$$

对 $\hat{x}_a(t)$ 进行傅立叶变换即可得到其频谱。因为 $\hat{x}_a(t)$ 是一个由脉冲序列构成的信号，可以使用傅立叶变换的线性性质 (3.1) 和时移性质 (3.2) 来计算频谱：

$$\mathcal{F}\left\{\sum_{n=-\infty}^{\infty} x_a(t)\delta(t - nT)\right\} = \sum_{n=-\infty}^{\infty} \mathcal{F}\{x_a(t)\delta(t - nT)\} \tag{3.1}$$

$$\mathcal{F}\{\hat{x}_a(t)\} = \sum_{n=-\infty}^{\infty} X_a(j\Omega)e^{-j\Omega nT} \tag{3.2}$$

式 (3.2) 是一个无穷级数，它可以识别为一个周期性信号的傅立叶变换。由于 $\hat{x}_a(t)$ 是由脉冲组成的离散信号，它的频谱 $\hat{X}_a(j\Omega)$ 是周期性的。周期为 $\frac{2\pi}{T}$。因此，离散信号的频谱由连续信号的频谱 $X_a(j\Omega)$ 通过频率平移和周期化得到。采样后信号的频谱可以写成：

$$\hat{X}_a(j\Omega) = \frac{1}{T}\sum_{n=-\infty}^{\infty} X_a\left(j\Omega - jn\frac{2\pi}{T}\right)$$

考虑奈奎斯特采样定理，需要将信号 $\hat{x}_a(t)$ 通过一个低通滤波器，并且同时补偿频谱的幅度，其目的是只让基带频谱能通过。一个理想的低通滤波器对应着一个 Sa 函数。记 $\Omega_s = 2\pi/T = 2\pi_{f_s}$，有：

$$G(j\Omega) = \begin{cases} T_s, & |\Omega| \le \Omega_s/2 \\ 0, & |\Omega| > \Omega_s/2 \end{cases}$$

$$g(t) = \mathrm{Sa}\left(\frac{\Omega_s}{2}t\right) = \frac{\sin\left(\frac{\Omega_s}{2}t\right)}{\frac{\Omega_s}{2}t}$$

直接计算时域上卷积积分的结果：

$$
\begin{aligned}
y(t) = \hat{x}_a(t) * g(t) &= \int_{-\infty}^{\infty} \left[\sum_{n=-\infty}^{\infty} x_a(\tau)\delta(\tau - nT) \right] g(t-\tau)d\tau \\
&= \sum_{n=-\infty}^{\infty} \int_{-\infty}^{\infty} x_a(\tau) g(t-\tau)\delta(\tau - nT)d\tau \\
&= \sum_{n=-\infty}^{\infty} x_a(nT) g(t-nT) \\
&= \sum_{n=-\infty}^{\infty} x_a(nT) \frac{\sin(\frac{\pi}{T}t - n\pi)}{\frac{\pi}{T}t - n\pi}
\end{aligned}
\tag{3.3}
$$

令式 (3.3) 中的 $\frac{\sin(\frac{\pi}{T}t-n\pi)}{\frac{\pi}{T}t-n\pi}$ 为重建核(Reconstruction kernel)。Sa 函数的性质是沿着左右方向都会逐渐递减到 0，这个性质可以称为局部支撑(Local support)，如 Figure 3。从这个角度上看，$y(t)$ 是由重建核和离散数据重新生成的连续函数。

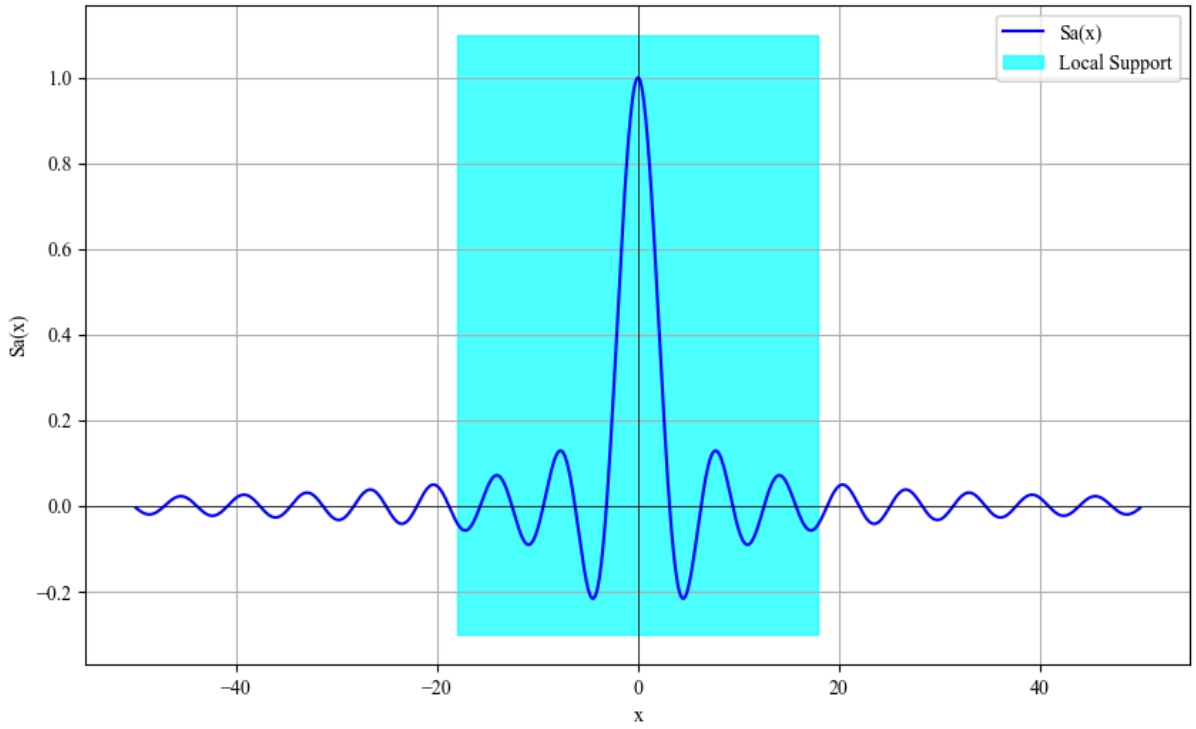

Figure 3: The Sa Function and its Local Support.

在有了上述理论基础后，可以开始介绍 EWA(Elliptical Weighted Average) Splatting[3] 了。EWA 里将要处理的整个图形模型看作对一个连续函数进行不规则的采样的结果，这是一个很自然的假设，例如 3DGS 里的点云，传统的三角面片表示，NeRF 的渲染过程，确实是采样的结果。我们可以像前面推导理想情况下的插值函数一样，用一个重建核来重建这个连续函数 $f_c(\mathbf{u})$：

$$
f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u})
$$

其中 $\mathbf{u}$ 是一个采样的位置, w_k 是采样出的权值, $r_k(\cdot)$ 是重建核, $\mathbb{IN}$ 表示 Integral Number。之后在 EWA splatting 的设计里，第一步是将从源空间里的 $f_c(\cdot)$ 投影到屏幕空间，然后得到一个连续的函数 $g_c(\mathbf{x})$，这一步被表述为：

$$g_c(\mathbf{x}) = \{\mathcal{P}(f_c)\}(\mathbf{x})$$

其中 $\mathcal{P}$ 代表一个投影算子，$\mathbf{x}$ 是屏幕上的二维坐标。将这个算子与求和符号交换位置，即先计算重建核投影后的函数，记作：

$$g_c(\mathbf{x}) = \sum_{k\in\mathbb{IN}} w_k p_k(\mathbf{u})$$

其中 $p_k(\cdot) = \mathcal{P}(r_k(\cdot))$，然后，像数字信号处理里教的那样，要将这个屏幕空间里的连续信号变得带限，线通过一层预滤波器 $h(\mathbf{x})$：

$$g_c'(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x})$$

展开积分，将加权时的求和符号与卷积的积分符号交换位置：

$$\begin{aligned} g_c'(\mathbf{x}) &= \int_{\mathbb{IR}^2} \left\{ \mathcal{P}\left(\sum_{k\in\mathbb{IN}} w_k p_k \right) \right\} (\eta) h(\mathbf{x}-\eta) d\eta \\ &= \sum_{k\in\mathbb{IN}} w_k \int_{\mathbb{IR}^2} p_k(\eta) h(\mathbf{x}-\eta) d\eta \\ &= \sum_{k\in\mathbb{IN}} w_k \rho_k(\mathbf{x}) \end{aligned}$$

其中 $\rho_k(\mathbf{x}) = (p_k \otimes h)(\mathbf{x})$，说明可以像刚才投影时候一样，先计算此时 $p_k(\cdot)$ 预滤波后的结果，再带入计算。框架构造到这里，可能之所以选择“高斯分布”就已经呼之欲出了，因为高斯分布有个很好的性质，两个高斯分布的卷积仍然是高斯分布。上述公式想说明，对采样后整个场景的操作，都可以归结为对重建核进行操作。

3.2 Derivation of Splatting

接下来需要将上面构造的这三步操作，结合到体渲染的具体情景中。如上面所述，Splatting 的过程是反过来的，我们希望将物体投影到平面上。第一步仍然将源空间下的坐标转换到相机空间里，这是一步仿射变换。然后，与其计算相机空间里的每个基元关于平面的投影，再计算哪些部分属于哪些像素，不如直接应用一个投影变换，将从相机原点发射的光线，映射为平行光。这样，被变换后的坐标，就可以很方便的进行体渲染了。变换后得到的空间被称为“光线空间”。这个投影变换并不显然，因为如果直接这么应用一个投影变换，会带来一些问题，之后的部分会具体讨论 。将从源空间映射到相机空间的仿射变换记作 $\varphi(\cdot)$，将相机空间映到光线空间的投影变换记作 $\Phi(\cdot)$。

接下来，从体渲染的公式出发，推导出通常意义下的 Splatting 算法。在 EWA splatting 这篇里，所用的体渲染公式和 NeRF 里的相比，说法上更加严谨，是传统意义上的体渲染公式的写法：low albedo approximation，即低反照近似，大概是指得这个式子本身不考虑颜色随角度变化，于是不怎么拟合那种 non-Lambertian 的东西，但数学形式上是完全

一样的。下面这一部分的内容有两个目的，一个是给出 splatting 框架下做体渲染的方法，另一个是将刚才“连续-离散”的框架在这个例子里“实例化”。

EWA splatting 的原文中，作者为了后续的一个方便（将 3D 积分掉一维变成 2D）以及“光线空间”的引入，修改了一下符号意义。对于光线空间里的坐标 $\mathbf{x}=(x_0,x_1,x_2)^T$，由于 x_2 在的维度会经常被积分掉，所以作者标记了一下 $(\mathbf{x},x_2)^T$。此时体渲染的方程为：

$$I(\mathbf{x})=\int_0^L c(\mathbf{x},\xi)f_c'(\mathbf{x},\xi)e^{-\int_0^\xi f_c'(\mathbf{x},\mu)d\mu}d\xi$$

其中 $I_{\lambda(\cdot)}$ 表示光强，$f_c'(\mathbf{x},\xi)$ 是消光系数，用于建模光的“自我遮挡”，$e^{\cdot}$ 作为衰减因子，$c(\mathbf{x},\xi)$ 表示“发射系数”。这个公式基本的物理原理与 NeRF 类似。

由于将源空间到相机空间记作 $\varphi(\cdot)$，相机空间到光线空间记作 $\Phi(\cdot)$，那么对于之前定义的 $f_c(\mathbf{u})$，对于一个光线空间里的坐标 $\mathbf{x}$，可以依次进行逆变换，然后计算此时光线空间里坐标为 $\mathbf{x}$ 的位置对应的属性。假设 $f_c(\mathbf{u})$ 就是源空间里的消光系数，所以自然有：

$$f_c'(\mathbf{x})=f_c(\varphi^{-1}(\Phi^{-1}(\mathbf{x})))=\sum_{k\in\mathbb{N}}w_k r_k'(\mathbf{x})$$

变换的作用可以直接吸收进重建核中，如果选择高斯分布作为重建核，如果对高斯分布进行简单的旋转，平移，缩放，那么它就还是高斯分布。但是变换 $\Phi^{-1}(\cdot)$ 不保证这样的性质，这点后续将详细讨论。在交换积分和求和符号顺序后将 $f_c'(\mathbf{x})$ 带入体渲染方程：

$$\begin{aligned}
I(\mathbf{x})&=\int_0^L c(\mathbf{x},\xi)\sum_{k\in\mathbb{N}}w_k r_k'(\mathbf{x},\xi)e^{-\int_0^\xi\sum_{j\in\mathbb{N}}w_j r_j'(\mathbf{x},\mu)d\mu}d\xi\\
&=\sum_{k\in\mathbb{N}}w_k\left(\int_0^L c(\mathbf{x},\xi)r_k'(\mathbf{x},\xi)e^{-\sum_{j\in\mathbb{N}}w_j\int_0^\xi r_j'(\mathbf{x},\mu)d\mu}d\xi\right)\\
&=\sum_{k\in\mathbb{N}}w_k\left(\int_0^L c(\mathbf{x},\xi)r_k'(\mathbf{x},\xi)\prod_j e^{-w_j\int_0^\xi r_j'(\mathbf{x},\mu)d\mu}d\xi\right)
\end{aligned}\tag{3.4}$$

此时式(3.4) 中的 $I(\mathbf{x})$ 的形式也是一个加权和，同时它也确实将三维的输入映射到了二维平面，且它和上文中的 $g_c(\mathbf{x})$ 具有一致的意义。所以现在用 $g_c(\mathbf{x})$ 替代 $I(\mathbf{x})$，精准的计算这个式子是困难的，于是就有了 Splatting 的一些基本假设，核心在于变换后的重建核是局部支撑的，也就是说，可以认为在积分从 0 到 L 的过程中，只有那些 $r_k'(\mathbf{x},\xi)$ 显著不为 0 的部分才是感兴趣的。一般一条光路上的采样点，这些支撑集都是不重叠的，于是自然假设对于每个 k 所对应的 $r_k'(\mathbf{x},\xi)$，在其支撑域内，$c(\mathbf{x},\xi)$ 近似为一个常数。这样可以将 $c(\mathbf{x},\xi)$ 从积分里拿出来。同时再对指数函数进行泰勒展开，由 $e^{-x}\approx 1-x$ 可以推出：

$$g_c(\mathbf{x})=\sum_{k\in\mathbb{N}}w_k c(\mathbf{x})\left(\int_0^L r_k'(\mathbf{x},\xi)\prod_j\left(1-w_j\int_0^\xi r_j'(\mathbf{x},\mu)d\mu\right)d\xi\right)\tag{3.5}$$

接下来，为了对齐连乘符号内外的积分，会引入一个 Ignore self-occlusion 的假设。这个假设说的是不那么细致的考虑每个基元内部的不透明度的变化，直接将穿透整个基元后

所消耗的那么多为最小单位。也就是将式 (3.5) 连乘符号内部积分的积分上限调整成 L，此时，整个连乘符号将跟外面的那个积分无关，于是可以得到下式：

$$\begin{aligned} g_c(\mathbf{x}) &= \sum_{k\in\mathbb{IN}} w_k c(\mathbf{x}) \left(\int_0^L r'_k(\mathbf{x},\xi) \prod_j \left(1 - w_j \int_0^L r'_j(\mathbf{x},\mu) d\mu \right) d\xi \right) \\ &= \sum_{k\in\mathbb{IN}} w_k c(\mathbf{x}) \left(\int_0^L r'_k(\mathbf{x},\xi) d\xi \right) \prod_j \left(1 - w_j \int_0^L r'_j(\mathbf{x},\mu) d\mu \right) \end{aligned}$$

记：

$$q_k(\mathbf{x}) = \int_{\mathbb{IR}} r'_k(\mathbf{x},\xi) d\xi \tag{3.6}$$

式 (3.6) 是一个很有用的式子，被称为足迹函数，它描述了对重建核沿某一个维度积分的结果。$q_k(\mathbf{x})$ 实际上代表了三维的重建核映到二维这一过程。于是最终的 $g_c(\mathbf{x})$ 为：

$$g_c(\mathbf{x}) = \sum_{k\in\mathbb{IN}} w_k c(\mathbf{x}) q_k(\mathbf{x}) \prod_{j=0}^{k-1} \big(1 - w_j q_j(\mathbf{x}) \big) \tag{3.7}$$

式 (3.7) 中的 $g_c(\mathbf{x})$ 是真正意义上的一个二维的连续函数，这个公式也是最朴素的意义下的 Splatting。同时有 $g_c(\mathbf{x}) = \sum_{k\in\mathbb{IN}} w_k p_k(\mathbf{u})$ 的形式。此时 w_k 也是一个关于 $\mathbf{x}$ 的连续函数，只是在推导时省略了。所以根据上面的框架，关注 $g'_c(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x})$ 的形态，尝试展开卷积：

$$\begin{aligned} g'_c(\mathbf{x}) &= g_c(\mathbf{x}) \otimes h(\mathbf{x}) \\ &= \sum_k w_k \int_{\mathbb{IR}^2} c_k(\eta) q_k(\eta) \prod_{j=0}^{k-1} \big(1 - w_j q_j(\eta) \big) h(\mathbf{x} - \eta) d\eta \end{aligned} \tag{3.8}$$

想达成 (3.8) 的目的，需要把 $c_k(\cdot)$ 和连乘项拿出来，所以直接的假设 $c_k(\mathbf{x})$ 是常数，同时连乘的这一项也是常数。于是可以得到：

$$\begin{aligned} \prod_{j=0}^{k-1} \big(1 - w_j q_j(\mathbf{x}) \big) &\approx o_k \\ g_c(\mathbf{x}) \otimes h(\mathbf{x}) &= \sum_k w_k c_k o_k \int_{\mathbb{IR}^2} q_k(\eta) h(\mathbf{x} - \eta) d\eta \\ &= \sum_k w_k c_k o_k (q_k(\mathbf{x}) \otimes h(\mathbf{x})) \\ &= \sum_k w_k c_k o_k \rho_k(\mathbf{x}) \end{aligned}$$

经过两次关于 $c_k(\cdot)$ 的假设，现在第 k 个基元的颜色是与坐标位置都是无关的，这正好对应了在 3DGS 中设定每个高斯椭球的颜色为一不随距离变化的定值。

3.3 Derivation of Gaussian

到现在为止，从上述的步骤中是可以觉察到，想要的重建核函数如果有特定的形式，会带来很多方便。假如这个重建核函数和预滤波器作卷积后仍然有和原来一样的表达，假如这个重建核函数经过线性变换后其函数参数也可以被线性的变化，假如这个重建函数沿某个维度积分后也能保持类似的函数结构等，都会有一定的好处。

高斯函数可以符合上述性质，为了和当下的情境更适配，将高斯分布里的 μ 改记作 $\mathbf{p}$(position)。协方差矩阵仍记为 Σ，于是一个位于 $\mathbf{p}$ 处的 n 维的高斯函数 $\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})$ 写作：

$$\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{p})^T\Sigma^{-1}(\mathbf{x}-\mathbf{p})}$$

注意到 Σ 是一个半正定的实对称矩阵，那么它一定可以被正交对角化：

$$\Sigma = \mathbf{Q}^T\Lambda\mathbf{Q}$$

由于它半正定，所以表示特征值的对角阵 Λ 可以被拆为 $\Lambda^{\frac{1}{2}}\Lambda^{\frac{1}{2}}$，于是可以将 Σ 写为：

$$\begin{aligned}\Sigma &= \underbrace{\mathbf{Q}^T\Lambda^{\frac{1}{2}}}_{\sigma}\underbrace{\Lambda^{\frac{1}{2}}\mathbf{Q}}_{\sigma^T} \\ &= \sigma\sigma^T\end{aligned} \tag{3.9}$$

于是 $\Sigma^{-1} = (\sigma^T)^{-1}\sigma^{-1}$，由于对任意一个矩阵 A 都有 $\left(AA^{-1}\right)^T = \left(A^{-1}\right)^T A^T = I$，所以 $\left(A^{-1}\right)^T = \left(A^T\right)^{-1}$，可以不在乎求逆和转置的先后顺序，所以 Σ^{-1} 可以简化写作 $\Sigma^{-1} = \sigma^{-T}\sigma^{-1}$。

接下来，先计算 n 维高斯函数的傅里叶变换：

$$\begin{aligned}\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right] &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\int_{-\infty}^{\infty} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{p})^T\Sigma^{-1}(\mathbf{x}-\mathbf{p})}e^{-j\mathbf{x}^T\omega}d\mathbf{x} \\ &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\int_{-\infty}^{\infty} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{p})^T\sigma^{-T}\sigma^{-1}(\mathbf{x}-\mathbf{p})}e^{-j\mathbf{x}^T\omega}d\mathbf{x}\end{aligned} \tag{3.10}$$

作换元 $\mathbf{z} = \sigma^{-1}(\mathbf{x}-\mathbf{p})$，于是 $\mathbf{x} = \sigma\mathbf{z} + \mathbf{p}$。带入 (3.10) 可得：

$$\begin{aligned}\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right] &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\int_{-\infty}^{\infty} e^{-\frac{1}{2}\mathbf{z}^T\mathbf{z}}e^{-j(\mathbf{z}^T\sigma^T+\mathbf{p})\omega}d\mathbf{x} \\ &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}e^{-j\mathbf{p}^T\omega}\int_{-\infty}^{\infty} e^{-\frac{1}{2}\mathbf{z}^T\mathbf{z}}e^{-j\mathbf{z}^T\sigma^T\omega}d\mathbf{x}\end{aligned}$$

注意到：

$$
\begin{aligned}
&-\frac{1}{2}(\mathbf{z}+j\sigma^T\omega)^T(\mathbf{z}+j\sigma^T\omega)-\frac{1}{2}\omega^T\sigma^T\sigma\omega \\
&=-\frac{1}{2}\left(\mathbf{z}^T\mathbf{z}+\underbrace{j\mathbf{z}^T\sigma^T\omega+j\omega^T\sigma\mathbf{z}}_{\text{scalar}}-\omega^T\sigma\sigma^T\omega\right)-\frac{1}{2}\omega^T\sigma^T\sigma\omega \\
&=-\frac{1}{2}\mathbf{z}^T\mathbf{z}-j\mathbf{z}^T\sigma^T\omega+\frac{1}{2}\omega^T\sigma^T\sigma\omega-\frac{1}{2}\omega^T\sigma^T\sigma\omega \\
&=-\frac{1}{2}\mathbf{z}^T\mathbf{z}-j\mathbf{z}^T\sigma^T\omega
\end{aligned}
$$

所以积分里的指数项可以化为：

$$
\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right]=\frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}e^{-j\mathbf{p}^T\omega}e^{-\frac{1}{2}\omega^T\sigma^T\sigma\omega}\int_{-\infty}^{\infty}e^{-\frac{1}{2}(\mathbf{z}+j\sigma^T\omega)^T(\mathbf{z}+\mathrm{j}\sigma^{\mathrm{T}}\omega)}d\mathbf{x}
$$

由重积分里的换元法，我们知道 σ 正好是 $\mathbf{x}$ 与 $\mathbf{z}$ 之间变换的雅克比矩阵，所以 $d\mathbf{x}=|\sigma|d\mathbf{z}=|\Sigma|^{\frac{1}{2}}d\mathbf{z}$，可以得到：

$$
\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right]=\frac{1}{(2\pi)^{\frac{n}{2}}}e^{-j\mathbf{p}^T\omega}e^{-\frac{1}{2}\omega^T\sigma^T\sigma\omega}\int_{-\infty}^{\infty}e^{-\frac{1}{2}(\mathbf{z}+j\sigma^T\omega)^T(\mathbf{z}+\mathrm{j}\sigma^{\mathrm{T}}\omega)}d\mathbf{z}
$$

进一步，令 $\mathbf{y}=\mathbf{z}+j\sigma^T\omega$，于是有：

$$
\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right]=\frac{1}{(2\pi)^{\frac{n}{2}}}e^{-\frac{1}{2}\omega^T\sigma^T\sigma\omega}\int_{-\infty}^{\infty}e^{-\frac{1}{2}\mathbf{y}^T\mathbf{y}}d\mathbf{y}
$$

由于 $\int_{-\infty}^{\infty}e^{-x^2}dx=\sqrt{\pi}$，由换元法 $x=\frac{t}{\sqrt{2}}$ 得 $\int_{-\infty}^{\infty}e^{-\frac{t^2}{2}}dt=\sqrt{2\pi}$，对于 n 维的情况便可以直接积分：

$$
\int\cdots\int_{-\infty}^{+\infty}e^{-\frac{1}{2}(y_1^2+y_2^2+\cdots+y_n^2)}dy_1dy_2\cdots dy_n=\left(\sqrt{2\pi}\right)^n
$$

于是：

$$
\begin{aligned}
\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x})\right]&=e^{-j\mathbf{p}^T\omega}e^{-\frac{1}{2}\omega^T\sigma^T\sigma\omega} \\
&=e^{-j\mathbf{p}^T\omega}e^{-\frac{1}{2}\omega^T\Sigma\omega}
\end{aligned}
$$

所以，高斯函数的傅里叶变换，仍然是一个高斯函数。可以观察到，变换后的高斯函数，协方差矩阵变为了 Σ^{-1}，均值被释放到了虚轴上。得到这一性质后，结合时域卷积定理，很容易得到，两个高斯函数卷积，结果仍是高斯函数：

$$
\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p_1},\Sigma_1}\right]*\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p_2},\Sigma_2}\right]=\mathcal{F}\left[\mathcal{G}^{(n)}_{\mathbf{p_1}+\mathbf{p_2},\Sigma_1+\Sigma_2}\right]
$$

同时，对高斯分布进行仿射变换 $\varphi(\mathbf{x})=\mathbf{M}\mathbf{x}+\mathbf{b}$，直接计算可以得到：

$$
\begin{aligned}
\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x}) &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{M}^{-1}(\varphi(\mathbf{x})-\mathbf{b})-\mathbf{p})^T\Sigma^{-1}(\mathbf{M}^{-1}(\varphi(\mathbf{x})-\mathbf{b})-\mathbf{p})} \\
&= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{M}^{-1}\varphi(\mathbf{x})-\mathbf{M}^{-1}\mathbf{b}-\mathbf{p})^T\Sigma^{-1}(\mathbf{M}^{-1}\varphi(\mathbf{x})-\mathbf{M}^{-1}\mathbf{b}-\mathbf{p})} \\
&= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}[\mathbf{M}^{-1}(\varphi(\mathbf{x})-\mathbf{b}-\mathbf{M}\mathbf{p})]^T\Sigma^{-1}[\mathbf{M}^{-1}(\varphi(\mathbf{x})-\mathbf{b}-\mathbf{M}\mathbf{p})]} \\
&= \frac{|\mathbf{M}|}{(2\pi)^{\frac{n}{2}}|\mathbf{M}|^{\frac{1}{2}}|\Sigma|^{\frac{1}{2}}|\mathbf{M}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\varphi(\mathbf{x})-\varphi(\mathbf{p}))^T(\mathbf{M}^{-1})^T\Sigma^{-1}\mathbf{M}^{-1}(\varphi(\mathbf{x})-\varphi(\mathbf{p}))} \\
&= |\mathbf{M}|\mathcal{G}^{(n)}_{\varphi(\mathbf{p}),\mathbf{M}\Sigma\mathbf{M}^T}(\varphi(\mathbf{x}))
\end{aligned}
\tag{3.11}
$$

从式 (3.11) 可以看出，进行仿射变换后，高斯分布仍然保持不变。另一个关心的性质是对 n 维高斯分布沿着一个维度进行积分，得到的 $n-1$ 维的分布仍然是高斯的。这一点实际上很好理解，由于指数运算的性质，在积分第 i 个维度时，总可以把所有不跟 x_i 交叉的项全部移出积分符号，最终得到的协方差矩阵中的第行和第列都会消失。在 3DGS 中，关注点是重建核进行投影，然后沿某个方向积分后，重建核还能否有简洁的表达，所以在这里，高斯函数仍然有圆满的答案：

$$
\int_{\mathbb{R}} \mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x}, x_2)dx_2 = \mathcal{G}^{(2)}_{\bar{\mathbf{p}},\bar{\Sigma}}(\mathbf{x})
$$

高斯函数的这些性质，完美符合了对重建核函数要求。

3.4 Derivation of Transformation

最后要处理的是投影变换，在 NeRF 中也处理过投影变换，但那时站在的角度是一个点如何打在相机的成像平面上。换句话说当时关注的是相机内参这一表征投影变换的东西本身，而这次更关注的是同一条光线上的不同点，被投影变换作用后的结果。这里可以不关心投影变换的矩阵形式，考虑相机空间里的坐标 $\mathbf{t} = (t_0, t_1, t_2)^T$，其中 t_2 所在的维度是 z 轴，即相机镜头的方向。记变换后坐标 $\mathbf{x} = (x_0, x_1, x_2)^T$，那么整个变换可以写成：

$$
\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \Phi(\mathbf{t}) = \begin{pmatrix} \frac{t_0}{t_2} \\ \frac{t_1}{t_2} \\ \|(t_0, t_1, t_2)\|^T \end{pmatrix}
$$

前文讲述投影变换时简单一笔带过了一次其作用，具体来说，我们不应忘记我们仍然是处于针孔相机模型这一假设下的。所以其实是存在一个视体(frustum) 的，在这个截锥体里发射的光线并不是平行的。在原始的 NeRF 中这并不是问题，因为我们本来就是从每条不同方向的光线上进行采样，然后查这些坐标的属性的。但在这里每次都计算不同方向的光线可能和哪些高斯椭球是“接触”是比较麻烦的，因为在绘制高斯椭球时我们往往会用其协方差矩阵作为轴的长度，类似一维时熟悉的 $\mu \pm \sigma$，其实在这个椭球之外的函数值就已经很小了。所以我们希望能在一个对于不同光线都是“平行”的空间里完成这件

事。注意 $\Phi(\mathbf{t})$ 的形式，对于同一条直线上的 $\mathbf{t}_i, \mathbf{t}_j$ 是相同的，只有 $\|(t_0, t_1, t_2)\|^T$ 不同，这样的处理会在并行实现上带来方便。

$\|(t_0, t_1, t_2)\|^T$ 其实是有深意的, 如果是常规的透视变换, 这里储存的"深度"往往是 t_2。因为在后面的渲染管线中我们需要这个来判定基元之间的前后顺序，这里需要取 $\mathbf{t}$ 的二范数是因为对于不同的角度的光线，在一个任意角度光线上等距离采样后，映射后的这些点之间的距离也是等间隔的, 由于透视除法把同一条光线上的点的非深度上的坐标都除成一样的了，所以距离的度量只取决于剩下的那个维度。这样带来了一些好处，但因为经过这个变换后，高斯函数的形式将不再封闭。这样保护了那么久的重建核的性质就被打破了。

EWA splatting 里提出一种办法，被称为 Local Affine Approximation。我们至少在每次投影时，其实都是知道高斯函数的均值的，所以我们可以直接计算 $\mathbf{x}_k = \Phi(\mathbf{t}_k)$。可以作线性近似，进行二阶泰勒展开，出现的雅可比矩阵会自然的提供一个线性变换，来提供一个 $\mathbf{t}_k$ 邻域到 $\mathbf{x}_k$ 邻域之间的变换，即:

$$\Phi_k(\mathbf{t}) = \Phi_k(\mathbf{t}_k) + \mathbf{J}_k(\mathbf{t} - \mathbf{t}_k)$$

其中，$\mathbf{J}_k$ 可以由 $\Phi_k(\mathbf{t})$ 直接求偏导得到:

$$\mathbf{J}_k = \begin{pmatrix} \frac{\partial x_0}{\partial t_0} & \frac{\partial x_0}{\partial t_1} & \frac{\partial x_0}{\partial t_2} \\ \frac{\partial x_1}{\partial t_0} & \frac{\partial x_1}{\partial t_1} & \frac{\partial x_1}{\partial t_2} \\ \frac{\partial x_2}{\partial t_0} & \frac{\partial x_2}{\partial t_1} & \frac{\partial x_2}{\partial t_2} \end{pmatrix} = \begin{pmatrix} 1/t_{k,2} & 0 & -t_{k,0}/t_{k,2}^2 \\ 0 & 1/t_{k,2} & -t_{k,1}/t_{k,2}^2 \\ \frac{t_{k,0}}{\|\mathbf{t}_k\|} & \frac{t_{k,1}}{\|\mathbf{t}_k\|} & \frac{t_{k,2}}{\|\mathbf{t}_k\|} \end{pmatrix}$$

利用二阶近似，可以解决投影变换非线性的问题。考虑一个世界坐标系下的坐标 $\mathbf{u}$，仿射变换 $\mathbf{t} = \varphi(\mathbf{u})$ 可以得到在特定相机空间下的坐标 $\mathbf{t}$，然后 $\mathbf{x} = \Phi_k(\mathbf{t})$ 可以得到在光线空间下的坐标表示。由于已经保证了这两个映射是线性的, 所以他们的复合也是线性的, 将其记作 $\mathbf{m}_k(\mathbf{u})$，将这个复合变换展开:

$$\begin{aligned} \mathbf{t} = \varphi(\mathbf{u}) &= \mathbf{M}\mathbf{u} + \mathbf{b} \\ \mathbf{x} = \Phi_k(\mathbf{t}) &= \mathbf{x}_k + \mathbf{J}_k(\mathbf{t} - \mathbf{t}_k) \\ &= \mathbf{x}_k + \mathbf{J}_k(\mathbf{M}\mathbf{u} + \mathbf{b} - \mathbf{t}_k) \\ &= \mathbf{J}_k\mathbf{M}\mathbf{u} + \mathbf{x}_k + \mathbf{J}_k(\mathbf{b} - \mathbf{t}_k) \end{aligned}$$

根据仿射变换前后的高斯函数的结果，可以直接写出:

$$\mathcal{G}^{(n)}_{\mathbf{p},\Sigma}(\mathbf{x}) = |\mathbf{J}_k\mathbf{M}|\mathcal{G}^{(n)}_{\mathbf{m}_k(\mathbf{p}),\Sigma'}(\mathbf{m}_k(\mathbf{u}))$$

变换后的协方差矩阵满足:

$$\Sigma' = \mathbf{J}_k\mathbf{M}\Sigma\mathbf{M}^T\mathbf{J}_k^T \tag{3.12}$$

用这样的矩阵变换，可以直接得到一个高斯椭球在光线空间的表示。也就是说，对于一个位置在 $\mathbf{p}$, 协方差为 Σ 的高斯椭球, 可以直接通过 $\mathbf{m}_k(\mathbf{p})$ 计算出在光线空间里椭球的位置，然后直接用 (3.12) 来得到投影后的协方差 Σ'。然后用 Splatting 的公式进行渲染,

对投影后的三维的高斯函数进行积分，借助高斯函数的性质可以直接跳过协方差矩阵的第三行和第三列，于是最终，整个流程被归结为从这些三维椭球投影到二维椭圆里，进行 Alpha Compositing。

3.5 Derivation of Gradient

在 3DGS 中，每个高斯基元的 $\mathbf{p}$ 和 Σ 都是借助 PyTorch 的自动微分框架来进行随机梯度下降的。其中颜色 c 是靠优化 RGB 三个通道上的球谐系数(Spherical Harmonics) 得到的，不透明度 α 是通过将当前像素点带入此时的高斯基元中计算得到的。优化 $\mathbf{p}, c, \alpha$ 都是比较普通的，但优化 Σ 是通过优化 Σ' 反传回去的。如果直接丢给 PyTorch 的计算图，追踪 (3.12) 这几个矩阵连乘里的乘法操作可不是个很令人满意的事情。所以有必要显式的给出 Σ 的梯度。

协方差矩阵是有物理意义的，我们不能随机初始化一组 3×3 的数组就当 Σ，上文已经用过它实对称矩阵的性质 (3.9)。将这里的对角阵和正交阵赋予一组实际的意义：缩放和旋转。即可重参数化一下 Σ:

$$\Sigma = \mathbf{RSS}^T\mathbf{R}^T$$

与通常使用欧拉角来描述旋转不同，3DGS 里使用归一化后的四元数 $\mathbf{q}$ 来表示旋转，这样可以显式的将待优化的参数从 9 个变为 4 个:

$$\mathbf{q} = q_r + q_i \cdot i + q_j \cdot j + q_k \cdot k$$

$$\mathbf{R}(\mathbf{q}) = 2\begin{pmatrix} \frac{1}{2} - \left(q_j^2 + q_k^2\right) & \left(q_i q_j - q_r q_k\right) & \left(q_i q_k + q_r q_j\right) \\ \left(q_i q_j + q_r q_k\right) & \frac{1}{2} - \left(q_i^2 + q_k^2\right) & \left(q_j q_k - q_r q_i\right) \\ \left(q_i q_k - q_r q_j\right) & \left(q_j q_k + q_r q_i\right) & \frac{1}{2} - \left(q_i^2 + q_j^2\right) \end{pmatrix}$$

根据链式法则，有:

$$\frac{d\Sigma'}{d\mathbf{s}} = \frac{d\Sigma'}{d\Sigma}\frac{d\Sigma}{d\mathbf{s}}$$
$$\frac{d\Sigma'}{d\mathbf{q}} = \frac{d\Sigma'}{d\Sigma}\frac{d\Sigma}{d\mathbf{q}}$$

为了推导各分量的梯度变化，只需要将 Σ' 里的每个元素关于 Σ 求导。为了推导上的简洁，省略雅可比矩阵的下标 k，同时记 $\mathbf{U} = \mathbf{JM}$。以及注意到只关注 Σ' 左上部分的 2×2 的元素，所以可以计算:

$$\begin{aligned}
\Sigma' &= \mathbf{U}\Sigma\mathbf{U}^T \\
&= \begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix}\begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix} \\
&= \begin{pmatrix} U_{11}\Sigma_{11} + U_{12}\Sigma_{21} & U_{11}\Sigma_{12} + U_{12}\Sigma_{22} \\ U_{21}\Sigma_{11} + U_{22}\Sigma_{21} & U_{21}\Sigma_{12} + U_{22}\Sigma_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix} \\
&= \begin{pmatrix} (U_{11}\Sigma_{11} + U_{12}\Sigma_{21})U_{11} + (U_{11}\Sigma_{12} + U_{12}\Sigma_{22})U_{12} & (U_{11}\Sigma_{11} + U_{12}\Sigma_{21})U_{21} + (U_{11}\Sigma_{12} + U_{12}\Sigma_{22})U_{22} \\ (U_{21}\Sigma_{11} + U_{22}\Sigma_{21})U_{11} + (U_{21}\Sigma_{12} + U_{22}\Sigma_{22})U_{12} & (U_{21}\Sigma_{11} + U_{22}\Sigma_{21})U_{21} + (U_{21}\Sigma_{12} + U_{22}\Sigma_{22})U_{22} \end{pmatrix}
\end{aligned}$$

计算 Σ' 关于 Σ 的偏导数：

$$\begin{aligned}\frac{\partial \Sigma'}{\partial \Sigma_{11}} &= \begin{pmatrix} U_{11}U_{11} & U_{11}U_{21} \\ U_{21}U_{11} & U_{21}U_{21} \end{pmatrix} \\ \frac{\partial \Sigma'}{\partial \Sigma_{ij}} &= \begin{pmatrix} U_{1i}U_{1j} & U_{1i}U_{2j} \\ U_{2i}U_{1j} & U_{2i}U_{2j} \end{pmatrix}\end{aligned} \tag{3.13}$$

式 (3.13) 用于在代码中方便地取出与 Σ_{ij} 有关的那些系数。这里求解出的系数分量被应用在 `diff-gaussian-rasterization/cuda_rasterizer/backward.cu` 中的 `__global__ void computeCov2DCUDA()` 函数中：

```
if (denom2inv != 0) {
    // Gradients of loss w.r.t. entries of 2D covariance matrix,
    // given gradients of loss w.r.t. conic matrix (inverse covariance matrix).
    // e.g., dL / da = dL / d_conic_a * d_conic_a / d_a
    dL_da = denom2inv * (- c * c * dL_dconic.x
                         + 2 * b * c * dL_dconic.y
                         + (denom - a * c) * dL_dconic.z
                        );
    dL_dc = denom2inv * (- a * a * dL_dconic.z
                         + 2 * a * b * dL_dconic.y
                         + (denom - a * c) * dL_dconic.x
                        );
    dL_db = denom2inv * 2 * (b * c * dL_dconic.x
                             - (denom + 2 * b * b) * dL_dconic.y
                             + a * b * dL_dconic.z
                             );

    // Gradients of loss L w.r.t. each 3D covariance matrix (Vrk) entry,
    // given gradients w.r.t. 2D covariance matrix (diagonal).
    // cov2D = transpose(T) * transpose(Vrk) * T;
    dL_dcov[6 * idx + 0] = (T[0][0] * T[0][0] * dL_da
                            + T[0][0] * T[1][0] * dL_db
                            + T[1][0] * T[1][0] * dL_dc
                           );
    dL_dcov[6 * idx + 3] = (T[0][1] * T[0][1] * dL_da
                            + T[0][1] * T[1][1] * dL_db
                            + T[1][1] * T[1][1] * dL_dc
                           );
    dL_dcov[6 * idx + 5] = (T[0][2] * T[0][2] * dL_da
                            + T[0][2] * T[1][2] * dL_db
                            + T[1][2] * T[1][2] * dL_dc
                           );

    // Gradients of loss L w.r.t. each 3D covariance matrix (Vrk) entry,
    // given gradients w.r.t. 2D covariance matrix (off-diagonal).
    // Off-diagonal elements appear twice --> double the gradient.
    // cov2D = transpose(T) * transpose(Vrk) * T;
    dL_dcov[6 * idx + 1] = 2 * T[0][0] * T[0][1] * dL_da
```

```
                        + (T[0][0] * T[1][1]
                        + T[0][1] * T[1][0]) * dL_db
                        + 2 * T[1][0] * T[1][1] * dL_dc;
    dL_dcov[6 * idx + 2] = 2 * T[0][0] * T[0][2] * dL_da
                        + (T[0][0] * T[1][2]
                        + T[0][2] * T[1][0]) * dL_db
                        + 2 * T[1][0] * T[1][2] * dL_dc;
    dL_dcov[6 * idx + 4] = 2 * T[0][2] * T[0][1] * dL_da
                        + (T[0][1] * T[1][2]
                        + T[0][2] * T[1][1]) * dL_db
                        + 2 * T[1][1] * T[1][2] * dL_dc;
}
```

对于一个 3×3 的协方差矩阵，由于是对称的，所以我们考虑的是它的上三角的 6 个元素，标号为

$$\Sigma = \begin{pmatrix} 0 & 1 & 2 \\ & 3 & 4 \\ & & 5 \end{pmatrix}$$

代码片段是在根据链式法则一层一层的求解，`a,b,c` 是跳过第三行第三列后，协方差矩阵上的值。因为在高斯函数的公式里要用到，所以前面先有一段计算 `dL_da,dL_db,dL_dc` 的过程。之后需要计算 $\frac{dL}{d\Sigma_{ij}}$，计算协方差矩阵第一行第一列的元素，由刚才计算出的关于的偏导，可得对应的系数是：

$$\begin{pmatrix} \underbrace{U_{11}U_{11}}_{a} & \underbrace{U_{11}U_{21}}_{b} \\ U_{21}U_{11} & \underbrace{U_{21}U_{21}}_{c} \end{pmatrix}$$

由于 dL_db 已经乘过 2 了，所以就有了

```
    dL_dcov[6 * idx + 0] = (T[0][0] * T[0][0] * dL_da
                          + T[0][0] * T[1][0] * dL_db
                          + T[1][0] * T[1][0] * dL_dc
                          );
```

同理也可以求出 $\frac{d\Sigma}{d\mathbf{s}}$ 和 $\frac{d\Sigma}{d\mathbf{q}}$：

$$\frac{d\Sigma}{d\mathbf{s}} = \frac{d\Sigma}{d\mathbf{M}}\frac{d\mathbf{M}}{d\mathbf{s}}$$
$$\frac{d\Sigma}{d\mathbf{q}} = \frac{d\Sigma}{d\mathbf{M}}\frac{d\mathbf{M}}{d\mathbf{q}}$$

在 3DGS 原文的补充材料里，作者直接写 $\frac{d\Sigma}{d\mathbf{M}} = 2\mathbf{M}^T$，实际上对于矩阵函数 $F(\mathbf{M}) = \mathbf{M}\mathbf{M}^T$，严格意义上它关于 $\mathbf{M}$ 的梯度应该是：

$$\Delta_{\mathbf{M}} F(\mathbf{M}) = (\mathbf{M}^T \otimes \mathbf{I}_m)(\mathbf{K}_{mm} + \mathbf{I}_{m^2})$$

对于 $\frac{d\Sigma}{d\mathbf{M}} = 2\mathbf{M}^T$ 的推导过程如下：

$$\Sigma = \mathbf{M}\mathbf{M}^T$$

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} i = a^2 + b^2 & j = ac + bd \\ k = ac + bd & l = c^2 + d^2 \end{pmatrix}$$

在反向传播过程中，考察损失 L，它是一个标量：

$$\frac{\partial L}{\partial \Sigma} = \begin{pmatrix} \frac{\partial L}{\partial i} & \frac{\partial L}{\partial j} \\ \frac{\partial L}{\partial k} & \frac{\partial L}{\partial l} \end{pmatrix}$$

关注：

$$\frac{\partial L}{\partial \mathbf{M}} = \begin{pmatrix} \frac{\partial L}{\partial a} & \frac{\partial L}{\partial b} \\ \frac{\partial L}{\partial c} & \frac{\partial L}{\partial d} \end{pmatrix}$$

进一步展开：

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial i} \cdot \frac{\partial i}{\partial a} + \frac{\partial L}{\partial j} \cdot \frac{\partial j}{\partial a} + \frac{\partial L}{\partial k} \cdot \frac{\partial k}{\partial a} \\ &= 2a\frac{\partial L}{\partial i} + c\frac{\partial L}{\partial j} + c\frac{\partial L}{\partial k} \\ &= 2a\frac{\partial L}{\partial i} + 2c\frac{\partial L}{\partial j} \end{aligned}$$

最后一个等号是因为协方差矩阵是个对称矩阵，那么其对称元素的梯度也应该是一样的。每一项都这么计算，就可以得到：

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{M}} &= \begin{pmatrix} 2a\frac{\partial L}{\partial i} + 2c\frac{\partial L}{\partial j} & 2b\frac{\partial L}{\partial i} + 2d\frac{\partial L}{\partial j} \\ 2a\frac{\partial L}{\partial k} + 2c\frac{\partial L}{\partial l} & 2b\frac{\partial L}{\partial k} + 2d\frac{\partial L}{\partial l} \end{pmatrix} \\ &= 2\begin{pmatrix} a & c \\ b & d \end{pmatrix}\begin{pmatrix} \frac{\partial L}{\partial i} & \frac{\partial L}{\partial j} \\ \frac{\partial L}{\partial k} & \frac{\partial L}{\partial l} \end{pmatrix} \\ &= 2\mathbf{M}^T\frac{\partial L}{\partial \mathbf{M}} \end{aligned}$$

于是在 `backward.cu` 的 `__device__ void computeCov3D()` 函数中，就可以简单的一行来完成这件事：

```
// Compute loss gradient w.r.t. matrix M
// dSigma_dM = 2 * M
glm::mat3 dL_dM = 2.0f * M * dL_dSigma;
```

最后就只剩下 $\frac{d\mathbf{M}}{d\mathbf{s}}$ 和 $\frac{d\mathbf{M}}{d\mathbf{q}}$，$\mathbf{M} = \mathbf{RS}$ 实际就是：

$$\mathbf{M} = \begin{pmatrix} s_x\left(1-2\left(q_j^2+q_k^2\right)\right) & 2s_y\left(q_iq_j-q_rq_k\right) & 2s_z\left(q_iq_k-q_rq_j\right) \\ 2s_x\left(q_iq_j+q_rq_k\right) & s_y(1-2(q_i^2+q_k^2)) & 2s_z\left(q_jq_k-q_rq_i\right) \\ 2s_y\left(q_iq_k-q_rq_j\right) & 2s_y\left(q_jq_k-q_rq_i\right) & s_z\left(1-2\left(q_i^2+q_j^2\right)\right) \end{pmatrix}$$

可以直接看出：

$$\frac{d_{M_{ij}}}{ds_k} = \begin{cases} R_{i,k}, & j=k \\ 0, & j \neq k \end{cases}$$

关于四元数 $\mathbf{q}$ 的梯度也能直接计算出来。由于四元数 $\mathbf{q}$ 在写进旋转矩阵前会被强制归一化一下来保证旋转矩阵的性质，所以最后还有一步归一化导致的梯度变化。令 $\mathbf{p}$ 来指代归一化后的四元数，有：

$$\mathbf{p} = \frac{1}{\|\mathbf{q}\|} \cdot \mathbf{q}$$

$$\frac{\partial p_n}{\partial q_n} = \frac{1}{\|\mathbf{q}\|} - \frac{q_n^2}{\|\mathbf{q}\|^3} \quad n \in \{r,i,j,k\}$$

至此，3DGS 相关的需要推导的公式和与代码强相关的内容就结束了。

4 Conclusion

不论是 NeRF 还是 3DGS，都有着与数学和物理非常密切的关系，矩阵论和偏微分方程以及概率论都有着非常大的占比。这些公式在看论文附录和各种博客时，推导花了非常多的时间，涉及到了非常多的我的知识盲区。而且 3DGS 部分计算和优化的内容直接体现在与 CUDA 相关的代码上，也带来了非常多的麻烦和困难。

参考文献

[1] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[2] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering.," *ACM Trans. Graph.*, vol. 42, no. 4, pp. 139–131, 2023.

[3] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "EWA splatting," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 3, pp. 223–238, 2002, doi: 10.1109/TVCG.2002.1021576.