

# 计算机图形学报告

## 基于 Gaussian Surfels 的 3DGS 表面重建与可视化实验

09023301 张凯轩

2026 年 1 月 7 日

### 1 实验目的与任务概述

本课程项目以 3DGS 为基础，阅读并尝试运行 2024 年 SIGGRAPH 论文“High-quality Surface Reconstruction using Gaussian Surfels”提出的 Gaussian Surfels 模型。根据项目要求和个人理解，我对本实验的目标与任务设置为：

1. 环境配置与编译：完成 Conda 环境搭建，包含 CUDA 光栅化模块与 SIBR 可视化等组件部分；
2. 代码阅读与理解：解释 3DGS 的关键训练/渲染流程，梳理 Gaussian Surfels 相比原 3DGS 的核心的改进；
3. 数据集验证与测试：首先在标准 DTU 场景上训练与测试，检查重建效果与论文描述是否一致；随后在 HOPE 数据集（桌面物体、存在遮挡）上训练，观察鲁棒性与新视角合成效果
4. surfels 论文改进使用的损失函数基于法线和深度一致性，类似一种自监督的损失函数，如果我们尝试引入带有一定先验信息的深度值来进行约束，测试训练效果会不会有所提升；实验中可以尝试引入单目深度估计（MDE）模型作为深度先验，增加单一深度信息的损失项并重新训练，对比观察一下训练结果。

本项目主要基于官方 Gaussian Surfels 实现进行实验与改进，相关链接如下：

Gaussian Surfels 项目主页：

[https://turandai.github.io/projects/gaussian\\_surfels/](https://turandai.github.io/projects/gaussian_surfels/)

DTU 相关格式整理数据集（包含法线信息）：

<https://huggingface.co/datasets/turandai/gaussian-surfels-dtu>

自测相关数据集 HOPE 数据集：

<https://github.com/swtyree/hope-dataset>

### 2 实验环境与配置

#### 2.1 硬件与系统环境

本项目的训练与可视化均在云服务器上进行，一些性能配置信息如下，可以结合此对实际的训练时间做参考

项目	配置
操作系统	Ubuntu 22.04
Conda	Miniconda (conda3)
Python	3.7
CUDA	11.6
GPU	vGPU-32GB (32GB) ×1
CPU	12 vCPU Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz
内存	72GB

表 1: 实验环境

## 2.2 环境创建与编译

Conda 环境创建：由于云服务器上直接 conda 安装整个环境 sovling 速度较慢，实际配置使用 pip 安装必要的库步步调试下载，具体不再赘述。

重新编译 CUDA 光栅化模块：

```

1 cd gaussian_surfels/submodules/diff-gaussian-rasterization
2 python setup.py install
3 pip install .

```

SIBR 浏览器的编译与安装依赖 CMake，结合调试及报错信息补充安装必要的库和依赖。

如果想要使用相关环境，可以直接使用课程实验 GitHub 仓库中或者网盘中的环境压缩包进行解压使用，包含训练以及 SIBR 等所有必要组件。

## 3 关键代码与分析

本节按照数据流动的逻辑，从 Surfel 表示构建、可微渲染、损失优化到最终表面重建，逐步梳理 Gaussian Surfels 的核心技术环节。代码路径均在本仓库中可直接定位。

### 3.1 方法总览

图1 给出了论文的 pipeline。左侧为 Gaussian Surfels 表示与初始化（从点云到 surfel），中部对应损失函数，右侧则是 volumetric cutting 与 mesh 重建步骤。

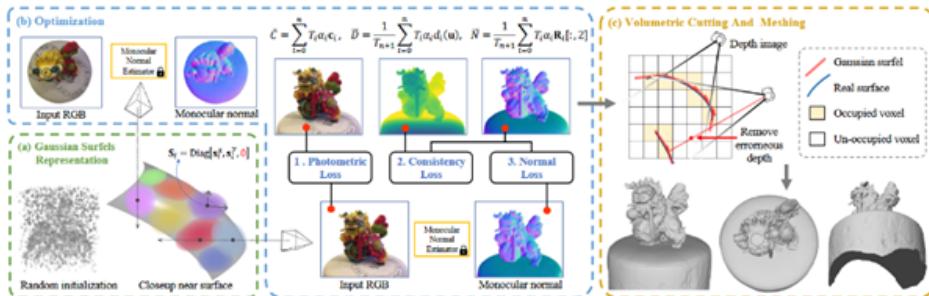


图 1: Gaussian Surfels 训练-重建全流程：左侧为 Surfels 表示与初始化，中间为三类损失，右侧为 volumetric cutting 与 mesh

### 3.2 Gaussian Surfels 表示与初始化

在经典 3DGS 中，高斯以位置、协方差（缩放 + 旋转）、颜色（SH 系数）与不透明度表示。Gaussian Surfels 的核心工作是强调使每个高斯的趋近于切平面：通过旋转参数生成法向，并在初始化时对某一轴缩放，使其更像面片。

**法向表示与提取** Gaussian Surfels 将每个高斯的旋转矩阵  $\mathbf{R}_i$  的第三列定义为该 surfel 的法向：

$$\mathbf{n}_i = \mathbf{R}_i \mathbf{e}_3 = \begin{bmatrix} r_{13} \\ r_{23} \\ r_{33} \end{bmatrix}$$

其中  $\mathbf{e}_3 = [0, 0, 1]^T$  为局部坐标系的 z 轴。旋转参数既控制高斯的朝向，也可以表示表面法向。

**Surfel 初始化：各向异性压扁** 为使高斯更像薄片状表面元素，初始化时对 z 方向缩放施加极小值：

$$\mathbf{S}_i = \text{diag}(s_x, s_y, s_z - 10^{10})$$

使协方差矩阵在法向方向上几乎退化，将体积高斯压扁为近二维的 surfel。同时，若点云提供初始法向  $\mathbf{n}_0$ ，通过 `normal2rotation` 将其转换为旋转四元数，确保 surfel 法向与表面法向对齐。

对应实现在 `gaussian_surfels/scene/gaussian_model.py`: `get_normal` 函数通过四元数旋转矩阵的第三列提取法向；`create_from_pcd` 函数在启用 surface 配置时，使用点云法向初始化旋转，并通过 `scales[..., -1] -= 1e10` 压扁 z 方向尺度。

初始化旋转与 z 缩放：

```

1 if self.config[0] > 0:
2     rots = normal2rotation(torch.from_numpy(normals).to(torch.float32)).to("cuda")
3     scales[..., -1] -= 1e10 # squeeze z scaling
4 else:
5     rots = torch.zeros((fused_point_cloud.shape[0], 4), device="cuda")
6     rots[:, 0] = 1

```

这样的高斯表示可以匹配后续的表面约束损失。

**深度与法向的计算** 初始化后，系统在渲染过程中需要精确计算每个像素的深度与法向，以支持表面约束。Gaussian Surfels 获得更可靠的深度图。如图2 所示，该方法先在图像空间中统一视线方向，再将前景高斯投影到深度平面，从而与 surfel 表示相匹配。

基于渲染的深度图  $D(u, v)$ （像素  $(u, v)$  处的深度值），通过反投影可将像素恢复为世界坐标 3D 点：

$$\mathbf{X}(u, v) = D(u, v) \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (1)$$

其中  $\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$  为相机内参矩阵 ( $f_x, f_y$  为焦距,  $c_x, c_y$  为主点)。

随后通过有限差分计算深度图的梯度，推导每个像素处的曲面法向：

$$\frac{\partial \mathbf{X}}{\partial u} = D(u, v) \mathbf{K}^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \frac{\partial D}{\partial u} \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (2)$$

$$\frac{\partial \mathbf{X}}{\partial v} = D(u, v) \mathbf{K}^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \frac{\partial D}{\partial v} \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (3)$$

法向由偏导数叉积的归一化得到：

$$\mathbf{n}_{\text{depth}} = \text{normalize} \left( \frac{\partial \mathbf{X}}{\partial u} \times \frac{\partial \mathbf{X}}{\partial v} \right) \quad (4)$$

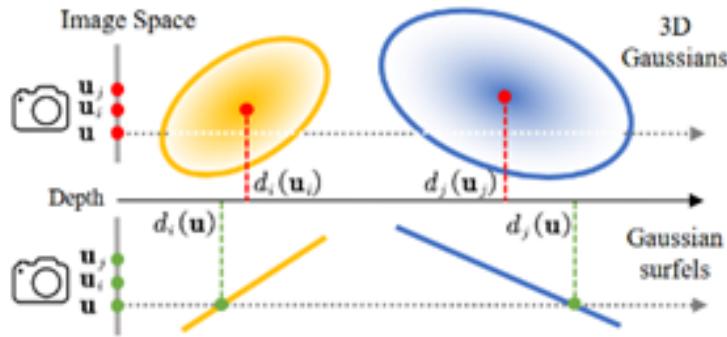


图 2: 更精确的深度/法向计算示意：统一视线后，再由深度平面匹配 surfel 中心

以下为该过程的 Python 实现示例（来自 `gaussian_surfels/utils/loss_utils.py`）：

```

1 def depth2normal(depth, mask, viewpoint_cam):
2     """从深度图计算法向：通过反投影与有限差分"""
3     K = viewpoint_cam.intrinsics # 内参矩阵
4     H, W = depth.shape
5
6     # 反投影：将像素坐标 (u,v) 转为相机坐标中的 3D 点
7     u, v = torch.meshgrid(torch.arange(W), torch.arange(H),
8                           indexing='ij')
9     u, v = u.float(), v.float()
10    ones = torch.ones_like(u)
11    uv1 = torch.stack([u, v, ones], dim=-1) # [H, W, 3]
12
13    # K^-1 @ [u, v, 1]
14    K_inv = torch.linalg.inv(K)
15    points_dir = (uv1 @ K_inv.T).squeeze() # [H, W, 3]
16
17    # 世界坐标 X = D * K^-1 @ [u, v, 1]
18    X = depth.unsqueeze(-1) * points_dir
19
20    # 有限差分: Sobel 算子计算梯度
21    dX_du = X[:, :-1, :] - X[:, 1:, :] # 水平梯度

```

```

22 dX_dv = X[:-1, :, :] - X[1:, :, :] # 坚直梯度
23
24 # 补齐边界，使尺寸对齐
25 dX_du = torch.cat([dX_du, dX_du[:, -1:, :], dim=1)
26 dX_dv = torch.cat([dX_dv, dX_dv[-1:, :, :], dim=0)
27
28 # 叉积：法向 = (X/u) × (X/v)
29 normal = torch.cross(dX_du, dX_dv, dim=-1)
30 normal = torch.nn.functional.normalize(normal, dim=-1)
31
32 return normal * mask.unsqueeze(-1)

```

这个过程无需真实标注的法向数据，而是基于渲染深度信息，指导 surfel 的优化方向。

### 3.3 可微渲染

基于 Surfel 表示，渲染管线仍采用 3DGS 的  $\alpha$ -blending 框架，但输出额外包含法向与深度信息用于几何监督。

**3DGS 渲染** 3D Gaussian Splatting 将场景表示为一组 3D 高斯函数的集合。每个高斯  $\mathcal{G}_i$  的参数定义为：位置（均值） $\mu_i \in \mathbb{R}^3$ ，协方差矩阵  $\Sigma_i = \mathbf{R}_i \mathbf{S}_i \mathbf{S}_i^T \mathbf{R}_i^T$ （其中  $\mathbf{R}_i$  为旋转矩阵由四元数参数化， $\mathbf{S}_i$  为缩放矩阵），球谐函数系数  $\mathbf{c}_i$  用于表示视角相关颜色，以及不透明度  $\alpha_i \in [0, 1]$ 。

渲染时，对于像素  $\mathbf{p}$ ，颜色由沿视线的高斯按深度排序后进行  $\alpha$ -blending，其数学表达为：

$$C(\mathbf{p}) = \sum_{i \in \mathcal{N}} c_i \alpha'_i \prod_{j=1}^{i-1} (1 - \alpha'_j) \quad (5)$$

其中加权不透明度定义为：

$$\alpha'_i = \alpha_i \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_i)^T \Sigma_i^{-1} (\mathbf{x}_i - \mu_i)\right) \quad (6)$$

而颜色  $c_i$  由球谐函数根据视角方向计算得到。

**渲染输出与监督** Gaussian Surfels 在渲染时不仅输出 RGB 图像，还同时输出法向图、深度图与不透明度图，用于几何监督，为后续的表面一致性约束提供了基础：

```

1 render_pkg = render(viewpoint_cam, gaussians, pipe, background, patch_size)
2 image, normal, depth, opac =
3     render_pkg["render"], render_pkg["normal"],
4     render_pkg["depth"], render_pkg["opac"]
5 )

```

### 3.4 损失函数与监督

损失函数实现主要位于 `gaussian_surfels/utils/loss_utils.py` 与 `gaussian_surfels/train.py`。完整损失函数为多项加权和：

$$\mathcal{L} = \mathcal{L}_{\text{RGB}} + \lambda_{\text{mask}} \mathcal{L}_{\text{mask}} + \lambda_{\text{surf}}(t) \mathcal{L}_{\text{surf}} + \lambda_{\text{opac}} \mathcal{L}_{\text{opac}} + \lambda_{\text{curv}} \mathcal{L}_{\text{curv}} \quad (7)$$

其中表面约束权重随训练逐步增强，定义为  $\lambda_{\text{surf}}(t) = 0.01 + 0.1 \cdot \min(2t/T, 1)$ 。

**(1) 颜色重建: L1 + DSSIM** 颜色监督采用 L1 范数与结构相似性 (SSIM) 的加权组合:

$$\mathcal{L}_{\text{RGB}} = (1 - \lambda_{\text{dssim}}) \|I - \hat{I}\|_1 + \lambda_{\text{dssim}} (1 - \text{SSIM}(I, \hat{I})) \quad (8)$$

其中  $I$  为真实图像,  $\hat{I}$  为渲染图像。

```
1 L11 = l1_loss(image, gt_image)
2 loss_rgb = (1.0 - opt.lambda_dssim) * L11 + opt.lambda_dssim * (1.0 - ssim(image,
    gt_image))
```

**(2) 表面约束: 深度-法向一致性** Gaussian Surfels 的核心创新在于自监督的表面一致性约束。基于第 3.2 节介绍的深度与法向计算方式, 我们用余弦距离约束渲染法向  $\mathbf{n}_{\text{render}}$  与深度推导的法向  $\mathbf{n}_{\text{depth}}$  一致:

$$\mathcal{L}_{\text{surf}} = 1 - \frac{1}{|\Omega|} \sum_{\mathbf{p} \in \Omega} \mathbf{n}_{\text{render}}(\mathbf{p}) \cdot \mathbf{n}_{\text{depth}}(\mathbf{p}) \quad (9)$$

其中  $\Omega$  为有效像素区域 (mask)。这种自监督约束无需真实法向标注, 即可促使高斯形成连续光滑的表面。图3展示了 surfel 分布、深度中心位置、法向方向以及深度-法向一致性的三种情况对比, 其中只有当三者都正确时才能形成高质量的表面重建。

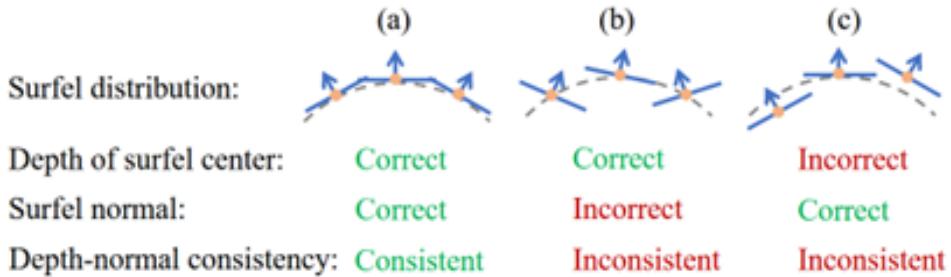


图 3: Surfel 深度-法向一致性对比: (a) 正确情况, (b) 法向错误, (c) 深度错误

```
1 normal = torch.nn.functional.normalize(normal, dim=0) * mask_vis
2 d2n = depth2normal(depth, mask_vis, viewpoint_cam)
3 loss_surface = cos_loss(normal, d2n)
```

**(3) 可选单目先验: 法向** 当提供单目法向预测 (如 normal prior) 且对应权重非零时, 训练会额外引入以下监督项:

$$\mathcal{L}_{\text{monoN}} = \|\mathbf{n}_{\text{render}} - \mathbf{n}_{\text{mono}}\|_2 \quad (10)$$

这些先验信息主要目的是提升几何稳定性, 特别是在纹理弱或存在高光等区域, 缓解不确定性。

**(4) opacity 与曲率** 不透明度正则项抑制高斯过度透明或堆积漂浮:

$$\mathcal{L}_{\text{opac}} = \sum_i \alpha_i (1 - \alpha_i) \quad (11)$$

该项在  $\alpha_i = 0.5$  时达到最大惩罚, 促使高斯向完全透明或完全不透明状态收敛。曲率正则项通过惩罚相邻 surfel 法向差异来平滑表面:

$$\mathcal{L}_{\text{curv}} = \sum_{i,j \in \mathcal{N}(i)} \|\mathbf{n}_i - \mathbf{n}_j\|_2 \quad (12)$$

### 3.5 训练策略

训练入口位于 `gaussian_surfels/train.py` 的 `training(...)`。其核心流程是：

1. 随机采样一个相机视角；
2. 调用可微渲染器得到彩色图、法向、深度与不透明度等；
3. 计算多项损失并反向传播；
4. 更新高斯参数，同时执行 `densify/prune` 等调整。

主要损失项的组合：

```

1 L11 = l1_loss(image, gt_image)
2 loss_rgb = (1.0 - opt.lambda_dssim) * L11 + opt.lambda_dssim * (1.0 - ssim(image,
    gt_image))
3 loss_surface = cos_loss(normal, d2n)
4 loss = 1.0 * loss_rgb + 0.1 * loss_mask
5 loss += (0.01 + 0.1 * min(2 * iteration / opt.iterations, 1)) * loss_surface

```

### 3.6 Mesh 重建与后处理

本项目的 mesh 生成在渲染脚本中完成：`gaussian_surfels/render.py` 在渲染 train 视角时会对可见区域进行点采样与清理，然后调用 `poisson_mesh(...)` 执行 Screened Poisson 重建。

(1) 从训练视角采样点云 渲染 train 集时，通过深度与法向进行采样并做 occupancy grid prune，聚合得到用于重建的点云。

(2) Screened Poisson 与网格清理 `gaussian_surfels/utils/general_utils.py` 中的 `poisson_mesh` 关键流程包括：

1. Screened Poisson 重建（输出 `_plain.ply`）；
2. 通过 KNN 估计顶点颜色和到采样点的距离；
3. 基于阈值剔除离群点/面，并尝试补洞；
4. Laplacian 平滑，生成 `_pruned.ply`。

核心片段：

```

1 ms.generate_surface_reconstruction_screened_poisson(depth=depth, preclean=True,
    samplespernode=1.5)
2 ms.save_current_mesh(path + '_plain.ply')
3
4 ms.compute_selection_by_condition_per_vertex(condselect=f"q>{thrsh}")
5 ms.meshing_remove_selected_vertices()
6 ms.meshing_close_holes(maxholesize=300)
7 ms.apply_coord_laplacian_smoothing(stepssmoothnum=3, boundary=True)
8 ms.save_current_mesh(path + '_pruned.ply')

```

本实验 Poisson 重建耗时很长（具体的底层计算我不太清楚），因此本项目仅在少量 DTU 场景的训练结果上进行了 mesh 测试；HOPE 场景也仅进行了训练与渲染展示。

图4 用来解释 volumetric cutting：alpha 混合导致的前景/背景交叉会直接体现在深度曲线上，通过 cut 操作可以去除漂浮的错误高斯并获得干净的深度带入 Poisson。

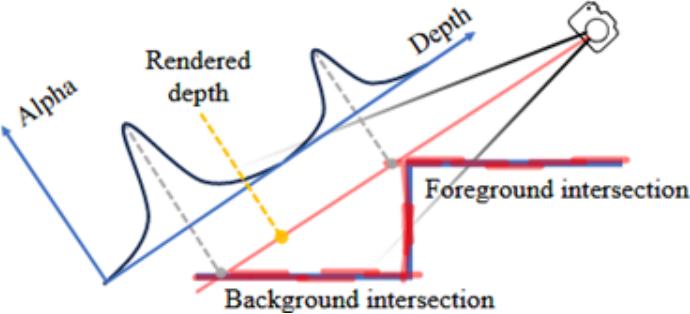


图 4: Volumetric cutting: 处理  $\alpha$  混合导致的深度错误

## 4 改动与结果分析

### 4.1 改动

SIBR 可视化与相关输出精细化；

**引入深度先验损失：**原始 Gaussian Surfels 仅包含法向先验监督，本项目额外引入了单目深度先验（如 DA3 depth）的 L1 损失项：

$$\mathcal{L}_{\text{depth}} = \|D_{\text{render}} - D_{\text{mono}}\|_1 \quad (13)$$

需要注意的是，源代码中作者也提供了使用深度先验约束的注释，作者在工作中应该也可能尝试了这种方法，但并未在论文中讨论。实验中引入该损失项进行了少部分场景的训练测试，结果并未显著提升，具体分析见后文。

### 4.2 实验设置：DTU 与 HOPE 数据集

**DTU** DTU 数据集选取若干场景进行训练，分别尝试：

全部视角训练；

1:1 half split（通过 `--idr_half_split` 划分 train/test）。

**HOPE** HOPE 数据集选取两个 scene 进行全部训练（子集与 full 版本，子集包含 100 张图片，full 包含全部 300 张左右帧图像），并在 SIBR 中进行渲染效果展示。

### 4.3 结果与现象分析

注意，SIBR 可视化的视频均在 videos/ 目录下，可以直接查看，但关于加入了深度先验的训练点云并没有拍摄保存。此外本节仅统计训练时间与 PSNR 等指标，关于倒角距离参数由于 DTU 的原始 GROUND TRUTH 没有获取，且 mesh 计算时间很长，故没有进行计算。

### 4.3.1 DTU: 全部训练 vs 1:1 half split

**场景与训练方式** DTU 部分训练了以下场景: scan105, scan106, scan110, scan114, scan118, 并对其中部分场景进行了 half split(--idr\_half\_split):scan105\_half, scan106\_half, scan110\_half, scan114\_half, scan118\_half。

**训练时间与指标** 训练耗时与质量指标统计如表 2。

场景	模式	训练时间 (分钟)	训练 PSNR	测试 PSNR
scan105	全部	22.14	34.48	NaN
scan105_half	half split	22.67	34.20	30.31
scan106	全部	22.71	35.80	NaN
scan106_half	half split	23.98	35.42	31.61
scan110	全部	24.02	32.91	NaN
scan110_half	half split	24.70	32.52	29.96
scan114	全部	24.02	31.75	NaN
scan114_half	half split	24.47	31.54	27.02
scan118	全部	24.16	35.62	NaN
scan118_half	half split	15.55	35.36	31.88

表 2: DTU 场景训练结果统计 (全部视角训练 vs 1:1 half split)

**现象分析** 综合表 2 的结果, 以及论文中原作者的分析: 实验中实际的训练时常相对作者报告的有增加, 可能与硬件差异及环境配置有关, 具体原因没有深究。但是 DTU 数据集各个场景的训练时间相对一致, PSNR 指标也在合理范围内, 与论文描述 30 左右基本一致。

### 4.3.2 DTU: Mesh 重建与后处理

由于 Poisson mesh 重建与后处理耗时较长, 本项目仅对 3 个 DTU 输出进行了 mesh 测试, 且均生成了 poisson\_mesh\_8\_plain.ply 与 poisson\_mesh\_8\_pruned.ply:

scan105 (全部训练输出); scan105\_half (1:1 half split 输出); scan106 (全部训练输出)。  
mesh 后处理流程由 poisson\_mesh 完成, 详细的效果图在后面部分体现。

### 4.3.3 HOPE: 训练结果统计

HOPE 部分训练了两个场景 (scene\_0000、scene\_0001), 分别包含子集与 full 版本; 另外还在 scene\_0000 上训练了引入深度先验的两个变体。表 3 统计了相关结果:

**观察与分析** 表 3 显示: HOPE 的训练 PSNR 相比 DTU 明显偏低, 应该与场景中存在的遮挡、视角稀疏、以及没有设置掩膜前景提取等因素有关; full 通常较子集难度更高 (耗时略长, PSNR 略低), 可能原因是引入更多的视角使得训练更复杂不稳定, 产生噪声或 floaters 更多; 深度先验配置在 HOPE 上表现出明显的加速效果, 但有可能是因为平台的算力调度有问题, 前面提到的训练时间的较大差异也可能是由于某些不确定因素, 这里不做具体解释探讨。

场景	配置	训练时间（分钟）	训练 PSNR
hope_scene_0000	baseline	13.49	24.58
hope_scene_0000_full	baseline	14.87	22.62
hope_scene_0001	baseline	13.35	25.34
hope_scene_0001_full	baseline	13.57	21.87
hope_scene_0000_metric	深度先验	8.12	23.80
hope_scene_0000_rel	深度先验	8.26	23.50

表 3: HOPE 场景训练结果统计

#### 4.3.4 深度先验实验对比

本项目在三个场景上验证了引入深度先验损失的效果。表 4 统计了深度先验与对应原始形式训练的对比结果：

场景	配置	训练时间（分钟）	训练 PSNR	PSNR 差值 (dB)
scan105	baseline	22.14	34.48	—
scan105_da3depth	深度先验	22.14	34.50	+0.02
hope_scene_0000	baseline	13.49	24.58	—
hope_scene_0000_metric	深度先验	8.12	23.80	-0.78
hope_scene_0000_rel	深度先验	8.26	23.50	-1.08

表 4: 深度先验实验对比

现象与分析 表 4 及相关观察显示：

##### 1. PSNR 指标：

- DTU 场景 (scan105): PSNR 几乎无差异，表明深度先验影响相较有限；
- HOPE 场景: PSNR 下降，可能原因包括：
  - 单目深度估计本身存在噪声与尺度歧义，当权重过大时可能会导致约束过度；
  - 纹理丰富、遮挡复杂的真实场景，过强的几何约束可能损害了 RGB 渲染精度。

##### 2. 权重调整:

后续工作应尝试类似的动态衰减权重或自适应平衡，以在保持几何质量的同时需要注意维持 RGB 精度。

由于无法计算倒角距离等几何指标，无法直接评估深度先验对几何质量的提升，这里只能使用 PSNR 作为参考。但从整体结果来看，实验中深度先验对最终的 RGB 重建精度影响有限，这与证明了作者使用的深度-法向一致性约束已经比较有效。同时也揭示了一个现象，在 3D 重建任务中，目前的科研工作往往需要在几何质量与渲染精度之间进行权衡，两者很难做到兼顾，这也为后续的科研工作提供了思路与方向。

#### 4.4 结果截图

screenshots/ 目录中保留了 MeshLab 可视化截图示例，包含不同场景的 mesh 重建结果与模型展示。

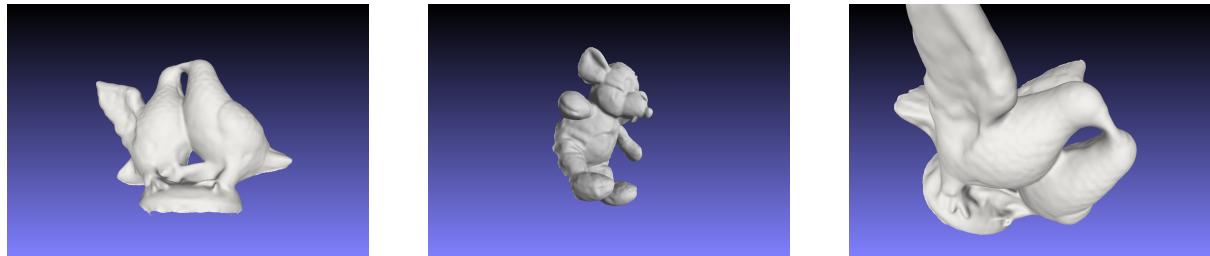


图 5: Mesh 重建结果展示: (左) Bird 103, (中) Toy 100, (右) Bird 204



图 6: 重建与模型可视化: (左) Toy 201, (中) Bird 模型 00, (右) Toy 模型 103

### 链接

课程实验 GitHub 仓库: <https://github.com/sylgha/computer-graphics-project-3DGS>  
完整项目文件（包含 data、output 等）: <https://pan.quark.cn/s/f81a73635991>, 提取码: KHCA