3D Gaussian Splatting算法介绍

探讨三维场景重建与二维图像渲染

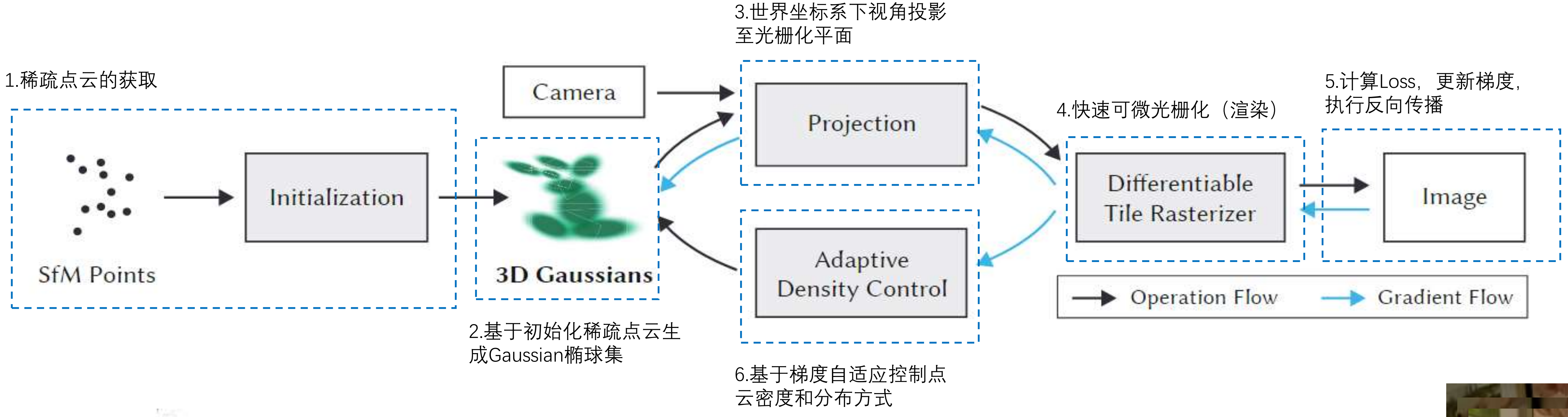
Paper:3D Gaussian Splatting for Real-Time Radiance Field Rendering

3D Gaussian Splatting对于实时辐射场的渲染

Page:https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

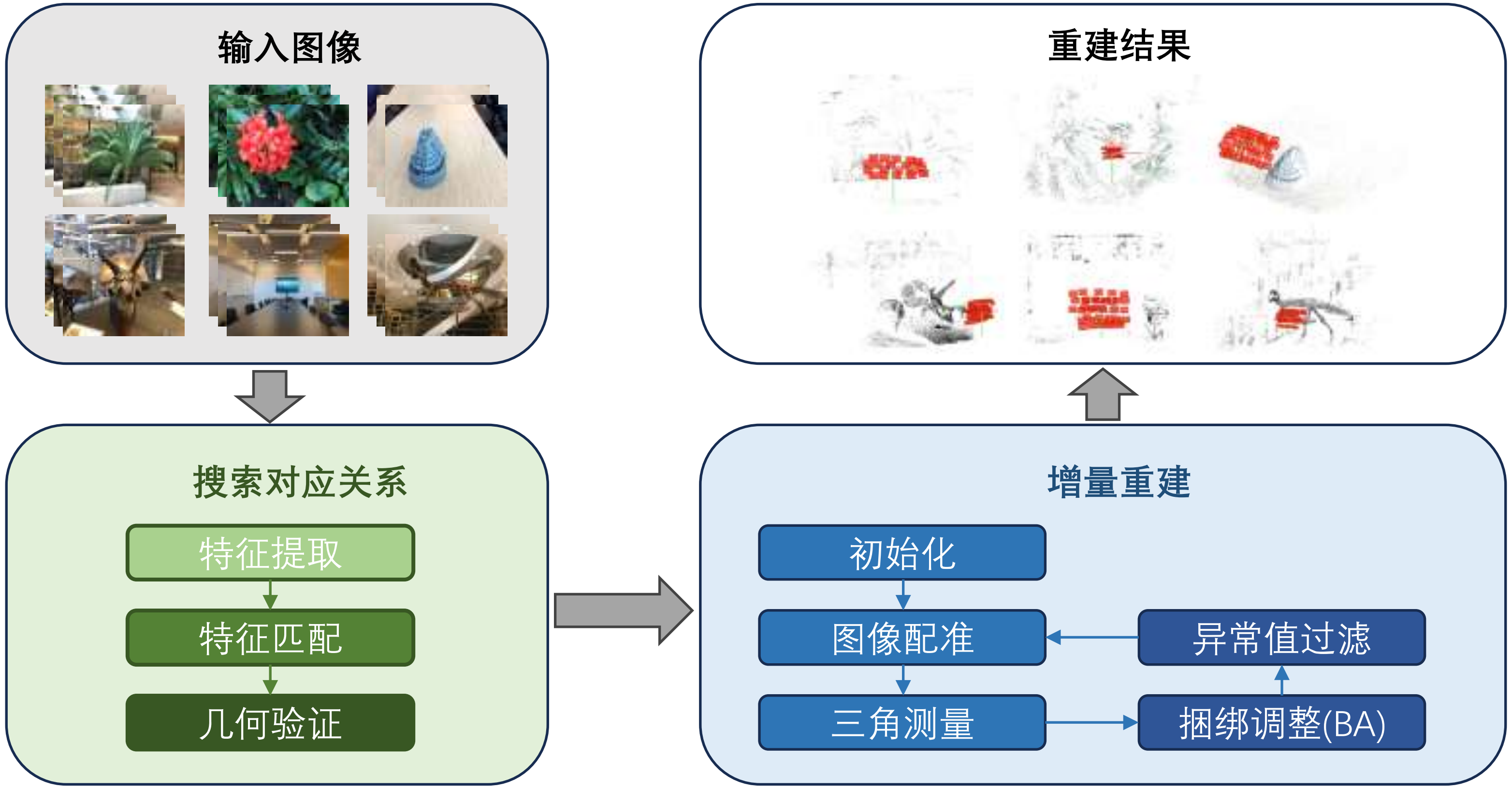
Code:https://github.com/graphdeco-inria/gaussian-splatting

3D Gaussian Splatting算法部件与流程：



Colmap方法

基于三维高斯分布的场景表示——SfM 获取初始点云理论分析



Paper:Structure-from-Motion Revisited

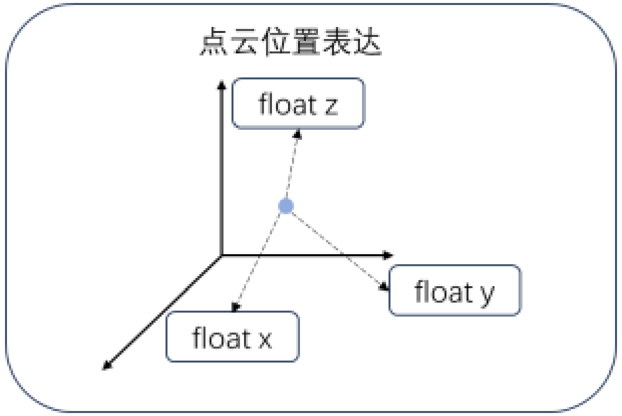
三维高斯分布模型构建的关键技术研究：

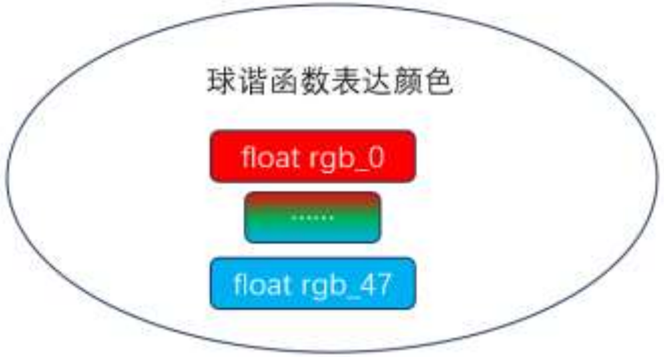
$$G(x) = e^{-\frac{1}{2}(x)^T \Sigma^{-1}(x)}$$

$$G(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n|\boldsymbol{\Sigma}|}} e^{\left(-\frac{1}{2}(x-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(x-\boldsymbol{\mu})\right)}$$

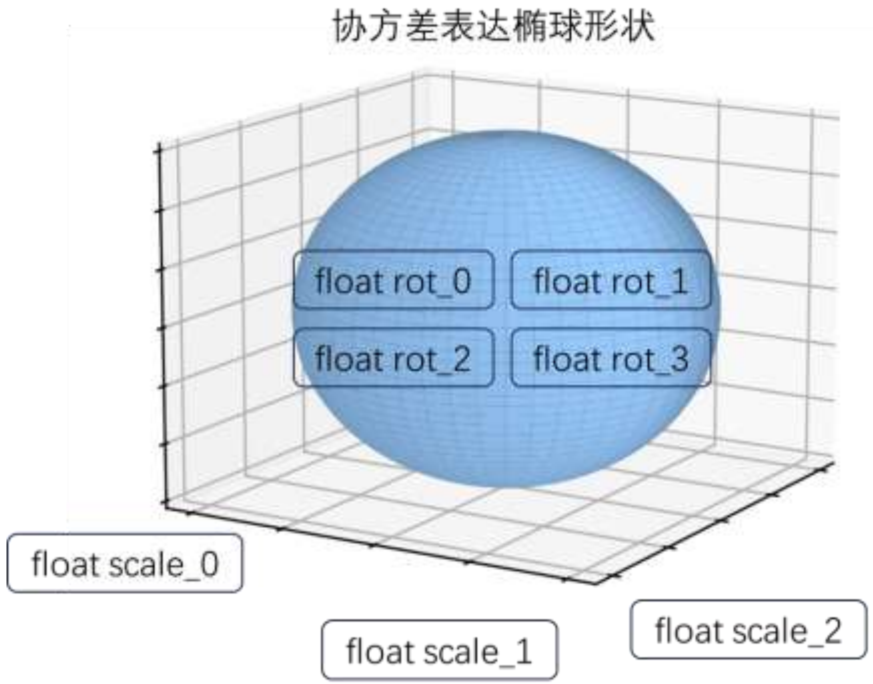
三维高斯分布模型构建的关键技术研究：

基于三维高斯分布的场景表示——预处理点云数据和初始化三维高斯椭球的理论分析





$$G(x) = e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x)}$$



三维高斯分布模型构建的关键技术研究：

形状的表达用的是协方差矩阵Σ，而投影的变化过程是一个非线性近似的过程，

可以用雅可比矩阵J来完成这个操作：$\Sigma' = JW\Sigma W^T J^T$

其中W是世界坐标系到相机坐标系的转换矩阵

雅可比矩阵表示一个多元函数在某一点的局部线性近似。

$$\Sigma' = JW\Sigma W^T J^T$$

三维高斯分布模型构建的关键技术研究：

三维高斯分布的相机模型分析——针孔相机模型与坐标系变换分析

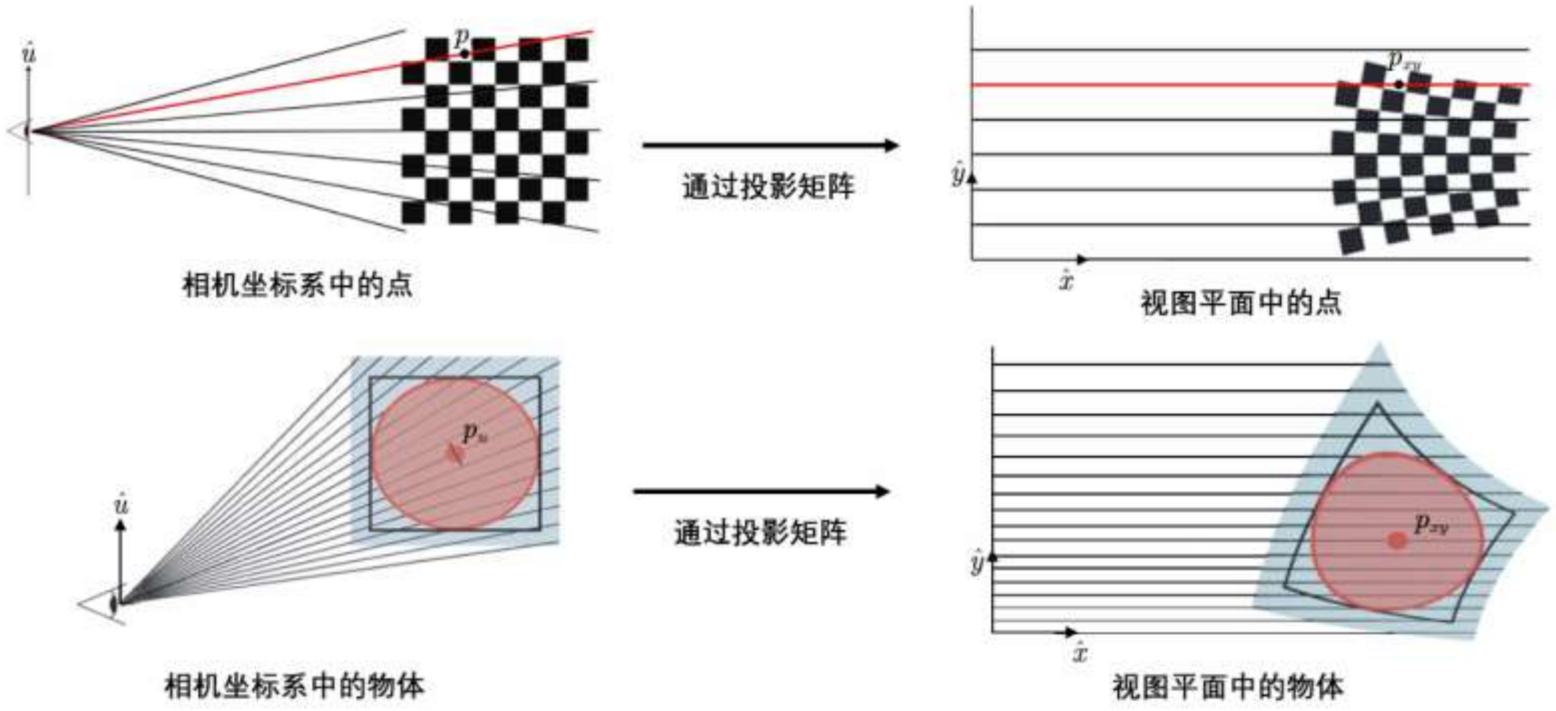
形状的表达用的是协方差矩阵Σ，而投影的变化过程是一个非线性近似的过程，

可以用雅可比矩阵J来完成这个操作:$\Sigma' = JW\Sigma W^T J^T$

雅可比矩阵表示一个多元函数在某一点的局部线性近似。

$$\Sigma' = JW\Sigma W^T J^T$$

$$W = \begin{bmatrix} sR_{3*3} & sR_{3*3}C_{3*1} \\ 0^T & 1 \end{bmatrix}, J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_2} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$



Paper:EWA Volume Splatting

三维高斯分布模型构建的关键技术研究：

三维高斯分布的相机模型分析——投影后的梯度变化与参数优化分析

$$\Sigma' = JW\underline{\Sigma}W^TJ^T$$

$$\Sigma = \underline{RS}S^TR^T$$

$$M = RS$$

$$\frac{\partial M}{\partial q_r} = 2\begin{pmatrix} 0 & -s_yq_k & s_zq_j \\ s_xq_k & 0 & -s_zq_i \\ -s_xq_j & s_yq_i & 0 \end{pmatrix}, \frac{\partial M}{\partial q_i} = 2\begin{pmatrix} 0 & -s_yq_j & s_zq_k \\ s_xq_j & -2s_yq_i & -s_zq_r \\ s_xq_k & s_yq_r & -2s_zq_i \end{pmatrix}$$

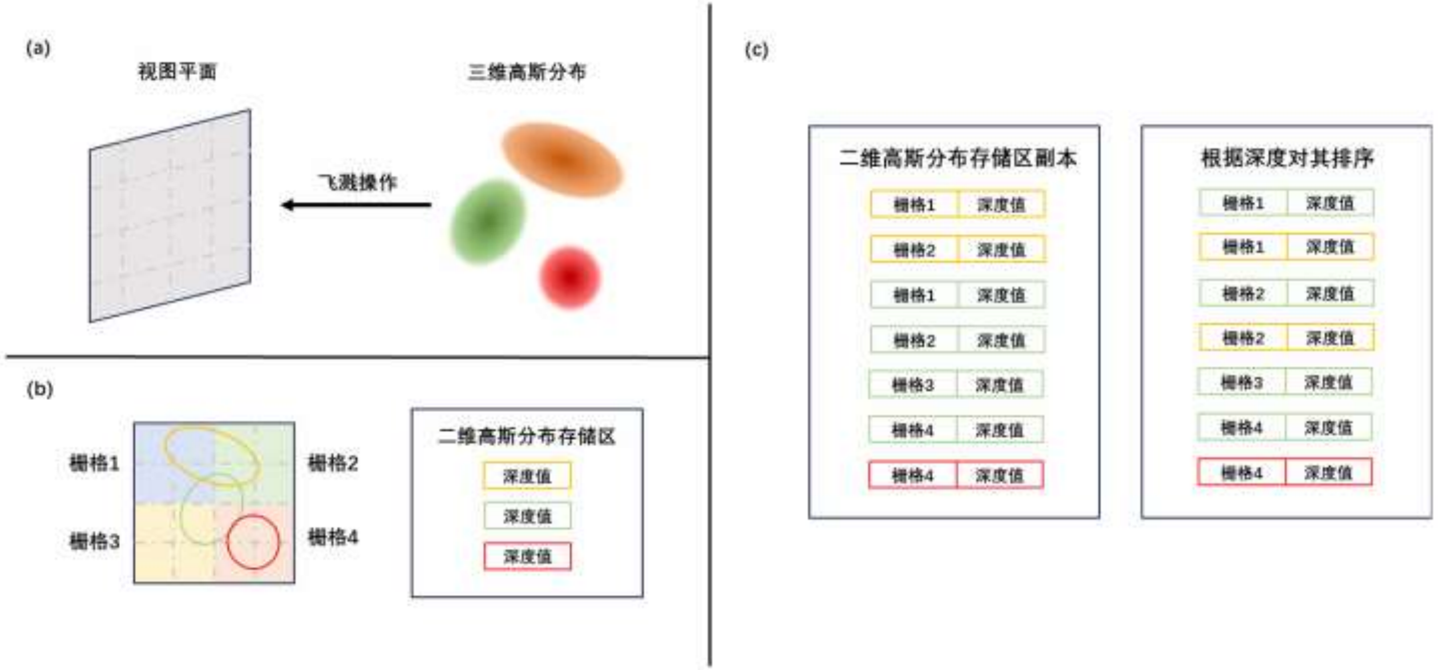
$$\frac{\partial M}{\partial q_j} = 2\begin{pmatrix} -2s_xq_j & s_yq_i & s_zq_r \\ s_xq_i & 0 & s_zq_k \\ -s_xq_r & s_yq_k & -2s_zq_j \end{pmatrix}, \frac{\partial M}{\partial q_k} = 2\begin{pmatrix} -2s_xq_k & -s_yq_r & s_zq_i \\ s_xq_r & -2s_yq_k & s_zq_j \\ s_xq_i & s_yq_j & 0 \end{pmatrix}$$

$$\frac{\partial M_{i,j}}{\partial s_k} = \begin{Bmatrix} R_{i,k} & if j = k \\ 0 & otherwise \end{Bmatrix}$$

三维高斯分布快速渲染的关键技术研究：

三维高斯分布快速渲染的关键技术研究：

快速可微光栅化器的设计



每个视锥只保留置信区间大于99%的Gaussian，剔除后对剩余的Gaussian根据深度快速排序。这样可以减小每个像素都要排序的成本。

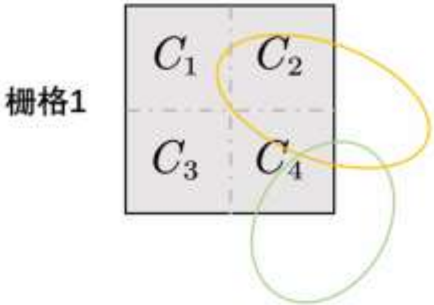
为每个图块生成一个列表，记录每个像素第一个和最后一个深度排序的Gaussian下标。然后对每个图块启动一个线程协助加载Gaussian数据包，对每个像素积分计算所有Gaussian的颜色和不透明度（这就是splatting的代码实现）。一旦有像素的不透明度达到饱和就停止对应线程。

在该图块所有像素都饱和时终止对图块的处理。

渲染公式：$C=\sum_{i\in N} c_i\alpha_i\prod_{j=1}^{i-1}(1-\alpha_j)$

α_i代表当前点$\boldsymbol{i}$的不透明度值，α_j代表$\boldsymbol{i}$之前每个点的不透明度值

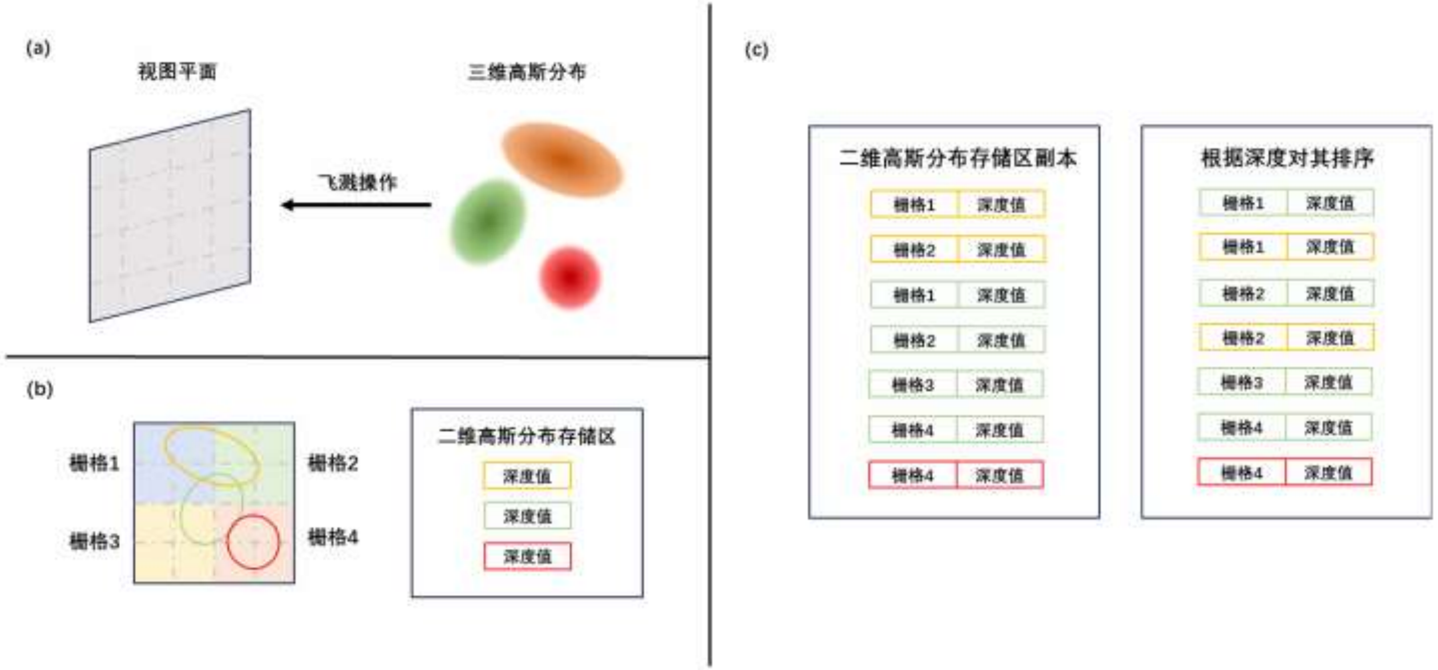
用$1-\alpha_j$进行累乘，作为颜色的权重，代表前面所有点$\boldsymbol{j}$越透明，该点$\boldsymbol{i}$的颜色对渲染的贡献越大



$$C_1=\alpha_1'c_1+\alpha_1'c_2(1-\alpha_1')$$
$$C_2=\alpha_2'c_1+\alpha_2'c_2(1-\alpha_2')$$
$$C_3=\alpha_3'c_1+\alpha_3'c_2(1-\alpha_3')$$
$$C_4=\alpha_4'c_1+\alpha_4'c_2(1-\alpha_4')$$

三维高斯分布快速渲染的关键技术研究：

快速可微光栅化器的设计



Algorithm 2 GPU software rasterization of 3D Gaussians

w, h: width and height of the image to rasterize

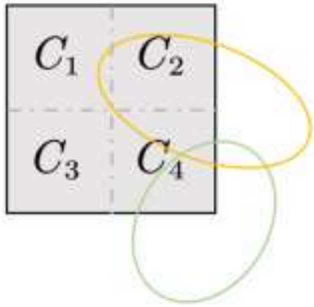
M, S: Gaussian means and covariances in world space

C, A: Gaussian colors and opacities

V: view configuration of current camera

function RASTERIZE(w, h, M, S, C, A, V)
    CullGaussian(p, V) ▷ Frustum Culling
    $M', S' \leftarrow$ ScreenspaceGaussians(M, S, V) ▷ Transform
    $T \leftarrow$ CreateTiles(w, h)
    $L, K \leftarrow$ DuplicateWithKeys(M', T) ▷ Indices and Keys
    SortByKeys(K, L) ▷ Globally Sort
    $R \leftarrow$ IdentifyTileRanges(T, K)
    $I \leftarrow 0$ ▷ Init Canvas
    for all Tiles t **in** I **do**
        for all Pixels i **in** t **do**
            $r \leftarrow$ GetTileRange(R, t)
            $I[i] \leftarrow$ BlendInOrder(i, L, r, K, M', S', C, A)
        end for
    end for
    return I
end function

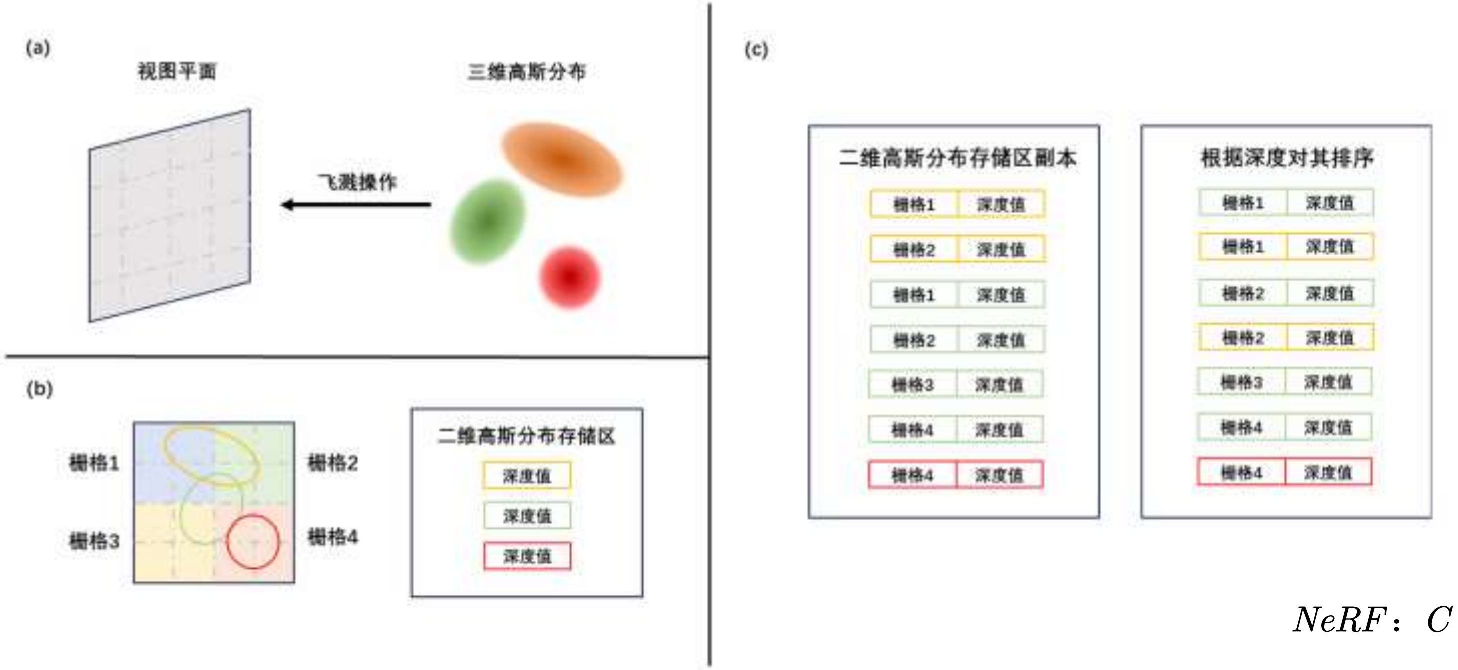
栅格1

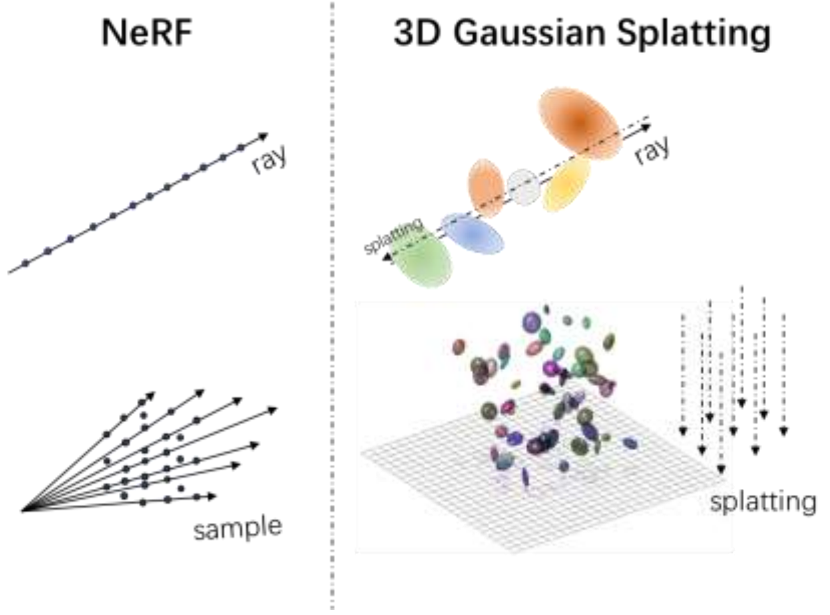


$$C_1 = \alpha_1' c_1 + \alpha_1' c_2 (1 - \alpha_1')$$
$$C_2 = \alpha_2' c_1 + \alpha_2' c_2 (1 - \alpha_2')$$
$$C_3 = \alpha_3' c_1 + \alpha_3' c_2 (1 - \alpha_3')$$
$$C_4 = \alpha_4' c_1 + \alpha_4' c_2 (1 - \alpha_4')$$

三维高斯分布快速渲染的关键技术研究：

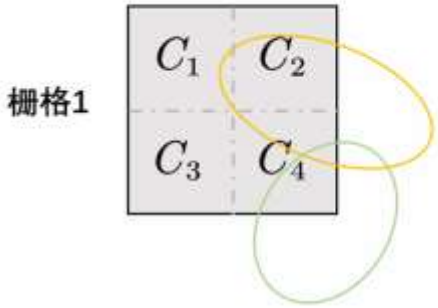
快速可微光栅化器和NeRF渲染方法的区别





$$NeRF\text{：}\ C=\sum_{i=1}^{N} T_i \alpha_i c_i, with\ \alpha=\left(1-e^{-\sigma_i \delta_i}\right)\ and\ T_i=\prod_{j=1}^{i-1}(1-\alpha_j)$$

像素 $\xrightarrow{\text{反向寻找}}$ 体素 $\xrightarrow{\text{渲染}}$ 像素



$$C_1=\alpha_1' c_1+\alpha_1' c_2(1-\alpha_1')$$
$$C_2=\alpha_2' c_1+\alpha_2' c_2(1-\alpha_2')$$
$$C_3=\alpha_3' c_1+\alpha_3' c_2(1-\alpha_3')$$
$$C_4=\alpha_4' c_1+\alpha_4' c_2(1-\alpha_4')$$

$$3D\ Gaussian\ Splatting\text{：}\ C=\sum_{i\in N} c_i \alpha_i \prod_{j=1}^{i-1}(1-\alpha_j)$$

体素 $\xrightarrow{\text{直接影响}}$ 像素

三维高斯分布快速渲染的关键技术研究：

自适应密度控制器的实现

两个关键点：**Pruning**减小伪影出现和**Densification**处理过度重建和欠采样。

Pruning

为了稳定计算过程，先以较低分辨率（4倍下采样）预热计算，在250次和500次迭代时进行两次上采样。

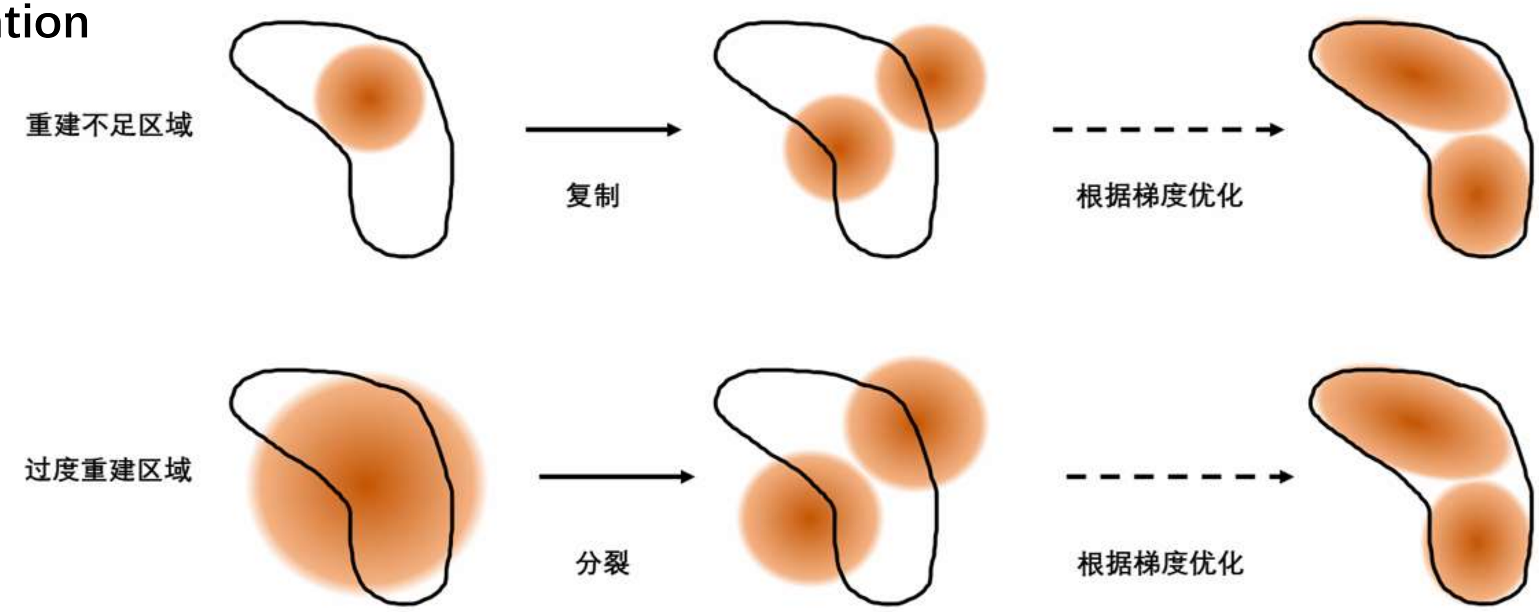
在预热之后，每经过100次迭代就增加密度，并移除基本透明的$Gaussian$，即$\alpha < \varepsilon_\alpha$的$Gaussian$。

每经过1000次迭代移除不透明度小于阈值的点，周期性将不透明度重置为0以去除漂浮物，并移除形状较大点避免重叠。

每经过3000次迭代就将α设置为接近0，然后优化，在需要的$Gaussian$上增加α，并剔除$\alpha < \varepsilon_\alpha$的$Gaussain$。

三维高斯分布快速渲染的关键技术研究：

Densification



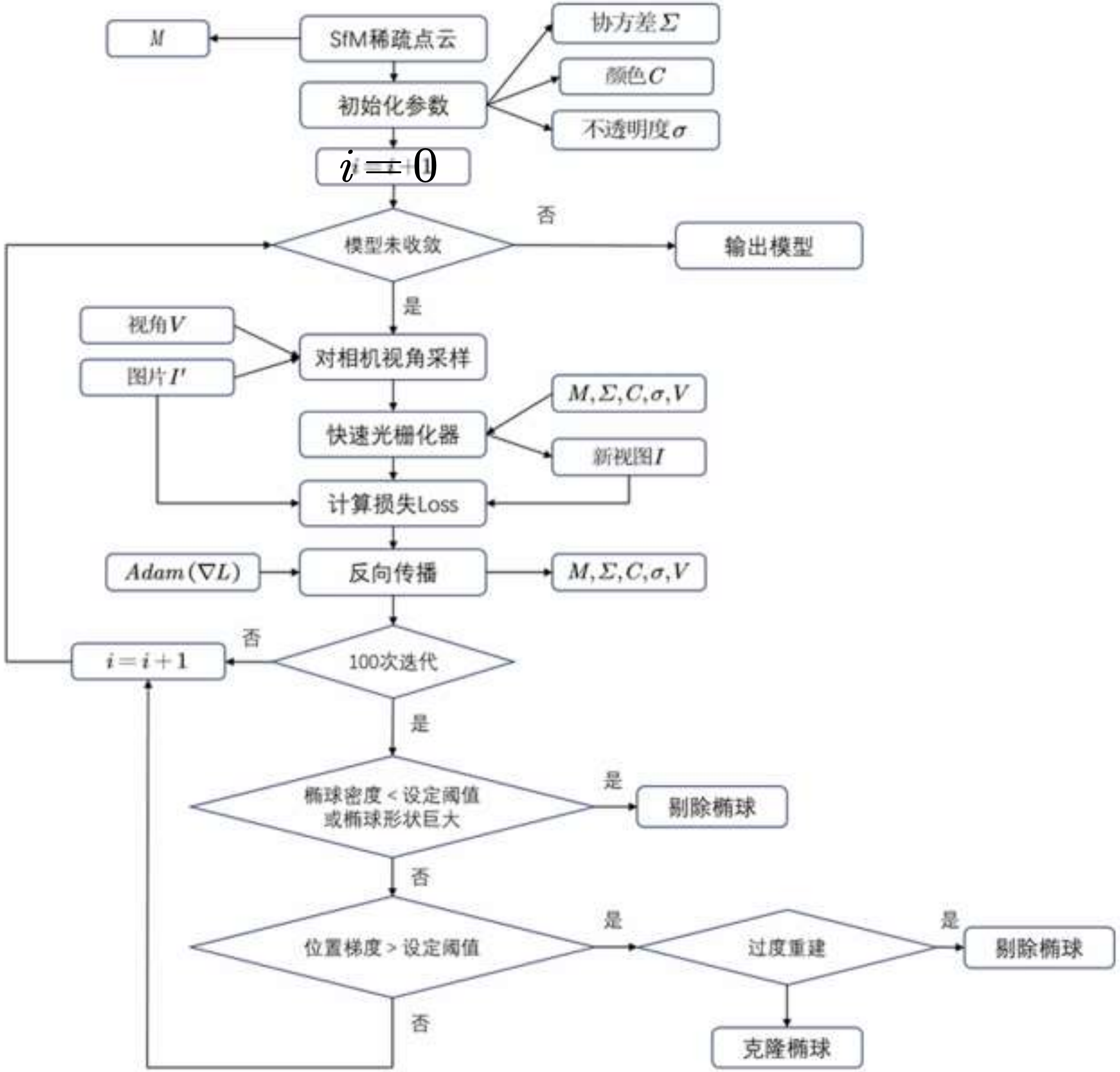
过度重建(*Gaussian*分布覆盖场景大面积区域)和不完整重建(缺失几何特征)往往会有较大的梯度。

$Under-reconstruction$：$clone$，简单创建相同大小的副本，并沿着位置梯度的方向移动它。

$Over-reconstruction$：$split$，将他们除以因子$\phi=1.6$，并分裂成两个较小的*Gaussian*。

这两种情况都需要对*Gaussian*增加密度，对视图空间位置梯度平均幅值超过阈值ι_{pos}的*Gaussian*增加密度。

实验设置：



Algorithm 1 Optimization and Densification

w, h: width and height of the training images

$M \leftarrow$ SfM Points ▷ Positions
$S, C, A \leftarrow$ InitAttributes() ▷ Covariances, Colors, Opacities
$i \leftarrow 0$ ▷ Iteration Count
while not converged **do**
  $V, \hat{I} \leftarrow$ SampleTrainingView() ▷ Camera V and Image
  $I \leftarrow$ Rasterize(M, S, C, A, V) ▷ Alg. 2
  $L \leftarrow Loss(I, \hat{I})$ ▷ Loss
  $M, S, C, A \leftarrow$ Adam(∇L) ▷ Backprop & Step
  if IsRefinementIteration(i) **then**
    for all Gaussians (μ, Σ, c, α) **in** (M, S, C, A) **do**
      if $\alpha < \epsilon$ or IsTooLarge(μ, Σ) **then** ▷ Pruning
        RemoveGaussian()
      end if
      if $\nabla_p L > \tau_p$ **then** ▷ Densification
        if $\|S\| > \tau_S$ **then** ▷ Over-reconstruction
          SplitGaussian(μ, Σ, c, α)
        else ▷ Under-reconstruction
          CloneGaussian(μ, Σ, c, α)
        end if
      end if
    end for
  end if
  $i \leftarrow i + 1$
end while