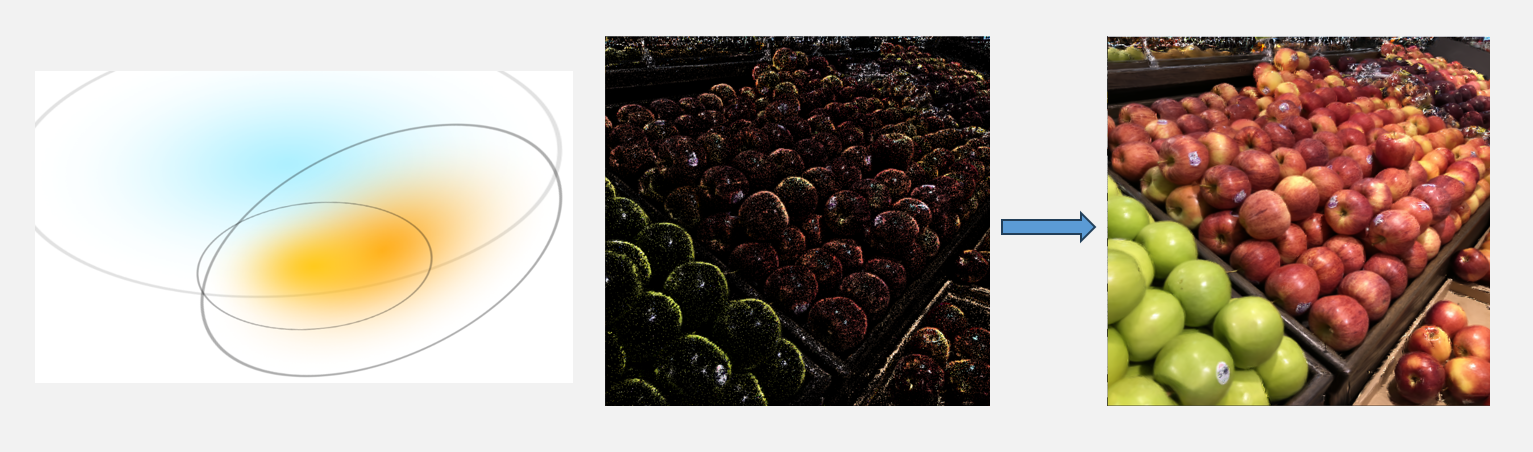
**3D高斯泼溅内存占用的优化方法**

**1.3D Gaussian Splatting介绍**

3D高斯泼溅在新视角合成中提供了出色的视觉质量，具有快速的训练和实时渲染能力这种技术无缝整合了基于点的渲染和泼溅的原理，通过基于泼溅的光栅化实现了快速渲染和可微计算。该方法的核心思想是将一组2d图像转换成点云，然后将点云变成3d空间中的高斯椭球，每个高斯椭球都拥有位置、大小、颜色和不透明度等属性。当把这些椭球体混合在一起时，就可以产生从任何角度渲染的完整的可视化效果



**2.论文创新点**

3DGS方法提供了出色的视觉质量，训练和渲染速度都很快，但是对于存储和传输的内存要求过大，仍有很大的提升空间。

作者认为3DGS中涉及使用内存的主要有三个部分：

1.3DGS构建的场景会出现较多的冗余高斯核，理想的增密策略应避免这种情况。作者通过确定哪些高斯点是多余的来解决密度过大的问题；这在很大程度上取决于观察到的细节的规模和分辨率。

2.3DGS高斯核属性内存占用如下：位置、比例和颜色（每个3个浮点数），旋转（四元数的4个浮点数），不透明度（1个浮点数），以及SH系数(3阶时为45个)，每个图元需要59个浮点数存储，**其中球谐系数占据了高斯核大部分。导致了浪费的内存利用**。

3.**高斯核的一些属性不需要太高的精度**，包括不透明度、比例、旋转和SH系数。

为此作者提出了三种解决方法

**1.提出一种修剪高斯核方法。**

**2.引入自适应选择球谐系数的调整方法。**

**3.基于密码本以及半浮点表示高斯核参数。**

**3. 修剪高斯核方法**

该方法为每个高斯核设定一个球形区域，然后找出球形区域内重叠的其他高斯核的数量。

为了选择球形区域的大小，**我们考虑视图𝑘观察到的单位像素长度投影，并根据高斯核深度投影到世界空间中得到长度b。**

并且长度b是在所有视图中找到相机离该高斯核最近，并且高斯核在摄像机视锥内的视图来计算球体半径bmin。这种选择使冗余测试变得保守，这可以看作是选择确定图元最大采样率的摄像机（视场角、分辨率、深度）。然后，**我们找到围绕高斯核的半径为𝑏 min *√3 / 2的球内相交的其他高斯图元的数量，***（即每个面的面积为𝑏 min 的立方体对角线的一半）。每个高斯由其均值的椭球表示，椭球的轴长度/方向与3DGS定义的高斯的比例/旋转相对应。理想情况下，围绕高斯核的3D区域应由少量高斯核占据。



**为了降低相交计算，首先对每个高斯核进行k-NN搜索，计算最近距离的30个顶点。**此外，由于球体和椭圆的相交计算比较复杂，**作者通过按球的半径缩放椭球的轴，从而化简为椭球和点的相交测试（近似成一个点？），**那么只需要判断点是不是在椭球内部即可，通过计算相交测试的高斯核数量，得出以每个高斯为中心的球形区域的空间冗余值。

计算相交次数可以看作是在感兴趣的点对空间冗余得分场进行采样**；我们使用高斯的中心作为采样位置，然后将这些位置计算的得分传给相交的其它高斯核。**每个椭球会选择相交的所有球形区域中最小的得分。动机很简单：如果高斯与一个不冗余填充的区域重叠，它可能在那里贡献关键细节。

现在每个高斯核都有冗余得分；我们根据得分对高斯核进行排序并过滤那些得分大于自适应阈值𝜐𝑞 = (𝜈 +𝜇𝑠 𝜏)的图元。这里，𝜈和𝜏是所有高斯核冗余得分的均值和标准差。这个自适应阈值修剪了冗余得分比均值多𝜇𝑠标准差的高斯，𝜇𝑠 = 1在实验中设置为1。

**由于高斯核之间不是互相独立的，删除一个高斯核也肯能会改变周围高斯核的冗余度**，删除所有得分大于𝜐𝑞的高斯核可能会导致某些区域过度剔除。所以作者按照不透明度从低到高进行排序，只删除50%的不透明度低的过滤高斯核。因为剔除低不透明度的图元对图像质量影响更小。

作者发现剔除冗余得分小于等于3的图元（即一个图元至少贡献到一个与另外两个图元重叠的区域）会对质量产生不利影响。因此将𝜐𝑞修改为𝜐𝑞 = max(𝜈 + 𝜇𝑠 𝜏, 3)。即不会删除冗余度小于4的高斯核。

作者还通过在训练过程中添加𝑀1稀疏性项来修改损失，使不透明度值尽可能低。这有助于鼓励创建低贡献图元，这在剔除50%冗余候选图元时特别有效。在优化过程中，每1000次迭代应用一次剔除过程。

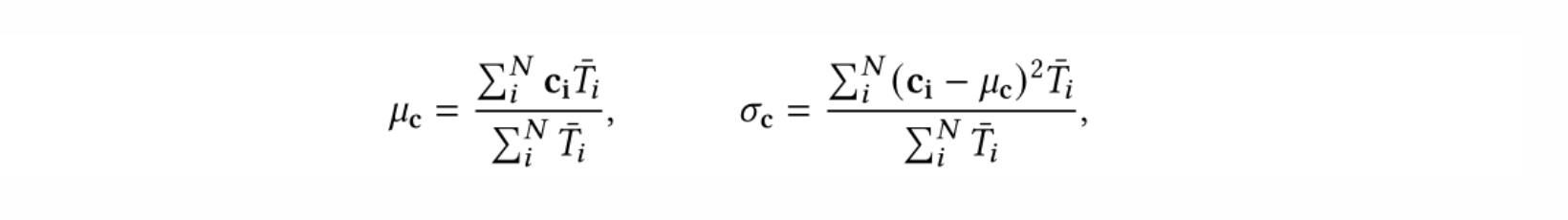
**2. 自适应选择球谐系数**

**在许多情况下，高斯核想表达的事物使用RGB 颜色值就可以足够表达**，所以只要确定高斯核是否需要视角依赖，就可以将不需要的高斯核的球谐系数进行简化。

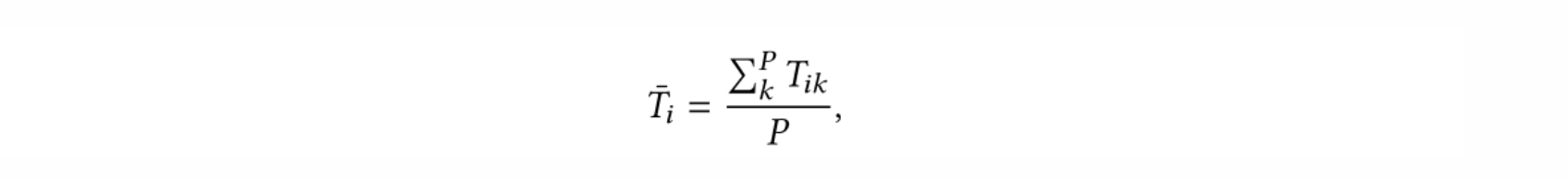
作者通过评估所有输入视图的球谐系数函数来做到这一点；观察颜色在哥哥视图之间是否发生显著变化。然后使用这些信息通过两种方法剔除 SH 带。

第一种方法是，如果一个图元从所有输入视图接收到相同的颜色，则不需要视角依赖效应； 第二种方法是，如果在用较少的低阶球谐系数进行评估时图元的颜色变化不大，则可以忽略高阶球谐系数。

对于第一种方法，我们确定从所有视点评估的高斯核颜色是否具有低方差。如果是这种情况，则可以用 RGB 颜色（0 阶）建模，而不需要表示视角依赖的外观（高阶）。对于每个视点，我们计算每个高斯核的颜色的通道平均值和标准差，并以平均透射率加权。



其中N是包含高斯核的视图数，Ti是高斯核在视图i所分布像素的平均透射率，给出：



平均透射率中， P是高斯核在视图i中分布到的像素数，Tik是高斯核在特定像素k的透射率。

然后，作者将所有方差低于阈值的高斯核的RGB颜色替换为平均颜色，并禁用所有高阶球谐函数。在所有实验中作者将方差阈值设置为 0.04。

对于第二种方法，对于每个视点，仅使用到第r阶的系数评估颜色，每个图元的颜色结果为c\_r，其中 r∈[0,3]。然后，计算完整颜色与剩余三者之间的欧几里得距离，得到三个颜色距离e0,e1,e2。最后，对所有视图计算每个距离值的加权平均值，权重为平均透射率。

最终选择（均值？）低于阈值的最低阶数 r，并移除剩余的高阶系数，阈值设置为 0.04。应用这些步骤后，一些高斯核保持所有阶数，而大多数高斯核最终仅保留 2 个或更少的阶数带，从而无需存储相应的高阶系数。

禁用球谐系数可能导致小的质量下降；为了减轻这种影响，作者在 3DGS 训练的中途（15K 次迭代，即密集化停止时）应用 球谐系数剔除，并允许剩余的优化步骤补偿调整。作者再次通过在 SH 系数上引入稀疏性损失来修改原始 3DGS 优化。从而尽量阻止高斯核使用高阶系数。

**3. 基于密码本以及半浮点表示高斯核参数**

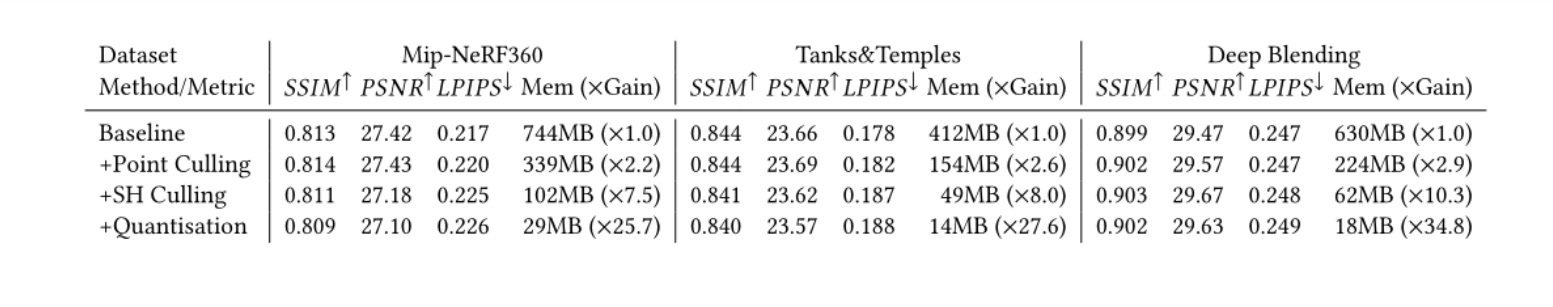
经实验，高斯核的大多数属性只需要存储有限的动态范围和精度就可以有很好效果。

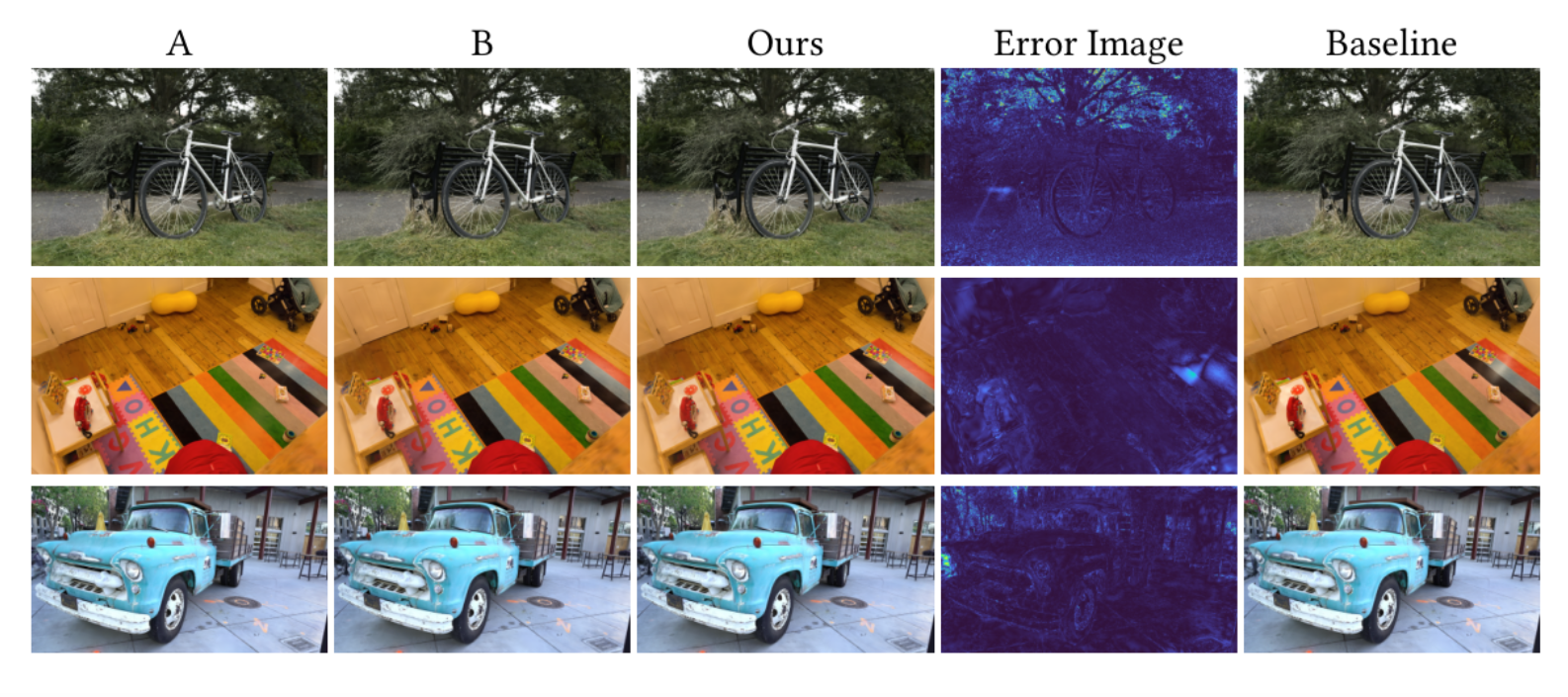
作者使用 K-means 聚类创建一个码本，即，不存储每个高斯核属性的确切值，而是存储到固定大小码本中最接近值的索引。对于向量属性（例如缩放维护一个共享码本，每个分量有一个索引。根据实验表明，1 字节索引允许最大压缩并且质量降解最小，因此码本大小为 256 个条目。

例如，如果存储N个高斯缩放（包含浮点数的 3D 向量）的初始成本是 3Nx4字节，则成本变为 3N+4×256 字节，包括共享码本的大小。

为高斯图元属性使用的密码本如下：一个用于不透明度，一个用于三个缩放分量，一个用于四元数旋转的实部，一个用于虚部，一个用于球谐基系数，15个球谐系数中每个3通道颜色分量用一个密码本。

**高斯简化将内存占用减少到原方法的 32% 到 52% 之间。对 PSNR 的平均影响很小，在 -0.32 到 +0.16 dB 之间，对视觉质量几乎没有影响。**甚至在某些情况下有提升，原因是高斯核和球谐剔除也可以作为正则化策略，迫使优化找到更具泛化性的解决方案。对于存储大小，该方法最终在 3DGS使用的所有数据集上实现了平均 27 倍的大小减少。





**在阅读本篇论文时，作者还没有开源代码，为了进一步加深对论文的理解，我尝试自己实现论文给出的效果，受限于时间因素，我仅仅复现了第一个创新点的代码，其实现效果如下：**

**原始效果**

PSNR 27.39 （540MB）



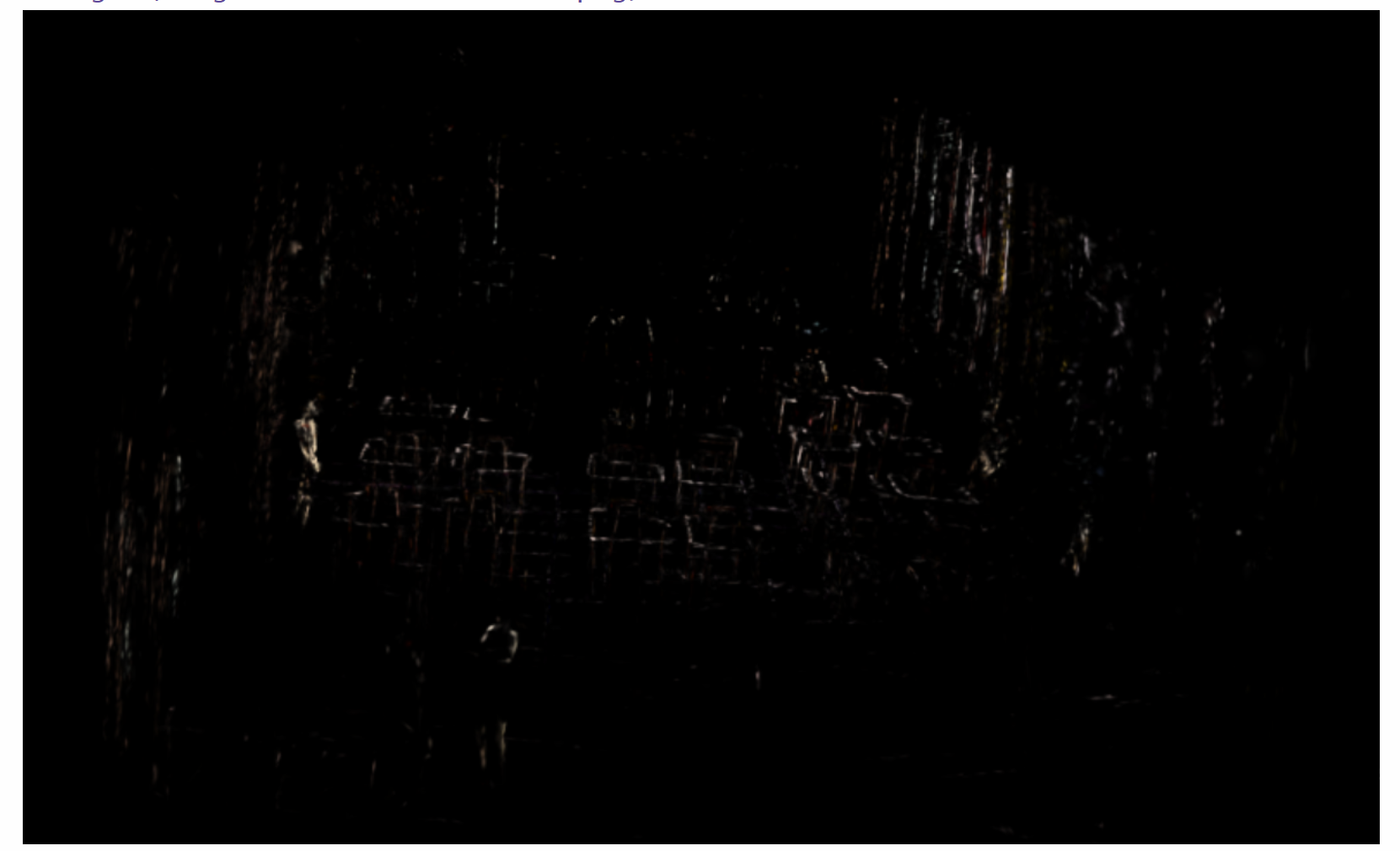
**使用修剪高斯核方法后**

PSNR 27.27 （400MB）



应用高斯核修建方法后，模型的效果略微下降，但是size下降了非常多。

我将删减的高斯核显示出来，发现基本是在一些边缘位置，高斯核容易在边缘位置生成过多的高斯核，从而导致高斯核。



**修改原3dgs的部分代码如下**

***gaussian\_model.py***

修改simple\_knn代码

添加不透明度损失函数

修剪高斯核（主函数）

def reduce\_gaussian(self,cameras):  
   indict = findindict(self.\_xyz.float().cuda())  
   N = indict.size(0) // 30  
   indict = indict.reshape(N, 30)  
   indict = indict.long()  
   xyz\_coords = self.\_xyz[indict]  
​  
   cube\_length = self.count\_mincube(cameras)  
   point\_xyz = self.\_xyz.unsqueeze(1).expand(-1, 30, -1)  
   dists = torch.norm(xyz\_coords - point\_xyz, dim=2)  
​  
   # 筛选符合条件的点（球点相交）  
   valid\_indices = (dists < cube\_length.unsqueeze(1)) & (indict != torch.arange(N).unsqueeze(1).cuda())  
   mask\_intersect = valid\_indices  
   mask\_point = valid\_indices.sum(dim=1)  
      
   mask\_point = self.update\_redundancy(mask\_point,indict,mask\_intersect,N)  
   mask\_point = mask\_point.float()  
​  
   mean\_value = torch.mean(mask\_point)  
   std\_value = torch.std(mask\_point)  
   us = 1  
​  
   valid\_mask = mask\_point > (max(3,mean\_value+std\_value\*us))  
   filtered\_opacity = self.\_opacity[valid\_mask].squeeze()  
      
   # 获取有效顶点的原始索引  
   valid\_indices = torch.nonzero(valid\_mask).squeeze(1)  
   sorted\_opacity, sorted\_idx = torch.sort(filtered\_opacity)  
​  
   # 取前50%的顶点进行删除  
   num\_to\_delete = sorted\_opacity.size(0) // 2  
   indices\_to\_delete = valid\_indices[sorted\_idx[:num\_to\_delete]]  
​  
   mask = torch.zeros(N, dtype=torch.bool).cuda()  
   mask[indices\_to\_delete] = True  
   self.prune\_points(mask)  
​  
   torch.cuda.empty\_cache()

更新冗余值

def update\_redundancy(self,mask\_point, indict,mask\_intersect,N):  
   updated\_redundancy = mask\_point.clone()  
      
   #完全更新  
   """#queue = [i for i in range(N)]  
       while queue:  
           point = queue.popleft()  
           遍历和point相交的顶点，如果当前冗余值比point顶点大，更新冗余值，并且放入队尾"""  
      
   #更新一次  
   intersection\_indices = torch.where(mask\_intersect, indict, torch.tensor(N, device=indict.device)) # 用N填充无效的索引  
   valid\_intersections = intersection\_indices < N  
​  
   min\_redundancies = torch.full((N, 30), torch.iinfo(torch.long).max, dtype=torch.long, device=indict.device)  
   min\_redundancies[valid\_intersections] = mask\_point[intersection\_indices[valid\_intersections]]  
   min\_redundancies, \_ = torch.min(min\_redundancies, dim=1)  
​  
   updated\_redundancy = torch.min(mask\_point, min\_redundancies)  
​  
   return updated\_redundancy

计算高斯核判定区域长度

def count\_mincube(self,cameras):  
   xyz = self.get\_xyz  
   distance = torch.ones((xyz.shape[0]), device=xyz.device) \* 100000.0  
   valid\_points = torch.zeros((xyz.shape[0]), device=xyz.device, dtype=torch.bool)  
   focal\_length = 0.  
   for camera in cameras:  
       R = torch.tensor(camera.R, device=xyz.device, dtype=torch.float32)  
       T = torch.tensor(camera.T, device=xyz.device, dtype=torch.float32)  
​  
       xyz\_cam = xyz @ R + T[None, :]  
       valid\_depth = xyz\_cam[:, 2] > 0.2  
       x, y, z = xyz\_cam[:, 0], xyz\_cam[:, 1], xyz\_cam[:, 2]  
       z = torch.clamp(z, min=0.001)  
​  
       x = x / z \* camera.focal\_x + camera.image\_width / 2.0  
       y = y / z \* camera.focal\_y + camera.image\_height / 2.0  
​  
       in\_screen = torch.logical\_and(  
           torch.logical\_and(x >= -0.15 \* camera.image\_width, x <= camera.image\_width \* 1.15),  
           torch.logical\_and(y >= -0.15 \* camera.image\_height, y <= 1.15 \* camera.image\_height))  
​  
       valid = torch.logical\_and(valid\_depth, in\_screen)  
​  
       distance[valid] = torch.min(distance[valid], z[valid])  
       valid\_points = torch.logical\_or(valid\_points, valid)  
       if focal\_length < camera.focal\_x:  
           focal\_length = camera.focal\_x  
​  
   distance[~valid\_points] = distance[valid\_points].max()  
   cube\_length = distance / focal\_length \* 5  
   return cube\_length