Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance

潘成

摘要

IDR 是用于解决多视图三维曲面重建问题的神经网络架构，它同时学习未知的几何形状、相机参数，以及一个用于近似从表面反射到相机的光的神经渲染器。几何图形被表示为一个神经网络的零水平集，而神经渲染器，从渲染方程派生出来，能够 (隐式地) 建模一组广泛的照明条件和材料。我们用 DTU MVS 数据集中具有不同材料属性、光照条件和有噪声的相机初始化的物体的真实世界 2D 图像训练我们的网络。实验使用 DTU MVS 数据集中具有不同材料属性、光照条件和有噪声的相机初始化的物体的真实世界 2D 图像训练 IDR 网络。该模型可以产生高保真度、分辨率和细节的最先进的 3D 表面重建。

关键词：神经网络；三维重建；渲染方程；纹理迁移

1 引言

从 2D 图像中学习 3D 形状是一个基本的计算机视觉问题。最近成功解决这一问题的神经网络方法涉及使用 (神经) 可微渲染系统以及选择 (神经)3D 几何表示。微分渲染系统大多基于射线投射或者跟踪，或栅格化，而常用的表示 3D 几何的模型包括点云，三角形网格，在体积网格上定义的隐式表示，最近还有神经隐式表示，即神经网络的零水平集。

隐式神经表示的主要优点是它们在表示具有任意形状和拓扑的表面时的灵活性，以及无网格 (即，没有固定的先验离散化，如体积网格或三角形网格)。到目前为止，具有隐式神经表示的可微分渲染系统没有结合在图像中产生忠实的 3D 几何外观所需的照明和反射率属性，也没有处理可训练的相机位置和方向。

本次实验的目标是实现一个端到端神经架构系统，可以从蒙板 2D 图像和粗略的相机估计中学习 3D 几何，且不需要额外的监督，见图 1。为此，我们将像素的颜色表示为场景中三个未知因素 (几何形状、外观和摄像机) 中的可微函数。这里，外观是指定义表面光场的所有因素的总和，不包括几何形状，即表面双向反射率分布函数 (BRDF) 和场景的照明条件。我们称这种架构为隐式可微渲染器 (IDR)。我们表明，IDR 能够近似光反射从一个三维形状表示为神经网络的零水平集。该方法可以处理来自特定受限族的表面现象，即所有表面光场都可以表示为表面上点及其法线和观测方向的连续函数。此外，将全局形状特征向量合并到 IDR 中可以提高其处理更复杂外观 (例如，间接照明效果) 的能力。

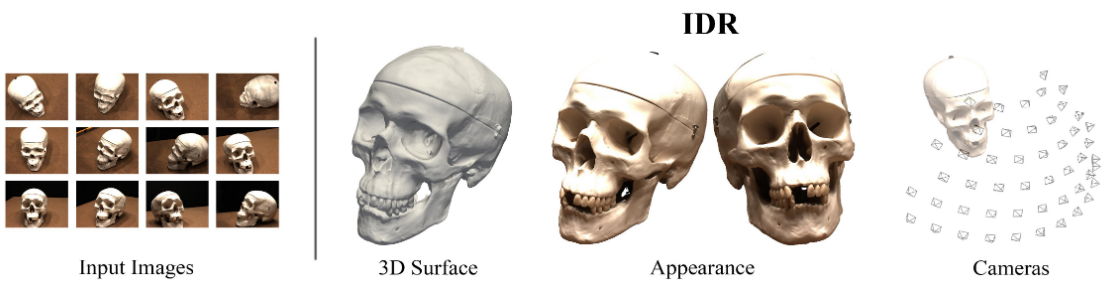

图 1: 介绍 IDR: 端到端学习几何，外观和相机从图像

该方法的主要贡献是:

1. 端到端架构，处理未知的几何形状、外观和摄像头。
2. 表达神经隐式曲面对相机参数的依赖关系。
3. 在精确和有噪声的相机信息中，从现实生活中的 2D 图像中生成具有广泛外观的不同物体的最先进的 3D 表面重建

2 相关工作

用于学习几何的可微渲染系统 (主要) 有两种风格: 可微光栅化和可微射线投射。由于目前的工作集中于第二类，我们首先集中讨论这一类工作。然后，我们将描述多视图曲面重建和神经视图合成的相关工作。

2.1 隐式表面可微光线投射

可微光线投射通常用于隐式形状表示，例如在体积网格上定义的隐式函数或隐式神经表示，其中隐式函数可以是占用函数、带符号距离函数 (SDF) 或任何其他带符号隐式。在相关工作中，使用体积网格来表示 SDF，并实现光线投射可微的渲染器。它们近似于每个体积单元中的 SDF 值和曲面法线。例如使用预先训练的 DeepSDF 模型的球面追踪，并近似深度梯度，即通过区分球面跟踪算法的各个步骤来实现 DeepSDF 网络的潜在代码[1]；或者使用场探测来促进可微光线投射[2]。与这些工作不同的是，IDR 利用了隐式曲面的精确和可微的曲面点和法线，并考虑了更一般的外观模型，以及处理噪声摄像机。

2.2 多视图曲面重建

在图像的捕获过程中，深度信息会丢失。假设摄像机已知，经典的多视点立体 (MVS) 方法[3]试图通过匹配不同视点上的特征点来再现深度信息。然而，需要经过深度融合[4]的后处理步骤，然后是泊松曲面重建算法，才能产生有效的 3D 水密曲面重建。最近的方法使用场景集合来训练深度神经模型，用于 MVS 流水线的子任务，例如特征匹配或深度融合，或者用于端到端的 MVS 流水线。当摄像机参数不可用时，并且给定来自特定场景的一组图像，应用运动结构 (SFM) 方法[5]来再现摄像机和稀疏 3D 重建。Tang 和 Tan 使用具有集成的可微束调整层的深层神经体系结构来提取参考帧深度的线性基础和来自附近图像的特征，并针对每一次正传中的深度和相机参数进行优化。与这些工作不同的是，IDR 是用来自单个目标场景的图像进行训练的，产生了准确的水密 3D 表面重建。

2.3 基于视图合成的神经表示

最近的工作训练神经网络来预测新的视角和 3D 场景或物体的一些几何表示，从具有已知摄像头的有限图像集合中预测。使用 LSTM 对场景几何体进行编码以模拟光线行进过程。使用神经网络来预测体积密度和视点相关的发射辐射，以从具有已知相机的一组图像合成新的视点。使用神经网络从输入图像和几何图形学习表面光场，并预测未知视图或场景照明。与 IDR 不同的是，这些方法不会生成场景几何体的 3D 曲面重建，也不会处理未知的摄影机。

3 本文方法

3.1 本文方法概述

我们的目标是从可能带有粗糙或噪声的摄像机信息的掩模 2D 图像中重建对象的几何形状。我们有三个未知数：(i) 由参数 $\theta \in R^m$ 表示的几何模型 (ii) 由 $\gamma \in R^n$ 表示的外观；以及 (iii) 由 $\tau \in R^k$ 表示的相机。符号和设置如图 2所示

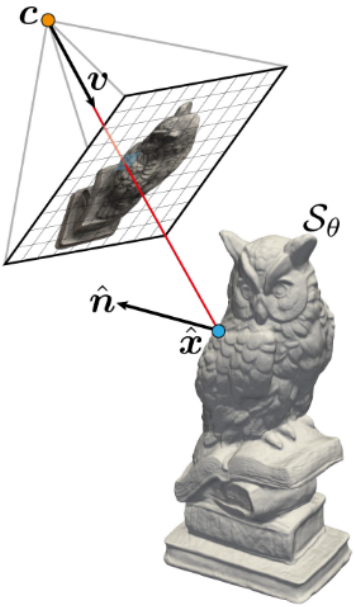

图 2: 方法示意图

我们将几何表示为神经网络 $(MLP)f$ 的零水平集，其中 θ 是一个可学习的参数

$$S_\theta = x \in R^3 | f(x;\theta) = 0 \tag{1}$$

3.2 网络架构

图 3可视化 IDR 架构。灰色块描述输入向量，橙色块描述输出向量，每个蓝色块描述具有 softplus 激活函数的全连接的隐藏层。在渲染网络中，我们使用隐藏层之间的重新激活和 $tanh$ 作为输出，以获得有效的颜色值。同时，对于隐式神经网络和渲染网络的输入，我们通过位置编码将输入映射到更高维的向量，使得网络可以更好的学习到高频变化的信息。

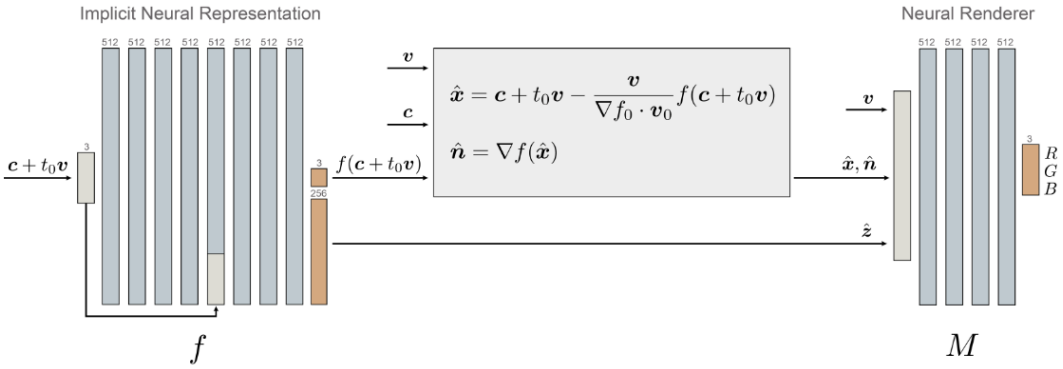

图 3: IDR 网络架构

由于光线追踪的部分是不可微的，所以我们需要按照公式 2 来计算 x，从而 x 课一堆对 θ, c, v 近似微分从而可以进行梯度回传去修改几何模型和相机参数。

$$x = c + t_0 v - \frac{v}{\bigtriangledown f_0 \cdot v_0} f(c + t_0 v) \tag{2}$$

3.3 光线步进

光线与物体求交的部分我们采用经典的 RayMarching 算法，该算法每次让光线前进当前位置距离几何模型的最小距离直到与模型相交。我们将球体跟踪步数限制为 10 步，以防止长时间收敛的情况，例如，当光线通过非常接近曲面而没有穿过曲面时。然后，对于不收敛的光线，我们从光线与单位球体的第二个 (最远的) 交点开始运行另 10 次球体跟踪迭代，得到符号。我们对符号 $t, \bar{t}$ 之间的 100 个相等大小的步长进行采样 $t < t_1 < t_2 < ... < t_{100} < \bar{t}$，并找到第一对连续的步长 t_i, t_{i+1}，使 $sign(f(c + t_i v)) \neq sign(f + c_{i+1} v)$，即，描述曲面的第一个光线交叉的符号过渡。然后，使用 ti 和 tj 作为初始值，运行 8 次割线算法以求交点 x_0 的近似值。对于与多个像素相关联的多条光线，该算法在 GPU 上并行

3.4 损失函数定义

$$S(\theta, \tau) \begin{cases} 1 & R(\tau) \cap S_\theta \neq \oslash \\ 0 & otherwise \end{cases} \tag{3}$$

$$S_\alpha(\theta, \tau) = sigmoid(-\alpha \min_{t \geqslant 0}(c + tv; \theta)) \tag{4}$$

$S(\theta, \tau)$ 用于表示光线与几何是否相交，相交用 1 表示，不相交用 0 表示，用于计算之后的 MASK 损失但是这样表示将会导致不可微，所以将 $S(\theta, \tau)$ 写成 $S_\alpha(\theta, \tau)$ 的形式。

设 $I_p \in [0, 1]^3, O_p \in 0, 1$ 为 RGB 值和掩码值 (分别为) 对应于用摄像机 $c_p(\tau)$ 和方向 $v_p(\tau)$ 拍摄的图像中的像素 p，其中 $p \in P$ 输入图像集合中的所有像素的索引，并且 $\tau \in R^k$ 表示场景中所有摄像机的参数。我们的损失函数的形式为：

$$loss(\theta, \gamma, \tau) = loss_{RGB}(\theta, \gamma, \tau) + \rho loss_{MASK}(\theta, \tau) + \lambda loss_E(\theta) \tag{5}$$

我们在 P 中的小批次像素上训练这种损失；为了保持记号简单，我们用 P 表示当前的小批次。对于每个 $p \in P$，我们使用球面追踪算法来计算射线 $R_p(\tau)$ 和 S_θ 的第一个交点 $c_p + t_{p,0}, v_p$。设 $p^{in} \in P$ 中的 P 是已找到交集的像素 p 的子集，且 $O_p = 1$。所以 RGB 损失为：

$$loss_{RGB}(\theta, \gamma, \tau) = \frac{1}{|P|} \sum_{p \in P^{in}} |I_p - L_p(\theta, \gamma, \tau)| \tag{6}$$

其中 $|\cdot|$ 表示 L1 范数。令 $P^{out} = P \setminus P^{in}$ 表示小批次中没有光线与几何相交或 $O_p = 0$ 的索引。所以 MASK 损失为：

$$loss_{MASK}(\theta, \tau) = \frac{1}{\alpha |P|} \sum_{p \in P^{out}} CE(O_p, S_{p,\alpha}(\theta, \tau)) \tag{7}$$

最后，我们强制 f 是具有隐式几何正则化 (IGR)[6]的近似符号距离函数，即包含 Eikonal 正则化：

$$loss_E(\theta) = E_x(\| \bigtriangledown_x f(x; \theta) \| - 1)^2 \tag{8}$$

4 复现细节

4.1 与已有开源代码对比

本次复现中，对于一些复杂的坐标的变换由于个人在编写时总是出问题，所以最后采用了源码的坐标变换，包括在计算相机打向像素的光线方向，以及输入数据中相机的内参以及外参等数据的处理。具体代码如下：

```
def get_camera_params(uv, pose, intrinsics):
    if pose.shape[1] == 7:
        cam_loc = pose[:, 4:]
        R = quat_to_rot(pose[:,:4])
        p = torch.eye(4).repeat(pose.shape[0],1,1).cuda().float()
        p[:, :3, :3] = R
        p[:, :3, 3] = cam_loc
    else:
        cam_loc = pose[:, :3, 3]
        p = pose
    batch_size, num_samples, _ = uv.shape
    depth = torch.ones((batch_size, num_samples)).cuda()
    x_cam = uv[:, :, 0].view(batch_size, -1)
    y_cam = uv[:, :, 1].view(batch_size, -1)
    z_cam = depth.view(batch_size, -1)
    pixel_points_cam = lift(x_cam, y_cam, z_cam, intrinsics=intrinsics)
    pixel_points_cam = pixel_points_cam.permute(0, 2, 1)
    world_coords = torch.bmm(p, pixel_points_cam).permute(0, 2, 1)[:, :, :3]
    ray_dirs = world_coords - cam_loc[:, None, :]
    ray_dirs = F.normalize(ray_dirs, dim=2)

    return ray_dirs, cam_loc
def get_camera_for_plot(pose):
    if pose.shape[1] == 7:
        cam_loc = pose[:, 4:].detach()
        R = quat_to_rot(pose[:,:4].detach())
    else:
        cam_loc = pose[:, :3, 3]
        R = pose[:, :3, :3]
    cam_dir = R[:, :3, 2]
    return cam_loc, cam_dir

def lift(x, y, z, intrinsics):
    intrinsics = intrinsics.cuda()
    fx = intrinsics[:, 0, 0]
    fy = intrinsics[:, 1, 1]
    cx = intrinsics[:, 0, 2]
    cy = intrinsics[:, 1, 2]
    sk = intrinsics[:, 0, 1]
    x_lift = (x - cx.unsqueeze(-1) + cy.unsqueeze(-1)*sk.unsqueeze(-1)/fy.unsqueeze(-1) - sk.unsqueeze(-1)*y/fy.unsqueeze(-1)) / fx.unsqueeze(-1) * z
    y_lift = (y - cy.unsqueeze(-1)) / fy.unsqueeze(-1) * z
    return torch.stack((x_lift, y_lift, z, torch.ones_like(z).cuda()), dim=-1)
```

图 4: 数据处理代码

剩余部分如网络的搭建以及光线追踪等模块由本人编写，由于代码过长，下面列出几个核心代码相关代码如下：

```
        total_dims = [dim_input] + total_dims + [dim_output + fea_size]
        embed_fn, input_ch = get_embedder(multires)
        self.embed_fn = embed_fn
        total_dims[0] = input_ch
        self.num_layers = len(total_dims)
        self.skip_in = skip_in
        for l in range(0, self.num_layers - 1):
            if l + 1 in self.skip_in:
                out_dim = total_dims[l + 1] - total_dims[0]
            else:
                out_dim = total_dims[l + 1]
            lin = nn.Linear(total_dims[l], out_dim)
            if l == self.num_layers - 2:
                torch.nn.init.normal_(lin.weight, mean=np.sqrt(np.pi) / np.sqrt(total_dims[l]), std=0.0001)
                torch.nn.init.constant_(lin.bias, -0.6)
            elif multires > 0 and l == 0:
                torch.nn.init.constant_(lin.bias, 0.0)
                torch.nn.init.constant_(lin.weight[:, 3:], 0.0)
                torch.nn.init.normal_(lin.weight[:, :3], 0.0, np.sqrt(2) / np.sqrt(out_dim))
            elif multires > 0 and l in self.skip_in:
                torch.nn.init.constant_(lin.bias, 0.0)
                torch.nn.init.normal_(lin.weight, 0.0, np.sqrt(2) / np.sqrt(out_dim))
                torch.nn.init.constant_(lin.weight[:, -(total_dims[0] - 3):], 0.0)
            else:
                torch.nn.init.constant_(lin.bias, 0.0)
                torch.nn.init.normal_(lin.weight, 0.0, np.sqrt(2) / np.sqrt(out_dim))
            if weight_norm:
                lin = nn.utils.weight_norm(lin)

            setattr(self, "lin" + str(l), lin)
        self.softplus = nn.Softplus(beta=100)
```

图 5: 隐式神经网络

```
        total_dims = [dim_input + fea_size] + total_dims + [dim_output]
        self.embedview_fn = None
        if multires_view > 0:
            embedview_fn, input_ch = get_embedder(multires_view)
            self.embedview_fn = embedview_fn
            total_dims[0] += (input_ch - 3)
        self.num_layers = len(total_dims)
        for l in range(0, self.num_layers - 1):
            out_dim = total_dims[l + 1]
            lin = nn.Linear(total_dims[l], out_dim)
            if weight_norm:
                lin = nn.utils.weight_norm(lin)
            setattr(self, "lin" + str(l), lin)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
    def forward(self, points, normals, view_dirs, feature_vectors):
        if self.embedview_fn is not None:
            view_dirs = self.embedview_fn(view_dirs)
        if self.mode == 'idr':
            rendering_input = torch.cat([points, view_dirs, normals, feature_vectors], dim=-1)
        elif self.mode == 'no_view_dir':
            rendering_input = torch.cat([points, normals, feature_vectors], dim=-1)
        elif self.mode == 'no_normal':
            rendering_input = torch.cat([points, view_dirs, feature_vectors], dim=-1)
        x = rendering_input
        for l in range(0, self.num_layers - 1):
            lin = getattr(self, "lin" + str(l))
            x = lin(x)
            if l < self.num_layers - 2:
                x = self.relu(x)
        x = self.tanh(x)
        return x
```

图 6: 渲染网络

4.2 实验环境搭建

通过下面命令来下载实验数据

```
bash data/download_data.sh
```

训练网络

```
python training/exp_runner.py --conf ./confs/dtu_fixed_cameras.conf --scan_id
    SCAN_ID
```

生成三维模型

```
python evaluation/eval.py --conf ./confs/dtu_fixed_cameras.conf --scan_id
    SCAN_ID --checkpoint CHECKPOINT [--eval_rendering]
```

4.3 创新点

源论文得到的三维模型有时候会有一些不真实的部分，但是渲染网络可以在三维模型有显著错误的情况下得到接近准确的渲染图像，推测是因为渲染网络太强，从而让三维模型的错误对结果的影响降低，尝试通过减少渲染网络的参数量或者层数等参数来弱化渲染网络，从而得到更好的三维模型。

5 实验结果分析

在完成对一个数据的训练之后，我们得到关于该数据三维模型的零水平集表示 $f, f(x)$ 表示空间一点 x 距离三维模型的最小距离，所以我们可以通过 Marching Cubes 算法来得到三维模型的三角网格表示。如图 7所示

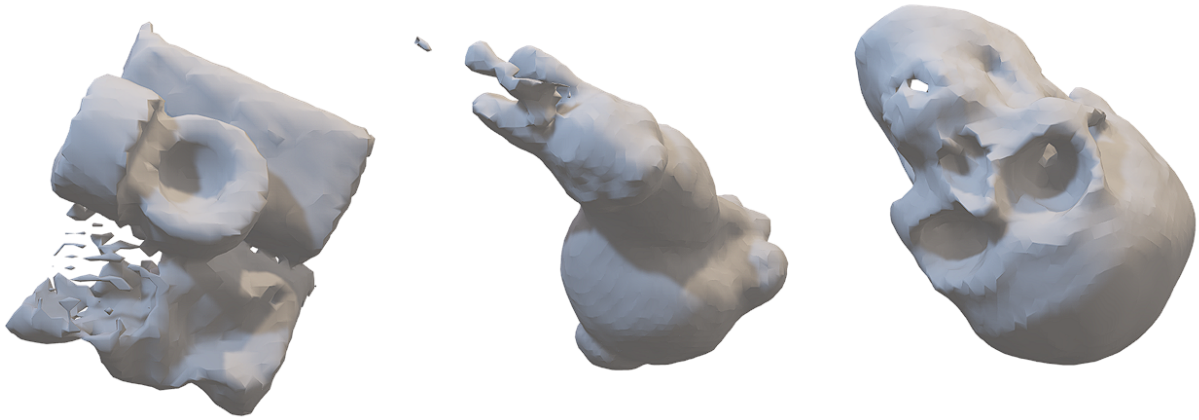

图 7: 实验结果示意

我们还可以对得到的三维模型通过渲染网络进行渲染，如图 8 所示，上半部分是渲染得到结果，下半部分是原图

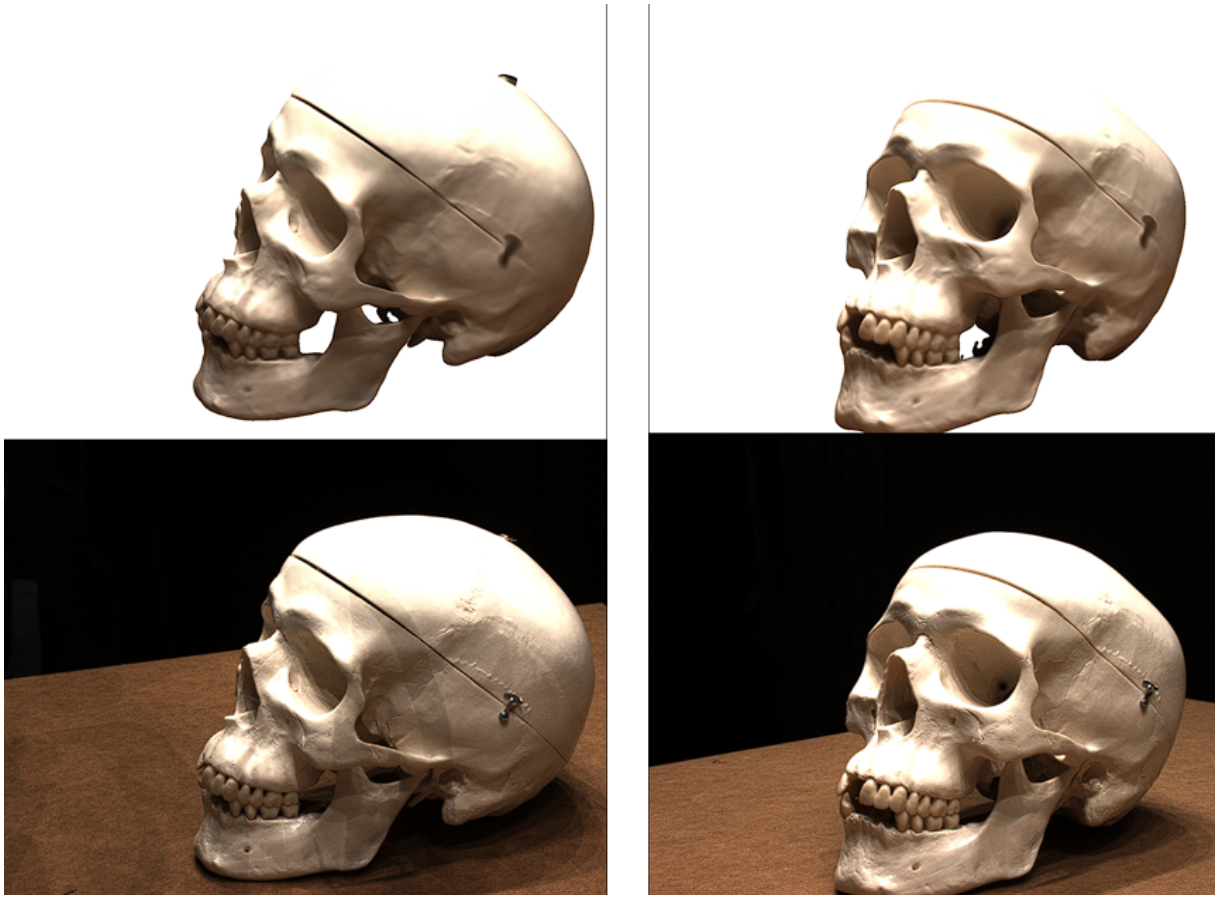

图 8: 实验结果示意

6 总结与展望

本次实验介绍了隐式可区分渲染器 (IDR)，这是一个端到端的神经系统，可以从蒙板的 2D 图像和噪声的相机初始化中学习 3D 几何图形、外观和相机。只考虑粗略的相机估计可以在真实场景中进行稳健的 3D 重建，在这些场景中，无法获得准确的相机信息。该方法的一个局限性是它需要合理的摄像机初始化，并且不能与随机摄像机初始化一起工作。未来可以考虑的方向是将 IDR 与直接从图像预测相机信息的神经网络相结合或者将表面光场进一步分解为材质和场景中的灯光

参考文献

[1] ShaohuiLiu, YindaZhang, SongyouPeng, et al. DIST: Rendering Deep Implicit Signed Distance Function with Differentiable Sphere Tracing[J]. arXiv: Computer Vision and Pattern Recognition, 2019.

[2] LIU S, SAITO S, CHEN W, et al. Learning to Infer Implicit Surfaces without 3D Supervision[J]. Neural Information Processing Systems, 2019.

[3] FURUKAWA Y, PONCE J. Accurate, Dense, and Robust Multi-View Stereopsis[J]. Computer Vision and Pattern Recognition, 2007.

[4] CURLESS B, LEVOY M. A volumetric method for building complex models from range images[C]//. 1996.

[5] SCHONBERGER J L, FRAHM J M. Structure-from-Motion Revisited[J]. Computer Vision and Pattern Recognition, 2016.

[6] GROPP A, YARIV L, HAIM N, et al. Implicit Geometric Regularization for Learning Shapes[J]. arXiv: Learning, 2020.