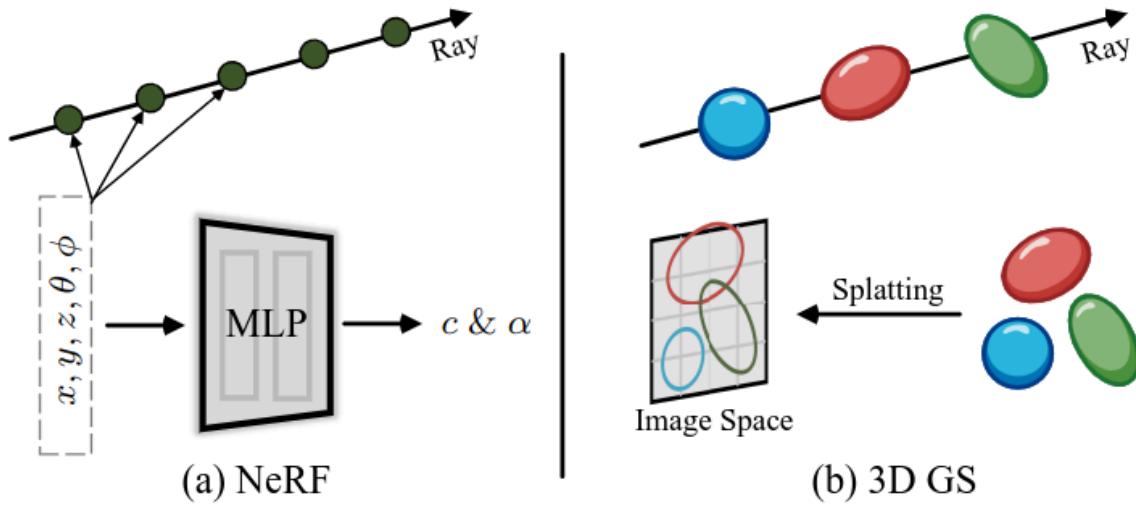


3D Gaussian Splatting for Real-Time Radiance Field Rendering

[Paper Website](#)

[Github](#)



NeRF - ray-tracer

3D Gaussian - Rasterizer

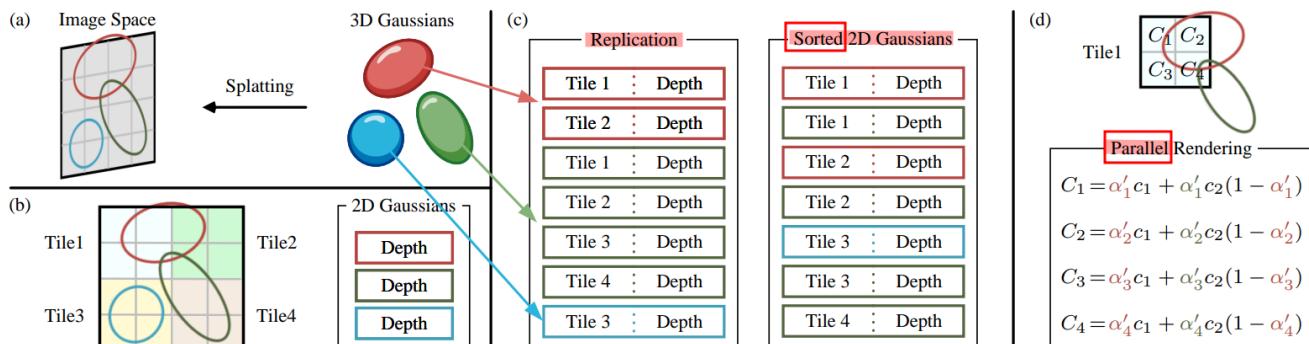


Fig. 4. An illustration of the forward process of 3D GS (see Sec. 3.1). (a) The splatting step projects 3D Gaussians into image space. (b) 3D GS divides the image into multiple non-overlapping patches, i.e., tiles. (c) 3D GS replicates the Gaussians which cover several tiles, assigning each copy an identifier, i.e., a tile ID. (d) By rendering the sorted Gaussians, we can obtain all pixels within the tile. Note that the computational workflows for pixels and tiles are independent and can be done in parallel. Best viewed in color.

[3D Gaussian Splatting: How's This The Future of 3D AI? - YouTube](#)

【论文讲解】用点云结合3D高斯构建辐射场

1. 没有用到 法向量
2. 3D Gaussian - TODO 详细推导

- i. 文章中的 3D Gaussian 表达式 没有 控制概率密度积分=1 的系数
- 3. $\Sigma = AA^T = RSS^T R^T$, 变换矩阵 A 拆分为先缩放再旋转($A = RS$)
- 4. Optimization
 - i. SGD 随机梯度下降
 - ii. 不透明度 激活函数 Sigmoid
 - iii. 协方差矩阵尺度 激活函数 Exponential
 - iv. 损失函数 $Loss = (1 - \lambda)L_1 + \lambda L_{D-SSIM}$ (SSIM 结构相似性损失)
- 5. Adaptive Control of Gaussians 自适应密度控制
 - i. 移除透明度小于阈值的点(源码似乎是用 mask)
 - ii. 重建不充分的区域(反向传播的梯度较大，需要修正量大) - split(gaussian 方差大) & clone(gaussian 方差小)
- 6. Tile-based Rasterizer 快速可微光栅化 - TODO

Table of Contents

- [3D Gaussian Splatting for Real-Time Radiance Field Rendering](#)
 - [Table of Contents](#)
- [3D Gaussian Splatting 源码 - Ubuntu](#)
 - [Overview](#)
 - [Requirements](#)
 - [Installation](#)
 - [Dataset](#)
 - [Visualization](#)
 - [Colab](#)
 - [下载文件](#)
 - [Explanation](#)
 - [Processing your own Scenes](#)
- [Gaussian Splatting 原理速通 - 中恩实验室](#)
 - [01 - 三维高斯 概念](#)
 - [02 - 球谐函数\(Spherical Harmonics\) 概念](#)
 - [03 - 迭代参数 & 渲染](#)
 - [04 - 伪代码流程](#)
- [讲人话 3D Gaussian Splatting 全解](#)
 - [捏雪球](#)
 - [抛雪球 \(3D -> Pixel\)](#)

- 雪球颜色
 - 高性能渲染 & 机器学习
 - Cite Info
 - Ideas from ShenLong Wang
-

3D Gaussian Splatting 源码 - Ubuntu

[Gaussian Splatting - Github](#)

Overview

4 main components:

1. A **PyTorch-based optimizer** to produce a 3D Gaussian model from SfM inputs
2. A **network viewer** that allows to connect to and visualize the optimization process
3. An **OpenGL-based real-time viewer** to render trained models in real-time.
4. A script to help you **turn your own images into optimization-ready SfM data sets**

have been tested on **Windows 10** and **Ubuntu Linux 22.04**

Requirements

[Your GPU Compute Capability - NVIDIA](#)

Optimizer

1. **Hardware Requirements**
 - i. CUDA-ready GPU with Compute Capability 7.0+
 - ii. 24 GB VRAM (to train to paper evaluation quality)
 - iii. I don't have 24 GB of VRAM for training, what do I do?
 - a. The VRAM consumption is **determined by the number of points that are being optimized**, which increases over

- time.
- b. If you only want to train to 7k iterations, you will need significantly less.
 - c. To do the full training routine and avoid running out of memory, you can
 - a. increase `--densify_grad_threshold` : 增大该值会减少高斯点的密集化，减少显存使用
 - b. increase `--densification_interval` : 增大该值可以减少密集化的频率，从而降低显存需求
 - c. reduce `--densify_until_iter` : 减少该参数值，可以缩短密集化的持续时间，降低显存占用
 - d. Note however that this will affect the quality of the result.
 - e. Also try setting `--test_iterations` to -1 to avoid memory spikes during testing. 禁用整个训练过程中的测试阶段
 - f. If `--densify_grad_threshold` is very high, no densification should occur and training should complete if the scene itself loads successfully.

2. Software Requirements

- i. Conda (recommended for easy setup)
- ii. C++ Compiler for PyTorch extensions (we used Visual Studio 2019 for Windows)
- iii. CUDA SDK 11 for PyTorch extensions, install after Visual Studio (we used 11.8, known issues with 11.6)
- iv. C++ Compiler and CUDA SDK must be compatible

Interactive Viewers

1. Hardware Requirements

- i. OpenGL 4.5-ready GPU and drivers (or latest MESA software)
- ii. 4 GB VRAM recommended
- iii. CUDA-ready GPU with Compute Capability 7.0+ (only for Real-Time Viewer)

2. Software Requirements

- i. Visual Studio or g++, not Clang (we used Visual Studio 2019 for Windows)

- ii. CUDA SDK 11, install after Visual Studio (we used 11.8)
- iii. CMake (recent version, we used 3.24)
- iv. 7zip (only on Windows)

```
import torch
if torch.cuda.is_available():
    print(torch.cuda.get_device_properties(0).name)    # NVIDIA GeForce RTX 3050 Laptop GPU
    print(torch.cuda.get_device_properties(0).major)   # 主版本号 : 8
    print(torch.cuda.get_device_properties(0).minor)   # 次版本号 : 6
else:
    print("No CUDA-ready GPU found.")
```

```
nvidia-smi --query-gpu=memory.total --format=csv
# memory.total [MiB]
# 4096 MiB

gcc --version # 检查 C++ 编译器
# gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
# Copyright (C) 2021 Free Software Foundation, Inc.

g++ --version
# g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
# Copyright (C) 2021 Free Software Foundation, Inc.

nvcc -V # 检查 CUDA SDK
# nvcc: NVIDIA (R) Cuda compiler driver
# Copyright (c) 2005-2024 NVIDIA Corporation
# Built on Tue_Feb_27_16:19:38_PST_2024
# Cuda compilation tools, release 12.4, V12.4.99
# Build cuda_12.4.r12.4/compiler.33961263_0

cmake --version
# cmake version 3.25.1
# CMake suite maintained and supported by Kitware (kitware.com/cmake).

glxinfo | grep "OpenGL version"
# OpenGL version string: 4.6 (Compatibility Profile) Mesa 23.2.1-1ubuntu3.1~22.04.2
```

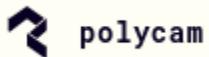
Installation

```
git clone https://github.com/graphdeco-inria/gaussian-splatting --recursive # HTTPS # recursive  
cd gaussian-splatting  
  
sudo apt install build-essential  
sudo apt install ninja-build  
  
  
pip3 install submodules/diff-gaussian-rasterization  
pip3 install submodules/simple-knn  
  
pip3 install plyfile  
pip3 install tensorflow  
  
  
# Interactive Viewers  
sudo apt install -y libglew-dev libassimp-dev libboost-all-dev libgtk-3-dev libopencv-dev libglfw3
```

Dataset

T&T+DB COLMAP (650MB)

Visualization



[PolyCam](#) - Go to library, Upload a 3D model

[3D Gaussian Splatting with Three.js](#) - 清华大学自动化系JI-GROUP

[splatviz](#) - 待测试

Colab

[gaussian_splatting_colab.ipynb](#)

```
%cd /content  
!git clone --recursive https://github.com/camenduru/gaussian-splatting  
!pip install -q plyfile  
  
%cd /content/gaussian-splatting  
!pip install -q /content/gaussian-splatting/submodules/diff-gaussian-rasterization  
!pip install -q /content/gaussian-splatting/submodules/simple-knn  
  
!wget https://huggingface.co/camenduru/gaussian-splatting/resolve/main/tandt_db.zip  
!unzip tandt_db.zip  
  
!python train.py -s /content/gaussian-splatting/tandt/train  
  
# !wget https://huggingface.co/camenduru/gaussian-splatting/resolve/main/GaussianViewTest.zip  
# !unzip GaussianViewTest.zip  
# !python render.py -m /content/gaussian-splatting/GaussianViewTest/model  
# !ffmpeg -framerate 3 -i /content/gaussian-splatting/GaussianViewTest/model/train/ours_30000/render%04d.mp4  
# !ffmpeg -framerate 3 -i /content/gaussian-splatting/GaussianViewTest/model/train/ours_30000/gt/%04d.mp4
```

下载文件

将文件夹下载到你的Google Drive中

```
from google.colab import drive  
drive.mount('/content/drive')  
!cp -r [srcFolder] /content/drive/MyDrive/ # 会将 srcFolder 复制到 Google Drive 中 [srcFolder]
```

或

Explanation

`train.py` 训练代码主入口

`scene/_init_.py` 场景训练数据,相机数据,gaussian模型初始化

`gaussian_model.py` gaussian模型核心代码

`gaussian_renderer/init.py` render渲染核心代码

Processing your own Scenes

COLMAP loaders expect the following dataset structure

```
<location>
|---images # 存储所有用于重建的图像
|   |---<image 0>
|   |---<image 1>
|   |---...
|---sparse # 存储重建生成的稀疏点云数据
|   |---0
|       |---cameras.bin # 相机参数
|       |---images.bin # 图像位姿, COLMAP 默认选择场景中第一个相机的位置作为原点
|       |---points3D.bin # 三维点
```

loader 实际上会从 images 文件夹中读取文件，而不是从 input 文件夹读取

1. `input` 文件夹 主要用于存放未经处理的原始图像
2. `images` 文件夹 最终存放无畸变的、经过处理的图像
3. `sparse` 文件夹 用于存放最终的稀疏三维重建结果
4. `distorted` 表示未经去畸变处理的中间数据

`convert.py`

对于视频 可以先用 ffmpeg 从中提取 图片

```
ffmpeg -i {video} -qscale:v 1 -qmin 1 -vf fps={fps} %4d.jpg
ffmpeg -i {video} -qscale:v 1 -qmin 1 -vf "fps={fps},scale=iw/2:ih/2" %4d.jpg # 添加 scale 滤镜来将图片缩小一半

# -qscale:v 1 -qmin 1 : 控制输出图像的质量, -qmin 1 : 设置最小质量
# -vf fps={fps} : 设置输出的帧率(vf = video filter), 将 {fps} 替换为所需的帧率, 比如 fps=30 表示每秒提供 30 帧
# %4d.jpg : 指定输出文件的命名格式, 从 0001.jpg 开始自动编号
```

COLMAP 的相机模型必须是 SIMPLE_PINHOLE 或 PINHOLE，这样才可以用于后续的光栅化渲染

`convert.py`

1. 对输入图像进行去畸变处理

2. 提取必要的结构化数据 `.bin`
3. 如果需要，也可以将图像按 1/2、1/4 和 1/8 的原始分辨率进行缩放(需要安装 ImageMagick)

```
python convert.py -s <location> [--resize]
```

准备数据：将所有需要处理的图像放入 `<location>/input`

```
python convert.py -s <location> [--resize]  
# -s <location>: 指定数据位置  
# --resize: 可选参数，是否对图像进行缩放
```

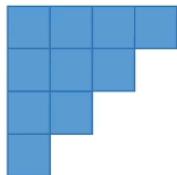
Gaussian Splatting 原理速通 - 中恩实验室

[中恩实验室 - B站主页](#)

[3D Gaussian Splatting 原理速通 - 中恩实验室](#)

01 - 三维高斯概念

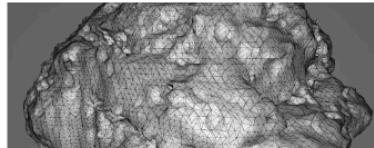
方块组成的世界：



三角面组成的世界：



椭圆组成的世界：

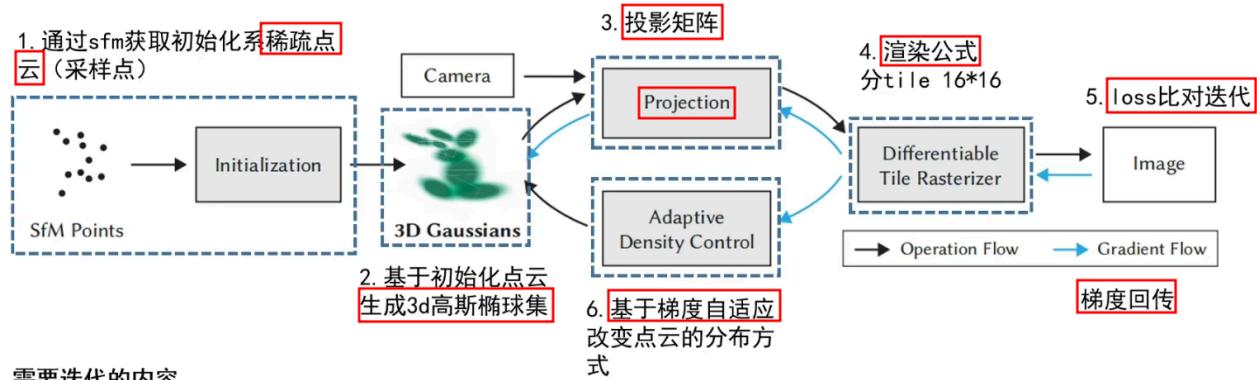


3D GS 使用 椭圆 拼接

效果

- 允许在 **1080p** 分辨率下实现高质量的实时 (**>30fps**) 的新视图合成
- 在提出该方法时，在现有公开数据集，确保实时渲染，达到 sota 质量
- mipnerf360 在渲染相同质量下，需要长达48h的训练时间，无法在高分辨率下实现实时渲染，10-15帧每秒

整体框架



需要迭代的内容

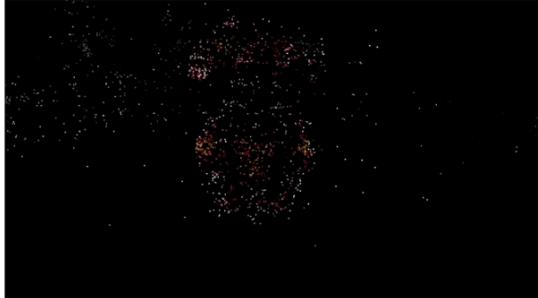
点云位置信息优化 (x, y, z) ，即高斯椭球的中心点

点云颜色 (r, g, b) —使用球谐函数来表示，使得点云在不同角度呈现不同颜色，并有利于提高迭代效率（代码中采用4阶）

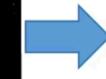
点云不透明度优化 α

高斯椭球的协方差矩阵 Σ — 包含高斯椭球的旋转矩阵R和在各个轴缩放矩阵S

其中 $\Sigma = RSS^T R^T$ (保证半正定的性质，几何意义是先将椭球旋转到与椭球世界平齐，然后沿着轴缩放，再旋转还原回去。这样处理的好处是：由于 Σ 不能进行随机初始化，能保证 Σ 仍保持半正定的性质，并且在一定程度也减少运算量)



初始化点云



高斯椭球们的中心点



带上不透明度，渲染画面



高斯椭球们的协方差



初始化：通过 **colmap** 推理照片对应的相机位姿，得到 特征点，投射到世界坐标，得到稀疏关键点云(比随机初始化更快收敛)，作为初始场景默认点

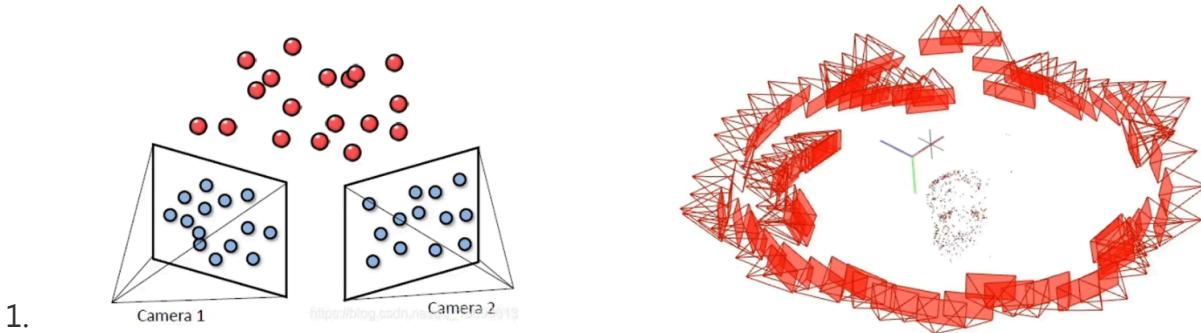
1、sfm初始化稀疏点云

优化从稀疏SfM点云开始，创建一组三维高斯分布。

这里通过 colmap 先创建出一个 初始化点云

通过一组照片就能进行估算

保持连续体积辐射场的理想特性，用于场景优化，同时避免了空白空间中不必要的计算



Gaussian 椭球集创建

2、3D高斯椭球集的创建——位置与形状

位置信息：点云位置信息优化 (x, y, z) ，即高斯椭球的中心点（即均值）

形状信息：高斯椭球的协方差矩阵 Σ — 包含高斯椭球的旋转矩阵R和在各个轴缩放矩阵S

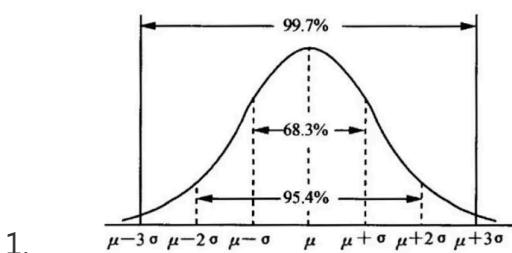
其中 $\Sigma = RSS^T R^T$ （保证半正定的性质，几何意义是先将椭球旋转到与椭球世界平齐，然后沿着轴缩放，再旋转还原回去。这样处理的好处是：由于 Σ 不能进行随机初始化，能保证 Σ 仍保持半正定的性质，并且在一定程度也减少运算量）

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

$$x \sim \mathcal{N}(\mu, \Sigma) = P(x | \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} \cdot |\Sigma|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right\}$$

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

$$\Sigma = RSS^T R^T$$



1. 协方差矩阵 Σ 决定形状，联想 PCA

(x, y, z)

02 - 球谐函数(Spherical Harmonics) 概念

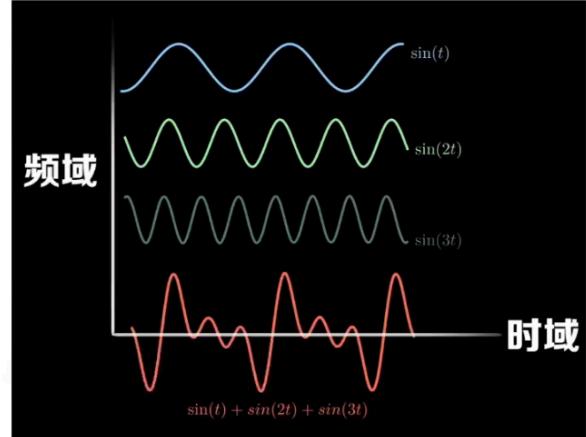
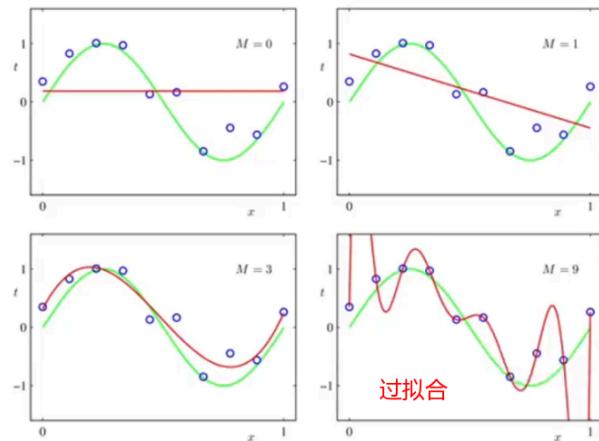
类似于傅里叶级数，也是由一堆 基函数 组成(用于球面)

2、3D高斯椭球集的创建——傅里叶级数展开与基函数

球谐函数可以类似看作是一种多项式的拟合（其实就是泰勒展开式子）

譬如 $f(x) = ax^3 + bx^2 + cx^1 + dx^0$ 是三次多项式拟合，共有4个参数，可以把 x^3, x^2, x^1, x^0 视作为构建三次多项式函数的基函数

也类似傅里叶级数的展开，分别用 \sin 和 \cos 作为基函数的组合可以近似拟合不同的周期函数



$$r = f(\theta, \phi)$$

左图中，每一行代表阶数(从0开始)

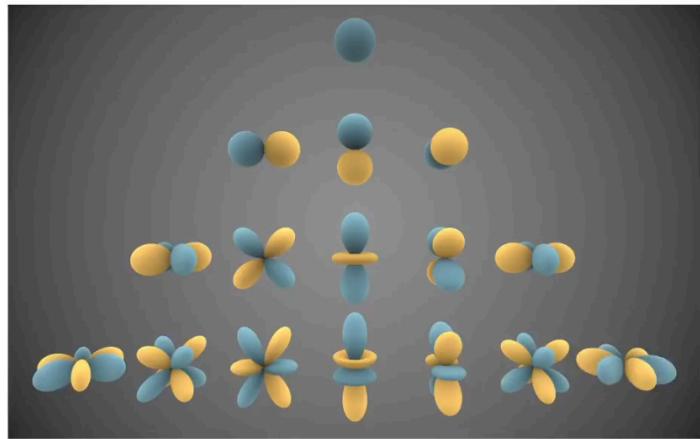
左下角图 展示 球谐函数 基函数 (阶数越高，表达能力越强)，通过 线性组合

球谐函数 只和 仰角 & 方位角 有关，与 半径 r 无关

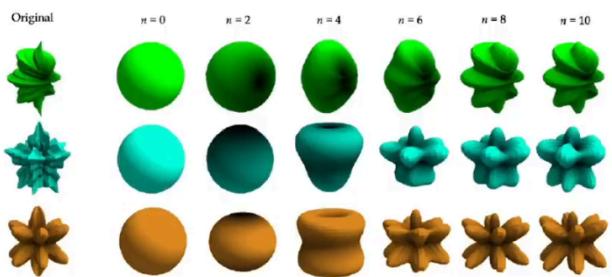
2、3D高斯椭球集的创建——球谐函数

球谐函数其实就是，一组能代表球面上不同位置的值基函数，阶数越高表达能力越强
球谐函数只和 θ, ϕ 有关，与 r 无关。参数量为阶数的平方

让整个球面的数据得到更为连续地表达（也可以理解成更加光滑，阶数越高能表达地形状更复杂）



如何看懂此图：此图像可视化是指不同基函数在传入 θ, ϕ 得到的 r 最后为出来的形状。将这不同阶数的基函数进行线性组合就能逼近不同的球体啦~



颜色表达类似， r,g,b 对应之前的半径 r

r,g,b 分别用三个不同的球谐函数储存颜色值

代码中使用 4 阶球谐函数（共有 0,1,2,3 阶数）

第 l 层的球数 = $2l + 1$

个人理解：每个球只存储一个权重值，从倒三角可以看出，球的累计数量(n 阶)为 n^2

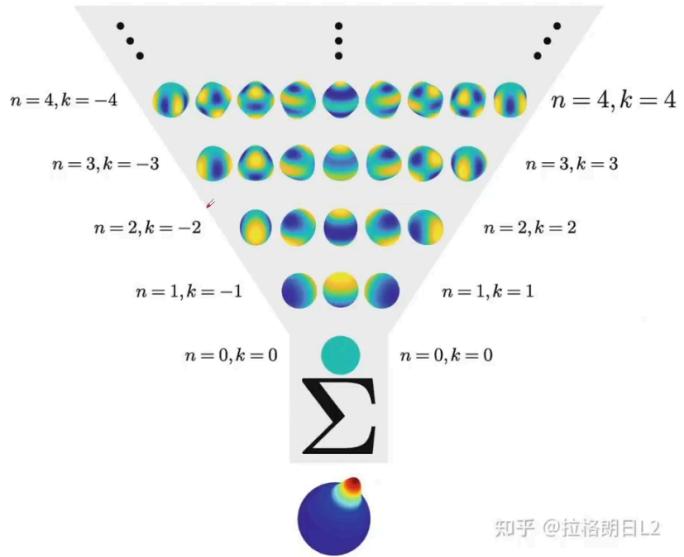
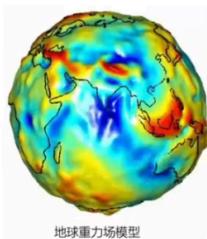
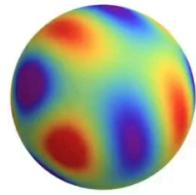
因为有 r,g,b 三个值，故使用 $3 \times 4^2 = 48$ 个参数，存储 RGB 值

实现效果：从不同角度看高斯点云能有不同颜色

球谐函数用在不同角度的颜色表达

在3d-gaussian-splatting中点云颜色 (r, g, b) --
使用球谐函数来表示，使得点云在不同角度呈现
不同颜色，并有利于提高迭代效率（代码中采用4
阶，参数数量为 n^2 ）
 $3 \times 16 = 3 + 45 = 48$

球谐函数在一定程度上能弱化高频信息，本质上
是一种有损压缩。并且能将离散的信息变为连续
信息，可计算梯度并进行迭代



03 - 迭代参数 & 渲染

PLY 文件 Polygon File Format

1. 用于存储三维模型的 点云数据 或 多边形网格
2. 存储 每个顶点 的 位置、颜色、法向量以及其他与几何模型相关的属性
 - i. ASCII 格式：以人类可读的形式存储，容易查看和修改，但文件较大，加载速度较慢
 - ii. 二进制格式：以二进制形式存储，文件更小，读取速度更快，但不易直接查看和编辑
3. 包含两个部分
 - i. 头部(Header)：描述文件内容的元数据(包含文件的格式，顶点、面的数量，以及对应属性)
 - a. 以 ply 开始，以 end_header 结束
 - ii. 数据部分(Data Section)：存储实际的点或面的数据

PLY 文件说明

1. element vertex 代表点数
2. property 代表每个点包含的信息
 - i. x,y,z : Gaussian 中心点

- ii. nx, ny, nz : 法向量信息
- iii. f_dc, f_rest : 颜色信息，共 $3 + 45 = 48$ 个参数(r,g,b 各 16)
- iv. opacity : 不透明度，密度信息
- v. scale : 缩放矩阵
- vi. rot : 旋转矩阵(旋转四元数)

2、3D高斯椭球集的创建——各迭代参数的含义

位置信息: (x, y, z)

法向量信息: (nx, ny, nz) ——应该没用到

颜色信息: 代表每个高斯分布的颜色RGB的球谐(SH)系数(f_dc_0, f_dc_1, f_dc_2)和(f_rest_0 到 f_rest_44)

不透明度/密度信息: opacity α

缩放矩阵: (scale_0, scale_1, scale_2), 三个方向的缩放轴, 即协方差矩阵的参数

旋转矩阵: (rot_0 到 rot_3), 使用旋转四元数表示

```

1  ply
2  format binary_little_endian 1.0
3  element vertex 623102 点数
4  property float x
5  property float y
6  property float z
7  property float nx
8  property float ny
9  property float nz
10 property float f_dc_0
11 property float f_dc_1
12 property float f_dc_2
13 property float f_rest_0
14 property float f_rest_1
15 property float f_rest_2
16 property float f_rest_3
17 property float f_rest_4
18 property float f_rest_5
19 property float f_rest_6
20 property float f_rest_7
21 property float f_rest_8
22 property float f_rest_9
23 property float f_rest_10
24 property float f_rest_11
25 property float f_rest_12
26 property float f_rest_13
27 property float f_rest_14
28 property float f_rest_15
29 property float f_rest_16
30 property float f_rest_17
31 property float f_rest_18
32 property float f_rest_19
33 property float f_rest_20
34 property float f_rest_21
35 property float f_rest_22
36 property float f_rest_23
37 property float f_rest_24
38 property float f_rest_25
39 property float f_rest_26
40 property float f_rest_27
41 property float f_rest_28
42 property float f_rest_29
43 property float f_rest_30
44 property float f_rest_31
45 property float f_rest_32
46 property float f_rest_33
47 property float f_rest_34
48 property float f_rest_35
49 property float f_rest_36
50 property float f_rest_37
51 property float f_rest_38
52 property float f_rest_39
53 property float f_rest_40
54 property float f_rest_41
55 property float f_rest_42
56 property float f_rest_43
57 property float f_rest_44
58 property float opacity
59 property float scale_0
60 property float scale_1
61 property float scale_2
62 property float rot_0
63 property float rot_1
64 property float rot_2
65 property float rot_3
66 end_header

```

包含
信息

投影矩阵 - 非线性变换

将原始形状进行挤压, 到光栅化平面空间, 将非平行线拉平, 对点进行排序 并 进行光栅化渲染(并行计算加速)

使用雅可比矩阵, 利用 泰勒公式(扩展到三维)一阶展开 进行线性近似

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

3、计算机图形学投影矩阵

3D Gaussian 协方差（缩放系数）变换

是非线性的变换

于是用泰勒一阶展开公式来进行线性的近似

$$f(x) \approx f(a) + f'(a)(x - a)$$

$$f(x) \approx f(a) + J(x - a)$$

泰勒公式扩展到三维空间，**需要用到雅可比矩阵**

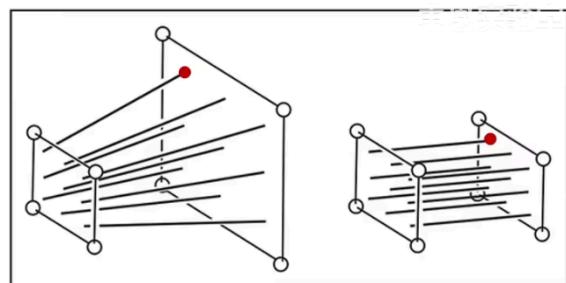
于是有这个公式：

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

其中雅可比矩阵代表(直接计算得到，不需要迭代)：

表示一个多元函数在某一点的局部线性近似

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$



对世界坐标系下的高斯椭球进行挤压到光栅化平面空间

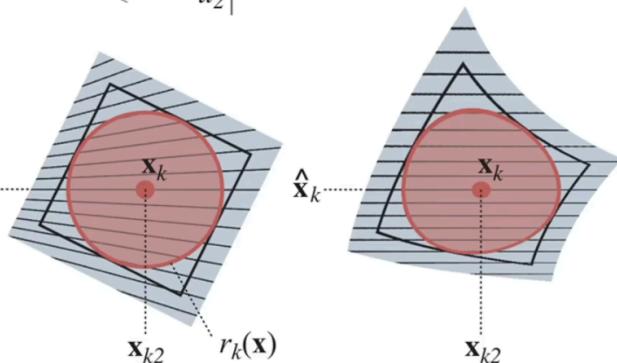
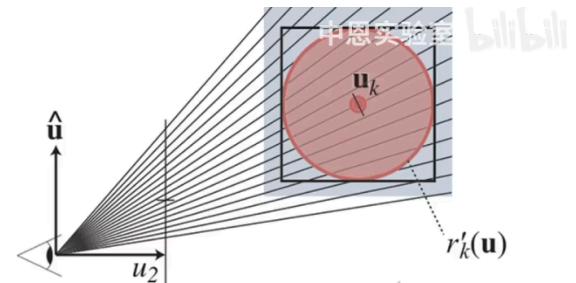
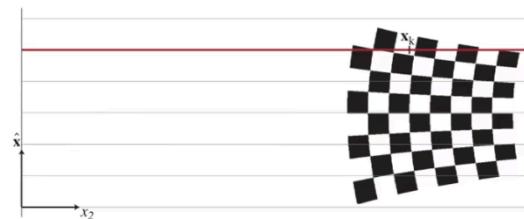
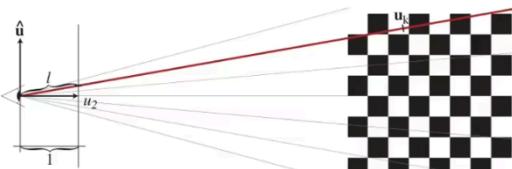


Figure 4: Mapping a reconstruction kernel from camera to ray space. Top: camera space. Bottom: ray space. Left: local affine mapping. Right: exact mapping.

渲染公式，与 NeRF 一样，都是 权重 × 不透明度 × 颜色

1. NeRF 的 点基于射线(视线)，权重是指透光率，不透明度是指体密度
2. 3D GS 的

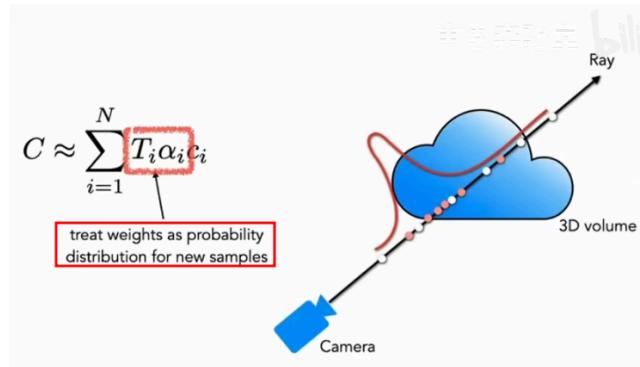
通过点云中一定半径范围内能够影响像素的 N 个有序点 计算像素的颜色 C

4、渲染公式

Nerf体渲染公式：

$$C = \sum_{i=1}^N T_i (1 - \exp(-\sigma_i \delta_i)) \mathbf{c}_i \quad \text{with} \quad T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right),$$

$$C = \sum_{i=1}^N T_i \alpha_i \mathbf{c}_i, \quad \text{with} \quad \alpha_i = (1 - \exp(-\sigma_i \delta_i)) \quad \text{and} \quad T_i = \prod_{j=1}^{i-1} (1 - \alpha_j).$$



3dgs也是通过类似point-based的渲染公式：

即通过点云中一定半径范围能影响像素的N个有序点来计算一个像素的颜色C

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

α_i 代表当前点*i*的不透明度（密度）值
 α_j 代表*j*之前前面点的不透明度（密度）值

用 $1 - \alpha_j$, 进行累乘, 作为权重weight
这代表着前面的点越透明, 这个weight就越大, 当前 α_i 的影响权重越

它的渲染公式其实跟nerf差不多

其中：影响范围用标准的球来简化计算，并分tile 16*16，一种快速的排序，cuda并行加速，光栅化
损失函数 Loss

1. $\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{D-SSIM}$
2. 论文使用值 $\lambda = 0.2$

给予梯度自适应改变点云分布，densification(密集化)

6、基于梯度自适应改变点云的分布方式

比如每隔100个epoch会判断点云的分布是否合理

Pruning减少伪影的出现：

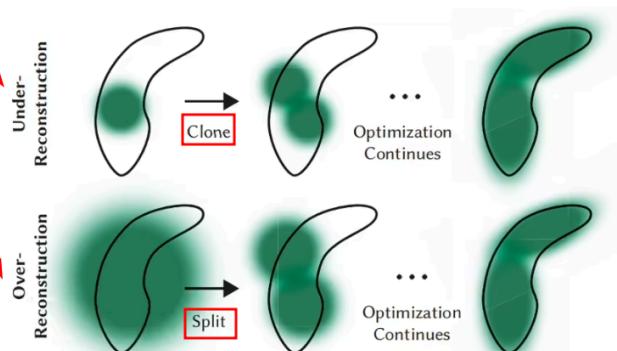
其中不透明度低于设置的阈值或者离相机近的一些点会进行删除

Densification过度重构或者欠采样（基于梯度变化来判断）：

方差很小-->克隆高斯来适应 Under-Reconstruction

方差很大-->分割成两个高斯 Over-Reconstruction

“自适应密度控制步骤交错，我们在优化过程中添加和偶尔删除三维高斯分布。优化过程产生了一个合理的紧凑、非结构化和精确的场景表示（所有测试场景的100-500万高斯值）”



初始化点云质量不高

在优化过程中，每个高斯点都有其颜色、位置和透明度的梯度变化

梯度是针对投影后的 2d gaussian

当某个高斯点的梯度大于预设的 梯度阈值 (**densify_grad_threshold**) 时，会触发 Densification

产生新的 Gaussian 点是三维的，继承原高斯点的三维坐标以及协方差矩阵的初始值

Densification 也可以按照固定的 间隔次数(**Densification Interval**) 触发，以确保在训练过程中不断增加高斯点的数量，从而逐渐提高场景的密集度。只有在到达 **Densification Interval** 的那一批次时，才会检查哪些点需要进行 split 或 clone，减少计算开销，控制密集化的频率和效果

`gaussian_model.py` 中 `densify_and_split()` (使用了随机偏移量，并且scaling也会调整),
`densify_and_clone()`

Densification 操作本身是用于动态调整高斯点的数量和分布，包括增加高斯点、更新其属性和剪枝(pruning)，这些操作不涉及梯度计算，减少显存消耗，优化训练效率

每次 Densification 后，需要重新建立 Loss，相当于自动将新旧点都连接到Loss中

04 - 伪代码流程

Algorithm 1 Optimization and Densification

w, h : width and height of the training images

```
M ← SfM Points                                ▷ Positions 初始化点云
scale S, C, A ← InitAttributes()      ▷ Covariances, Colors, Opacities
i ← 0                                         ▷ Iteration Count
while not converged do
    V,  $\hat{I}$  ← SampleTrainingView()          ▷ Camera V and Image 取数据集样本
    I ← Rasterize(M, S, C, A, V)        ▷ Alg. 2 光栅化&推理
    L ← Loss(I,  $\hat{I}$ )                  ▷ Loss
    优化参数 M, S, C, A ← Adam( $\nabla L$ )  ▷ Backprop & Step
    if IsRefinementIteration(i) then 超参, 训多少步进行优化
        for all Gaussians ( $\mu, \Sigma, c, \alpha$ ) in (M, S, C, A) do
            if  $\alpha < \epsilon$  or IsTooLarge( $\mu, \Sigma$ ) then           ▷ Pruning
                RemoveGaussian()                      透明度过小
            end if
            if  $\nabla_p L > \tau_p$  then 如果梯度大于阈值   ▷ Densification 自适应改变
                if  $\|S\| > \tau_S$  then                   ▷ Over-reconstruction 点云的分布
                    SplitGaussian( $\mu, \Sigma, c, \alpha$ )
                else                               ▷ Under-reconstruction
                    CloneGaussian( $\mu, \Sigma, c, \alpha$ )
                end if
            end if
        end for
    end if
    i ← i + 1
end while
```

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
```

讲人话 3D Gaussian Splatting 全解

讲人话 3D Gaussian Splatting 全解

3D GS Render Python(Python 替代 CUDA) - Github

基于 Splatting 和 机器学习 的 3D Reconstruction 方法 (无深度学习)

捏雪球

什么是 **splatting?**

1. 一种体渲染方法，从 3D 物体 渲染到 2D 平面
2. 形象理解
 - i. 输入是一些雪球，图片是砖墙
 - ii. 图像生成过程 是 向墙面扔雪球的过程
 - iii. 每扔一个雪球，墙面上会有扩散痕迹，足迹(footprint)
3. 主动 & 被动
 - i. Splatting : 主动算法，计算出 每个发光粒子如何影响(多个)像素点
 - ii. NeRF : 被动算法(Ray-Casting)，计算出 每个像素点受到发光粒子的影响来生成图像
4. 核心
 - i. 选择 雪球形状
 - ii. 投掷 雪球，3D to 2D 得到 footprint
 - iii. 合成 footprint

什么是 **3D Gaussian?**

1. 为什么使用 核?
 - i. 输入是 点云中的点 没有体积，需要给点一个核，进行膨胀
 - ii. 核 可以是 高斯 / 圆 / 正方形
2. 为什么选择 Gaussian? 因为有很好的数学性质:
 - i. 仿射变换后，高斯核仍然闭合(封闭性)
 - ii. 3D 降维到 2D (沿着某一个轴积分) 后，依然为高斯
3. 3D Gaussian 定义
 - i.
$$G(x) = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$
 - ii.
$$G(x, y, z) = \frac{1}{(2\pi)^{3/2} |\Sigma|^{1/2}} \exp \left(-\frac{1}{2} \begin{bmatrix} x - \mu_x \\ y - \mu_y \\ z - \mu_z \end{bmatrix}^T \Sigma^{-1} \begin{bmatrix} x - \mu_x \\ y - \mu_y \\ z - \mu_z \end{bmatrix} \right)$$
 - iii. Σ 表示 协方差矩阵(正定)，控制高斯核在不同方向上的扩散程度(形状和方向)

iv. $|\Sigma|$ 是该协方差矩阵行列式

4. P.S.: 低维(1,2 维) Gaussian 函数

$$\text{i. } f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

$$\text{ii. } f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{(x - \mu_x)^2}{2\sigma_x^2} - \frac{(y - \mu_y)^2}{2\sigma_y^2}\right)$$

$$\text{iii. } f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)} \left[\frac{(x - \mu_x)^2}{\sigma_x^2} + \frac{(y - \mu_y)^2}{\sigma_y^2} - 2\rho \frac{(x - \mu_x)(y - \mu_y)}{\sigma_x\sigma_y} \right]\right)$$

为什么 **3D Gaussian** 是 椭球?

1. 对于相同 协方差矩阵， $\exp()$ 前面的部分为 常数， $\exp()$ 指数为常数 对应了 整个 **3D Gaussian** 部分为常数(其余部分本身就是常数)
2. 令 $\exp()$ 指数为常数时，即 指数 = const 时，该表达式 为 椭球面
3. 概率密度函数的等值面(具有相同概率密度的 3D 点所组成的表面) 形成了 椭球面，也就对应 $3D \text{ Gaussian} = \text{const}$ 的形状为 椭球面
4. 而 **3D Gaussian** 可以取 **[0, 1]** 因此 全部可能可以看做 椭球(实心)

协方差矩阵 怎么就能 控制椭球形状呢? 怎么就能 用 旋转 和 缩放 矩阵表达?

高斯分布:

- $\mathbf{x} \sim N(\mu, \Sigma)$
- 均值 $[\mu_1, \mu_2, \mu_3]$
- 协方差矩阵 $\begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z^2 \end{bmatrix}$

高斯分布的仿射变换:

- $\mathbf{w} = A\mathbf{x} + b$
- 1. • $\mathbf{w} \sim N(A\mu + b, A\Sigma A^T)$
- 2. 任意 Gaussian 可以看做标准 Gaussian(均值=0, 协方差= I) 通过 变换 得到的
- 3. 重点在于协方差矩阵， Σ ，可以表示为 旋转 + 拉伸 + 旋转 动作
 - i. SVD 分解:
 - ii. EVD 分解: $\Sigma = U\Lambda U^T$ ，取 $A = U\Lambda^{1/2}$ ，则可得 $\Sigma = AA^T$

标准高斯分布:

- $\mathbf{x} \sim N(\vec{0}, I)$
- 均值 $[0, 0, 0]$
- 协方差矩阵 $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

$$\Sigma = A \cdot I \cdot A^T$$

任意高斯可以看作是标准高斯通过仿射变换得到

- a. Λ 为对角阵
 - b. U 是由特征向量构成的正交阵(旋转矩阵)，P.S. 实对称矩阵
特征向量一定正交
4. 如果所有维度上的方差都相同且没有相关性，那么高斯分布的等值面是一个球体，因为在每个方向上的扩展是对称的(对应 各向同性 **Isotropic**)
5. 协方差矩阵 Σ 的特征值和特征向量决定了高斯分布的主轴方向和扩展范围
 - i. EigenValue 决定了高斯分布在各个主轴方向上的扩展程度
 - ii. EigenVector 定义了椭球的主轴方向

Cholesky 分解， `scipy.linalg.cholesky` - TODO

各向异性 和 各向同性 是什么意思?

1. 各向同性(Isotropic)高斯核：协方差矩阵 Σ 是 标量矩阵 $\sigma^2 I$ ，各个方向方差相同，各个方向不相关，在每个方向上的扩散程度相同

$$\text{i. } G(x, y, z) = \frac{1}{(2\pi\sigma^2)^{3/2}} \exp\left(-\frac{(x - \mu_x)^2 + (y - \mu_y)^2 + (z - \mu_z)^2}{2\sigma^2}\right)$$

2. 各向异性(Anisotropic)高斯核：此时协方差矩阵 Σ 是一个一般的 3×3 对称正定矩阵，不同方向上具有不同的扩散程度

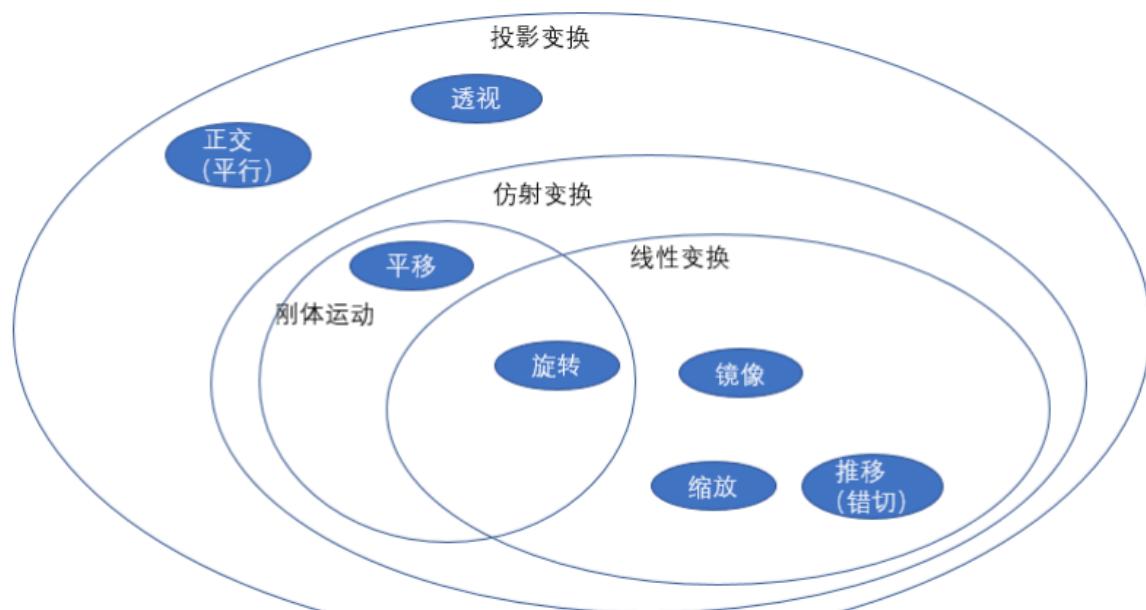
$$\text{i. } \Sigma = \begin{bmatrix} \sigma_x^2 & \rho_{xy} & \rho_{xz} \\ \rho_{xy} & \sigma_y^2 & \rho_{yz} \\ \rho_{xz} & \rho_{yz} & \sigma_z^2 \end{bmatrix}$$

ii. ρ 表示不同维度之间的协方差，表示各方向之间的相关性

抛雪球 (3D -> Pixel)

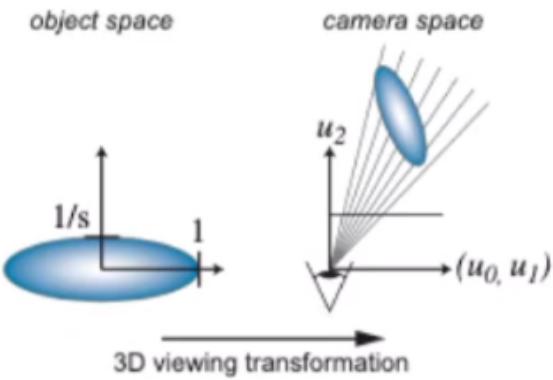
相机模型：世界坐标系(WCS) -> 相机坐标系(CCS) -> 归一化坐标系(NCS) -> 像素坐标系(PCS)

$$\text{i. } \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{PCS} \leftrightarrow \text{NCS}} \underbrace{\begin{bmatrix} \frac{1}{z_c} & 0 & 0 \\ 0 & \frac{1}{z_c} & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{NCS} \leftrightarrow \text{CCS}} \underbrace{\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}}_{\text{CCS} \leftrightarrow \text{WCS}} \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix}$$



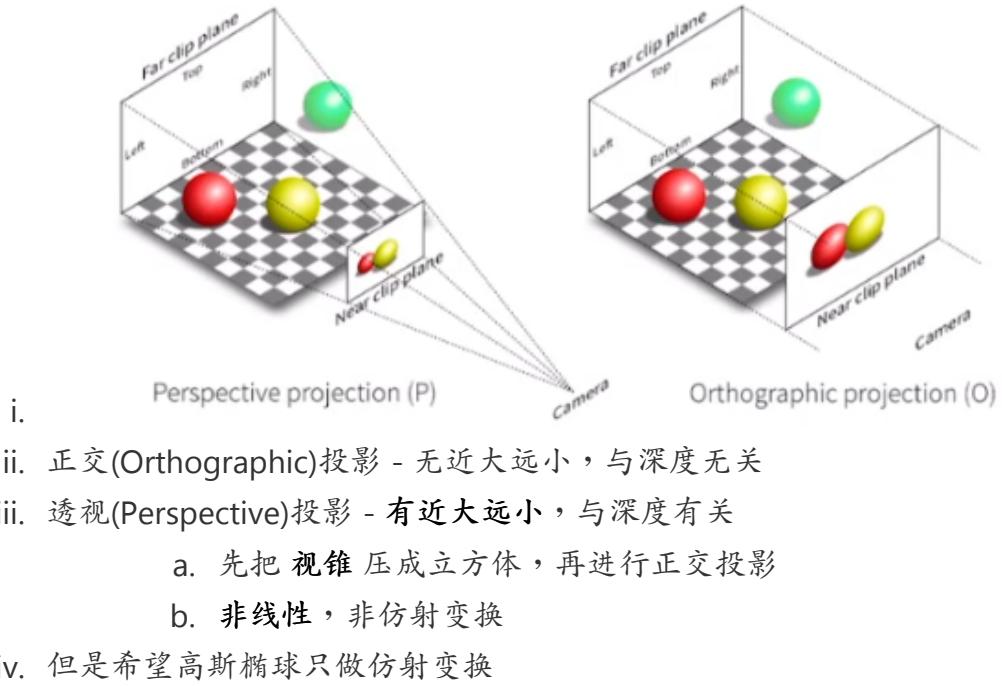
Computer Graphics 中

1. 详见 [GAMES101 Viewing Transformation 个人笔记](#)
2. 视图变换(View Transformation) (世界坐标系 -> 相机坐标系)



- i.
- ii. 本质是 平移加旋转
- iii. 已知 $T_{camera}^{world}, T_{object}^{world}$
- iv. 将物体变换到相机坐标系

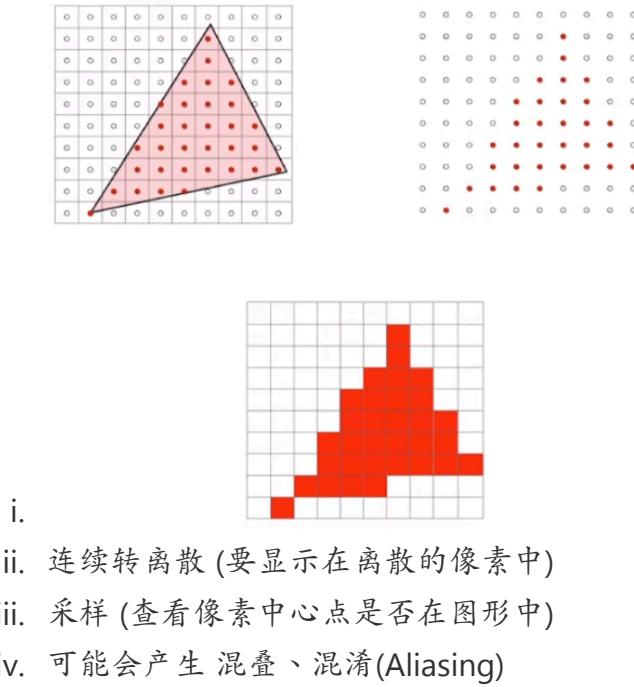
3. 投影变换(Projection Transformation) (相机坐标系 -> 2D 空间)



4. 视口变换

- i. 拉伸成想要的图片大小
- ii. 解决压成 canonical cube 的形变

5. 光栅化 (Rasterization) - 画在屏幕上



3D Gaussian 的 Viewing Transformation (均值 M & 协方差 Σ)

1. 视图变换

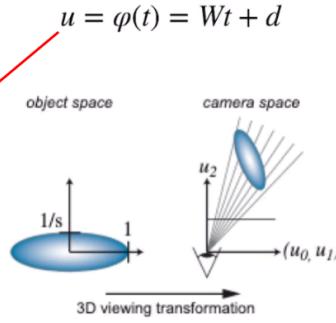
$$i. \quad u = \phi(t) = Wt + d$$

物理坐标系：

- 高斯核中心 $t_k = [t_0 \ t_1 \ t_2]^T$
- 高斯核: $r_k''(t) = G_{V_k}(t - t_k)$
- V_k'' 是协方差矩阵

相机坐标系：

- 高斯核中心 $u_k = [u_0 \ u_1 \ u_2]^T$
- 高斯核: $r_k'(u) = G_{V_k}(u - u_k)$
- 均值 $u_k = Wt_k + d$
- 协方差矩阵 $V_k' = WV_k''W^T$



ii.

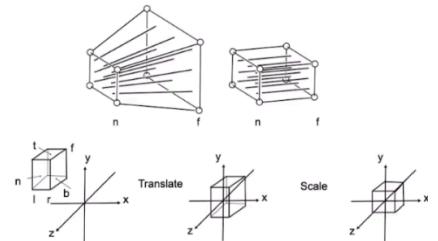
2. 投影变换

相机坐标系：

- 高斯核中心 $u_k = [u_0 \ u_1 \ u_2]^T$
- V_k' 是协方差矩阵

投影变换: !

- 高斯核中心 $x_k = [x_0 \ x_1 \ x_2]^T$
 - 高斯核: $r_k(x) = G_{V_k}(x - x_k)$
 - 均值 $x_k = m(u_k)$
 - 协方差矩阵 ?
- i.



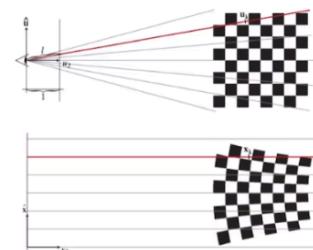
投影变换矩阵:

- $x = m(t)$
- 非线性

- ii. 均值就是一个点，不会形变
- iii. 协方差矩阵会形变，所以难求

可以直接使用投影变换吗?

- 不能
- 从透视投影到正交投影
- 非线性变换，非仿射

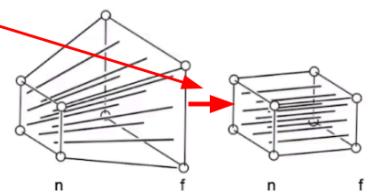


怎么办?

- 引入雅可比近似矩阵

iv.

- v. 引入 Jacobian Matrix(对非线性变换的局部线性近似(利用导数求变换趋势))

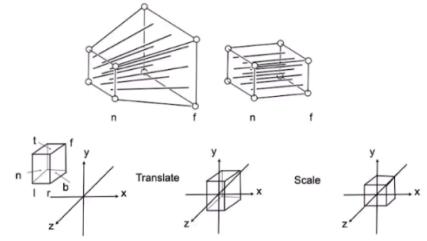


相机坐标系:

- 高斯核中心 $u_k = [u_0 \ u_1 \ u_2]^T$
- V'_k 是协方差矩阵

投影变换: 

- 高斯核中心 $x_k = [x_0 \ x_1 \ x_2]^T$
- 高斯核: $r_k(x) = G_{V'_k}(x - x_k)$
- 均值 $x_k = m(u_k)$ 非线性
- 协方差矩阵 $V_k = JV'_k J^T$
- vi. • $|V_k = JV'_k J^T = JWV'_k W^T J^T|$



雅可比矩阵:

$$\bullet \quad J = \frac{\partial m(u_k)}{\partial u}$$

- vii. 均值已经被投影到了 canonical cube, 需要 视口变换
 viii. 协方差在未缩放的正交坐标系中 $([l, r] \times [b, t] \times [f, n])$, 不需要 视口变换

已知:

$$\bullet \quad M_{persp \rightarrow ortho} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

• 视锥中的一个点 $[x \ y \ z \ 1]^T$

投影变换:

$$\bullet \quad \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - nf \\ z \end{bmatrix} = \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ \frac{(n+f)z - nf}{z} \\ 1 \end{bmatrix}$$

ix.

- x. 表达的三维坐标的点为 $[f_1(x), f_2(x), f_3(x)]^T = [\frac{nx}{z}, \frac{ny}{z}, n + f - \frac{nf}{z}]^T$, 求导即可得 Jacobian

投影变换:

$$\bullet \quad \begin{bmatrix} f_1(x) \\ f_2(y) \\ f_3(z) \end{bmatrix} = \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ (n+f) - \frac{nf}{z} \end{bmatrix}$$

雅可比矩阵:

$$\bullet \quad J = \begin{bmatrix} \frac{df_1}{dx} & \frac{df_1}{dy} & \frac{df_1}{dz} \\ \frac{df_2}{dx} & \frac{df_2}{dy} & \frac{df_2}{dz} \\ \frac{df_3}{dx} & \frac{df_3}{dy} & \frac{df_3}{dz} \end{bmatrix} = \begin{bmatrix} \frac{n}{z} & 0 & -\frac{nx}{z^2} \\ 0 & \frac{n}{z} & -\frac{ny}{z^2} \\ 0 & 0 & -\frac{nf}{z^2} \end{bmatrix}$$

xii. 视口变换(只对均值, 不对协方差)

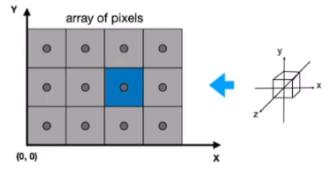
投影变换后:

- 高斯核中心 $x_k = [x_0 \ x_1 \ x_2]^T$
- 高斯核: $r_k(x) = G_{V_k}(x - x_k)$
- 足迹函数

像素坐标系:

- 高斯核中心 $\mu = [\mu_1 \ \mu_2 \ \mu_3]^T$
 - 平移 + 缩放
- 足迹渲染: 离散计算

$$\circ \quad G(\hat{x}) = \exp \left(-\frac{1}{2}(x - \mu)^T V_k^{-1}(x - \mu) \right)$$



雪球颜色

基函数 & 球谐函数

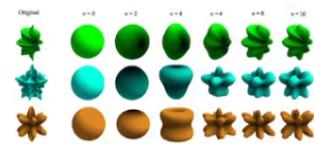
- 任何球面坐标函数可以用多个球谐函数近似

基函数：

- 任何一个函数可以分解为正弦与余弦的线形组合

$$\bullet \quad f(x) = a_0 + \sum_{n=1}^{+\infty} a_n \cos \frac{n\pi}{l} x + b_n \sin \frac{n\pi}{l} x$$

$\cos \frac{n\pi}{l} x, \sin \frac{n\pi}{l} x$ 是基函数



球谐函数：

- 任何一个球面坐标的函数可以用多个球谐函数来近似

$$\bullet \quad f(t) \approx \sum_l \sum_{m=-l}^l c_l^m y_l^m(\theta, \phi)$$

2. \bullet 其中， c_l^m 各项系数， y_l^m 是基函数

球谐函数：

$$\begin{aligned} f(t) &\approx \sum_l \sum_{m=-l}^l c_l^m y_l^m(\theta, \phi) \\ &= c_0^0 y_0^0 + \\ &\quad c_1^{-1} y_1^{-1} + c_1^0 y_1^0 + c_1^1 y_1^1 + \\ &\quad c_2^{-2} y_2^{-2} + c_2^{-1} y_2^{-1} + c_2^0 y_2^0 + c_2^1 y_2^1 + c_2^2 y_2^2 + \\ &\quad c_3^{-3} y_3^{-3} + \dots \end{aligned}$$

- 对方向的函数

3.

4. 其中

- l 表示 阶数
- m 表示 同一阶位置
- c_l^m 为一个 三元组，表示 **R,G,B** 颜色
- 共使用4阶，共 $1 + 3 + 5 + 7 = 16$ 个球
- 形成 16×3 的 系数matrix
- 0阶只有一个颜色

5. 在使用实数球谐函数时，虽然 ϕ 没有显式出现，但它的影响已被转换为方向向量在三维空间中的分量。因此，代码中的 dir(归一化方向向量) 实际上已经包括了 ϕ 的影响。这个方向向量是从 3D 高斯中心指向相机的方向，通过其分量的组合来反映不同方向上的球谐函数分量。

球谐函数 怎么就能 更好的表达颜色？

$y_l^m(\theta, \varphi) = \begin{cases} \sqrt{2} K_l^m \cos(m\varphi) P_l^m(\cos\theta) & m > 0 \\ \sqrt{2} K_l^m \sin(-m\varphi) P_l^{-m}(\cos\theta) & m < 0 \\ K_l^0 P_l^0(\cos\theta) & m = 0 \end{cases}$	$P_n(x) = \frac{1}{2^n \cdot n!} \frac{d^n}{dx^n} [(x^2 - 1)^n]$ $P_l^m = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{d^m}{dx^m} (P_l(x))$ $K_l^m = \sqrt{\frac{(2l+1)(l- m)!}{4\pi(l+ m)!}}$
--	--

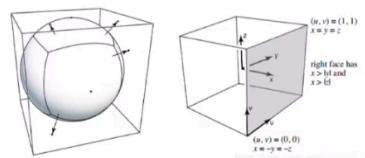
$$y_0^0 = \sqrt{\frac{1}{4\pi}} = 0.28$$

$$y_1^{-1} = -\sqrt{\frac{3}{4\pi}} \frac{y}{r} = -0.49 * \frac{y}{r}$$

$$y_1^0 = \sqrt{\frac{3}{4\pi}} \frac{z}{r} = 0.49 * \frac{z}{r}$$

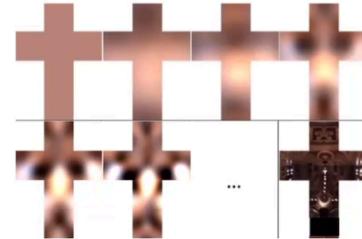
$$y_1^1 = -\sqrt{\frac{3}{4\pi}} \frac{x}{r} = -0.49 * \frac{x}{r}$$

1. 参数量更多
2. 球形环境贴图



在渲染中：

- 用球谐函数来重建亮度
- 1阶到6阶
- 当球谐函数的阶数越高，还原的效果越好



- i.
- ii. 可以用球谐函数重建亮度

3. 球谐函数的球(颜色空间)与高斯椭球(物理空间)无关

合成图片(足迹合成)

1. 直观上， α -blending
2. 实际上，对每个像素进行着色(GPU)，求出每个像素的颜色
对光线上粒子颜色进行求和：

- $T(s)$: 在s点之前，光线没有被阻碍的概率
- $\sigma(s)$: 在s点处，光线碰击粒子（光线被粒子阻碍）的概率密度
- $C(s)$: 在s点处，粒子光出的颜色

特别的：

- splatting没有找粒子的过程
- 需要对高斯球按照深度z排序

$$\begin{aligned} C &= T_i \alpha_i c_i \\ &= \sum_{i=1}^N T_i (1 - e^{-\sigma_i \delta_i}) c_i \end{aligned}$$

where $T_i = e^{-\sum_{j=1}^{i-1} \sigma_j \delta_j}$

3. • 3dgs怎么就快了?
4. 没有类似NeRF通过像素反向找射线粒子的过程
5. 但需要对高斯球按照深度z排序

高性能渲染 & 机器学习

3D GS 怎么就快了？

1. GPU 高性能运算
2. 并行：一个线程负责一个像素，找对应的高斯椭球渲染颜色即可
3. 图像分块 & 高斯对应分块
4. GPU 每个 Block 负责一个区
5. Block 之内可以共享内存

如何进行参数估计？

Cite Info

```
@Article{kerbl3Dgaussians,
  author      = {Kerbl, Bernhard and Kopanas, Georgios and Leimk\"uhler, Thomas and Drettakis, George},
  title       = {3D Gaussian Splatting for Real-Time Radiance Field Rendering},
  journal     = {ACM Transactions on Graphics},
  number      = {4},
  volume      = {42},
  month       = {July},
  year        = {2023},
  url         = {https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/}
}
```

Ideas from ShenLong Wang

Archeologists

1. despite recent hype of gsplats. point-based rendering, and geometric primitive based rendering has been there for years. trace back that direction
2. there exists faster solutions, there also exist better solution, but think about why Gsplats triggered people's excitement. what is the reason for their success?

Industrial Practitioners

1. try to think something coupled with the model itself.
2. some of our recent solutions are a bit too generic in VR/AR in general.
3. try to think out of box and come up with something unique or weird, or

entertaining, or ambitious.

Hackers

1. 2d color image fitting with a bunch of 2d gaussians could be a very interesting experiment.
 - i. take a reference how our last hacker implement differentiable rectangle rendering.
 - ii. i believe 1000 gaussians could fit some images very well.
 - iii. this is the result of geometric primitive fitting using non differentiable method.
 - iv. gaussians are differentiable to xy location by nature
 - v. [Primitive Image: Approximating Images with Basic Shapes, Creating SVG approximations of images using basic shapes](#)
 - vi. SIREN(Sinusoidal Representation Networks)'s image fitting could be another reference (you can consider its a nerf version)
2. rethink the design of sfm point cloud init.
 - i. could you come up with some other initialization that doesn't require sfm?
 - ii. could you theoretically analyze why or why not gsplats is sensitive to init?
 - iii. some toy 1d/2d fitting example might help.
3. repurpose monocular depth method to output gaussians splats
 - i. you can overfit a few images first
 - ii. think about what are the challenges and can it generalize