Differentiable Refraction-Tracing for Mesh Reconstruction of Transparent Objects

Jiahui Lyu

摘要

捕获透明物体的 3D 几何形状是一项具有挑战性的任务，通用的扫描和重建技术并不适合，因为无法处理镜面光传输现象。现有的最先进的专门为这项任务设计的方法，要么涉及一个复杂的设置来重建完整的折射光线路径，要么利用合成训练数据的数据驱动方法。在这两种情况下，重建的 3D 模型都存在过度平滑和细节缺失的问题。本文介绍了一种新颖的、高精度的固体透明物体的三维采集与重建方法。使用带有编码模式的静态背景，我们在摄像机视角光线和背景上的位置之间建立映射。然后使用折射光线路径的可微追踪直接优化物体的 3D 近似网格，同时确保轮廓一致性和平滑。大量的实验和比较表明，该方法具有较高的精度。

关键词：Computing methodologies；Computer graphics；Shape modeling；Mesh geometry models

1 引言

获取真实世界物体的三维几何形状一直是计算机图形学和计算机视觉领域长期存在的问题之一。现有的大多数三维获取方法，如激光扫描和多视图重建，都是基于物体是不透明的，其表面近似朗伯的假设。因此，这种方法不适用于由透明、折射材料制成的物体，因为它们与光相互作用的方式很复杂。现实世界里透明物体随处可见，针对透明物体研发出高效、高质量的算法有着重大意义。随着城市的发展，智慧城市的概念愈发深入人心。通过对整个城市进行三维建模，还原整个城市的几何信息和纹理信息，能对整个城市的统筹规划有着极其重要的作用。城市以玻璃材质为外墙的高楼大夏随处可见，如何高效透明重建，准确还原玻璃外墙信息，是一个迫在眉睫的问题。所以先从较小的透明物体开始，确立一套对透明物体重建行之有效的高效方法就具有重大意义。

2 相关工作

这篇论文的相关工作既包括了把透明前景提取出来的环境抠图，也包括了经典的透明表面重建，并且这篇论文是采用折射路径的可微追踪，因而也涉及到了光线-三角形的相关内容以及可微渲染。[1]。

2.1 Environment Matting

抠图是一种从图像中提取标量前景不透明度图的过程，通常称为 alpha 通道。环境抠图是 alpha 抠图的扩展，它还可以捕捉透明前景对象如何扭曲其背景，因此它可以在新的背景上合成。Zongker 等人的开创性工作是从一系列投影的水平和垂直条纹图案中提取环境哑光，假设每个像素只与矩形背景区域相关。为了提高环境抠图精度并更好地近似现实场景，Chuang 等人建议从周围环境中定位多个贡献源。其他工作提出了图像以外领域的解决方案，如小波域或频域。这篇论文的方法可以看作是环境抠图对透明对象重建任务的扩展，从某种意义上说，它逐步优化对象的重建形状，以便更好地匹配从多个视图捕获的环境抠图。

2.2 Transparent surface reconstruction

重建透明物体的表面几何是一个长期具有挑战性的问题。一些方法使用破坏性或侵入性技术，以获得详细的表面几何形状。非侵入式方法利用透明物体的折射特性，通过分析参考背景图像的扭曲来恢复其形状。从透明物体引起的光学畸变中恢复物体形状通常适用于单个折射率表面或参数模型，因为由多次反射和折射引起的光传输更难分析。除了固有折射，也有可能捕捉光传输的反射成分，并通过观察外部镜面高光估计形状几何。由于反射发生在最外层表面，因此有可能重建具有复杂几何形状和内部材质不均匀的物体。上述的这些方法，采集过程相当复杂，需要大量的手工工作来精确控制照明条件，以获得合理的结果。

最近，一些研究人员通过结合深度学习技术来解决透明物体重建的任务。Stets 等人使用编码器解码器架构来估计透明物体的单个输入图像的分割掩码、深度图和表面法线。Li 等人提出了一种不同的方法，其中渲染层嵌入到网络中，以考虑复杂的光传输行为。他们使用多视图图像实现了最先进的重建结果。由于难以获得足够数量的真实训练数据，这些数据驱动方法依赖于合成训练图像。虽然这些图像是使用高保真的照片级别渲染生成的，但真实图像和合成图像之间的差距仍然存在。具体来说，在合成图像上训练的网络很难泛化到真实的输入图像，因此它们很容易出现重建错误。这篇论文使用受控采集设置来捕获折射光路径，并使用直接的每个物体形状优化，这不需要由相似形状组成的训练集。

2.3 Light path triangulation

光路三角剖分是经典立体三角剖分的延伸，利用折射方向和表面法线之间的关系从光传输推断几何。Kutulakos 和 Steger 根据光线路径上的镜面反射和折射数量，对重建的可行性进行了理论分析。Tsai 等人揭示了深度正态模糊性。为了消除模糊性，Qian 等人提出了一种基于位置法向一致性的优化框架来恢复前后表面深度图。Wu 等人扩展了这种方法，并提出了第一个非侵入式方法来重建一般透明物体的完整形状。这篇论文利用可微渲染技术直接优化表面网格，能够恢复更细致的几何细节。

2.4 Differentiable rendering

许多工作利用可微渲染进行基于图像的 3D 网格重构。这些方法通常假设一个简化的图像形成模型，并仅限于朗伯场景。Li 等人介绍了一种通用可微射线示踪器，能够计算渲染图像上标量函数对任意参数的导数。这篇论文提出了直接基于射线-像素对应关系的折射损失，来反映了底层光传输的几何结构。光路的几何形状是由形状几何直接决定的，这是这篇论文解决的问题。

3 本文方法

3.1 本文方法概述

这篇论文首先使用灰色编码的背景图案折射穿过物体，获得物体的多个视图。然后使用空间雕刻的多视图轮廓集合重建物体的初始粗略形状。随后，基于环境抠图提取的视场光线与背景位置之间的对应关系，逐步优化粗糙模型，同时保持由轮廓提供的边界约束。为了在逐渐精细的尺度上恢复几何细节，这篇论文采用由粗到细的形状重新网格化的方式来优化损失函数。在每个阶段，通过最小化折射损失、轮廓损失和平滑损失三个项的组合来逐步更新重建的形状。

3.2 数据采集装置

为了获取穿过透明物体的折射光路信息，采用的数据采集装置如图 1所示：

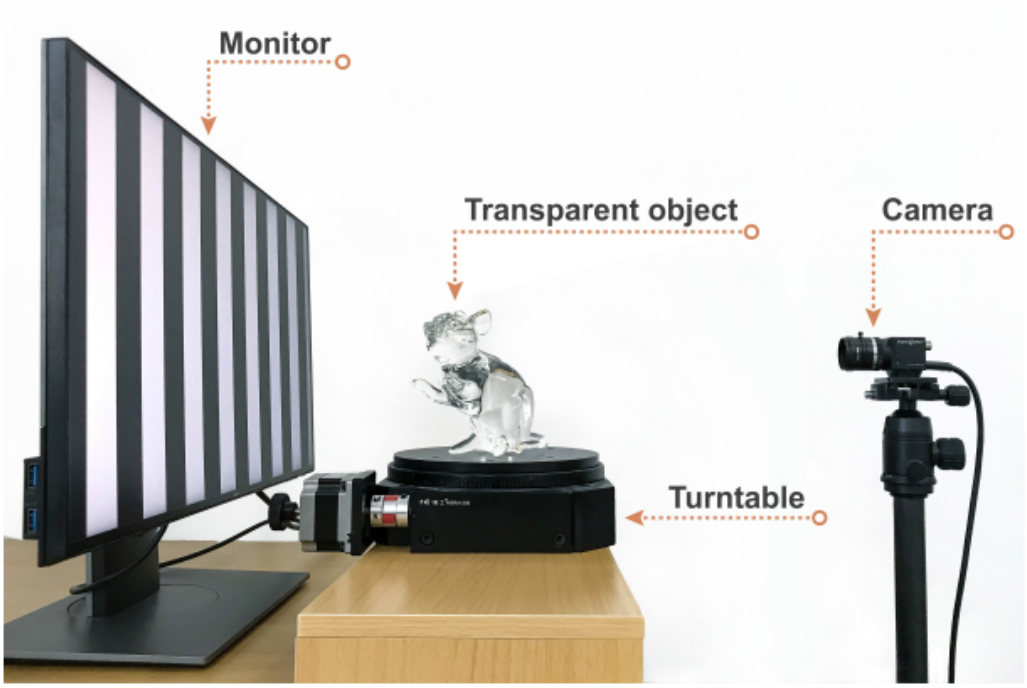

图 1: 数据采集装置

将待捕获的透明物体放置在转盘上，位于显示编码背景图案的液晶显示器和物体前面的静态摄像机之间。在采集开始之前，校准摄像机的内、外参数，以及显示器和转盘相对于摄像机的相对位置。为了捕捉一个物体，转盘被旋转到 72 个均匀采样的视角。在每个视角，灰色编码的背景图案显示在显示器上，同时提取轮廓和估计射线像素对应关系使用环境抠图。通过显示由 11 张垂直条纹图像和 11 张水平条纹图像组成的序列来生成灰色编码背景需要注意的是，为了避免环境光的影响，整个采集过程都在暗室中进行，使用背景显示器作为唯一光源。

3.3 损失函数定义

总损失由三部分组成，分别是折射损失、轮廓损失和平滑损失。

$$\ell = \alpha \ell_{refract} + \beta \ell_{silhouette} + \gamma \ell_{smooth} \tag{1}$$

α,β 和 γ 分别是折射损失、轮廓损失和平滑损失的权重系数。

3.3.1 折射损失

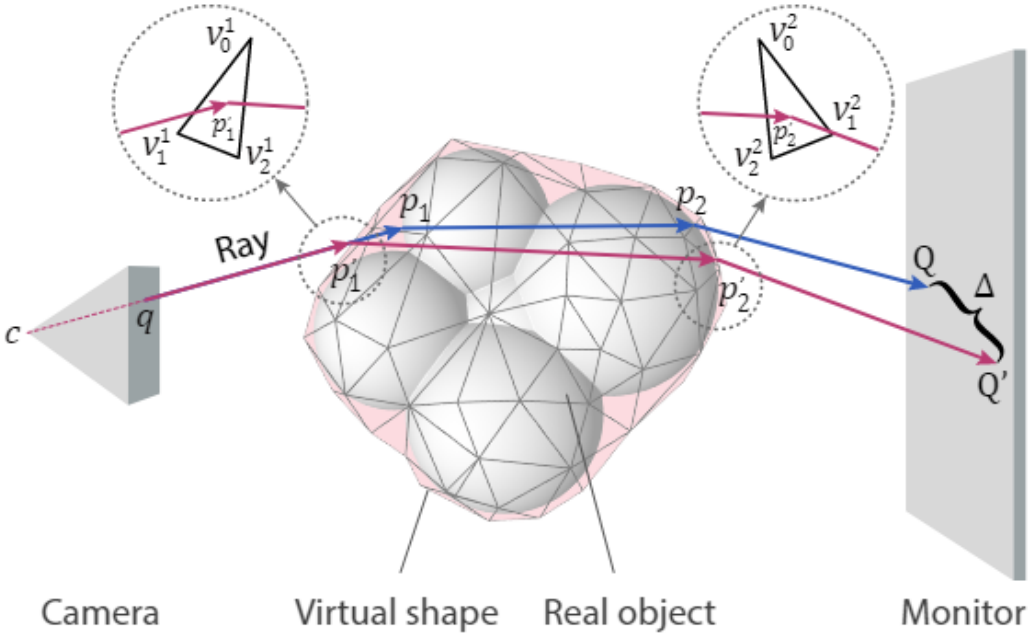

图 2: 折射损失示意图

经过图像像素 q 的模拟折射路径 (红色) 应到达观测到的背景点 Q，该背景点对应于真实射线路径 (蓝色) 与背景显示器的交点。粉色网格是优化的虚拟形状，初始化为可视球体。左上方和右上方的插图显示了单个模拟光线-像素对应关系的相关三角形和顶点。也就是说真实的折射光路径与显示器的交点是 Q，而模拟的折射光路径与显示器的交点是 Q'，折射损失就是两点之间的欧氏距离。

$$\ell_{refract} = \sum_{u=1}^{U}(\sum_{i \in I} ||Q - Q'||^2) \tag{2}$$

U 是捕捉视图的数量，I 是仅包含两次折射的折射光路径。

3.3.2 轮廓损失

图 3: 轮廓示意图

左边和中间的两张图分别代表真实的轮廓和模拟的轮廓，右边的图代表两者的重合示意图

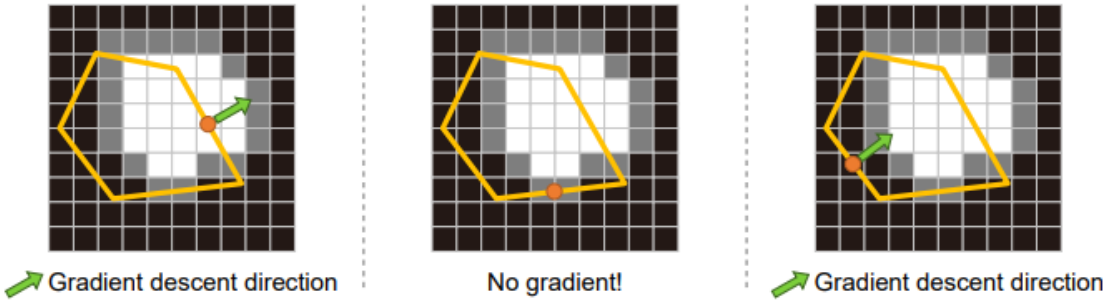

图 4: 轮廓损失梯度方向示意图

$$s_b = P_u \frac{v_1^2 + v_2^2}{2} \tag{3}$$

P_u 是视角 u 下的投影矩阵，b 是三维网格的一条边，v_1^2,v_2^2 是边的两个端点，得到的结果是投影边的中点。

$$-\bigtriangledown |\chi(s_b)| = \chi(s_b)||b||N_b \tag{4}$$

N_b 是投影边的外法向量的方向，$||b||$ 是投影边的长度。利用 $\chi(s_b)$ 来决定梯度的方向，如果中点落在真实轮廓外，值为-1；中点落在真实轮廓内，值为 1；中点落在真实轮廓上，值为 0.

$$\ell_{sihouette} = \sum_{u=1}^{U} \sum_{b \in B} |\chi(s_b)| \tag{5}$$

轮廓损失就是每条投影边人为定义损失的求和。

3.3.3 平滑损失

$$\ell_{smooth} = \sum_{e \in E} (-log(1 + < N_1^e, N_2^e >)) \tag{6}$$

平滑损失就是共用同一条边的相邻三角形法向量夹角的余弦值取对数。

4 复现细节

4.1 与已有开源代码对比

4.1.1 改进平滑损失函数

在阅读论文时，我对论文所定义的平滑损失函数感到疑惑。按照直观感受，共用同一条边的相邻三角形的法向量夹角余弦值应该趋向于 1，也就是法向量趋于一致。这应该是合理的，当用更小的三角形网格表示物体时，物体表面会更加平滑，那么相邻三角形也会更趋向于在同一平面内。而原论文的平滑损失会倾向于使相邻三角形的夹角余弦值逼近 0，使得相邻三角形相互垂直，使得三角形网格表示的物体更加尖锐。所以我把平滑损失改为如下形式：

$$\ell_{smooth} = \sum_{e \in E} (log(2 - < N_1^e, N_2^e >)) \tag{7}$$

```
def sm_loss(self):
    scene = self.scene
    #计算平滑损失
    dihedral_angle = scene.dihedral_angle() # cosine of angle [-1,1]
    #dihedral_angle = -torch.log(1+dihedral_angle)
    dihedral_angle = torch.log2(2 - dihedral_angle)
    sm_loss = dihedral_angle.sum()

    return sm_loss
```

图 5: 改进的平滑损失代码

在执行完代码后，我发现遇到了一个问题，就是结果对比于源代码的结果，在该平滑的地方的确是更平滑了，但是在一些不该平滑的地方也平滑了，造成了细节的缺失。我想这应该是原论文采用的由粗到细的重新网格化所造成的。在一开始，网格数量比较少，物体比较尖锐，相邻三角形的法向量夹角较大；在多轮优化，并重新网格化后，网格数量大大增加，物体比较平滑，相邻三角形的法向量夹角较小。也就是说，相邻三角形的法向量夹角应该是随着优化轮数的增加而逐渐变小的。所以平滑损失应该是一个随着优化轮数变化而变化的量。具体而言，可以将优化的轮数作为法向量夹角余弦值的次方参与运算。

```
def sm_loss(self,iteration,N_iteration):
    scene = self.scene
    #计算平滑损失
    dihedral_angle = scene.dihedral_angle() # cosine of angle [-1,1]
    #dihedral_angle = -torch.log(1+dihedral_angle)
    dihedral_angle = torch.log2(2 - dihedral_angle^(N_iteration-iteration))
    sm_loss = dihedral_angle.sum()

    return sm_loss
```

图 6: 更进一步的平滑损失代码

4.1.2 替换 Optix

原论文采用了 NVIDIA 所推出的光线追踪产品 Optix 来实现光线与三角形网格是否相交的判断，得到相交的光线与三角形网格序号。但在实现的过程中遇到了很大的问题。第一个问题是原论文是使用 Python 编写的，但 NVIDIA 的 Optix 是采用 C++ 编写的，这就涉及到了将 C++ 代码编译为一个 Python 库的混合编译问题。众所周知，混合编译是很容易出问题的，并且代码的迁移能力很差。后面我想在 jittor 框架下重新实现原论文，但混合编译使得这并不可能。第二个是 Optix 是并不开源的，里面的关键函数都无法得知其具体的实现，使得在 Optix 的光线追踪上面做出一些自己的改进是非常困难的。因为上述的原因，我选择开源的，并且同样是 Python 编写的 trimesh 库来代替 Optix 的功能。由于具体的实现是非常复杂的函数，所以我在这里只是简单的说明用 trimesh 分别实现了什么功能。

```python
def update_mesh(self, mesh_path):
    print(mesh_path)
    mesh = trimesh.load(mesh_path, process=False)
    assert mesh.is_watertight
    self.mesh = mesh
    self.vertices = torch.tensor(mesh.vertices, dtype=Float, device=device) #三角形的顶点坐标
    self.faces = torch.tensor(mesh.faces, dtype=torch.long, device=device) #构成一个三角形平面的对应顶点的索引
    self.triangles = self.vertices[self.faces] #[Fx3x3] #每个三角形的三个顶点的坐标


    self.ray_triang=ray_triangle.RayMeshIntersector(self.mesh)
    #self.ray_triang = ray_pyembree.RayMeshIntersector(self.mesh)

    self.init_VN()
    self.init_weightM()
    self.init_edge()  #权重初始化等等
```

图 7: 存储原始网格信息

创建一个 trimesh 中的 RayMeshIntersector 对象来存储原始的三角形网格信息，后面三角形网格信息的更新都会在这个对象中更改。

```python
def trimesh_intersect(self, ray:Ray):
    trimesh_o1 = ray.origin.to(torch.float32).to("cpu").detach().numpy()
    trimesh_d1 = ray.direction.to(torch.float32).to("cpu").detach().numpy()
    index_tri = np.zeros(trimesh_o1.shape[0], dtype=int)
    index_tri = self.ray_triang.intersects_first(trimesh_o1, trimesh_d1)  # 这个函数是得到光线第一次相交的三角形网格
    hitted_pai = np.full(index_tri.shape, True, dtype=bool)
    hitted_pai[index_tri == -1] = False
    faces_index = torch.tensor(index_tri, dtype=torch.long).to(device)
    hitted = torch.tensor(hitted_pai).to(device)
    return faces_index, hitted
```

图 8: 光线与三角形网格第一次相交

得到光线与三角形网格第一相交的光线以及对应的三角形网格的编号

```python
def update_verticex(self, vertices:torch.Tensor):
    #trimesh中ray_triang的网格的顶点坐标更新，但相应的面不变
    tri_vertices=vertices.to(torch.float32).to("cpu").detach().numpy()
    self.ray_triang.mesh.vertices=tri_vertices

    self.mesh.vertices = vertices.detach().cpu().numpy()
    self.vertices = vertices
    self.triangles = vertices[self.faces] #[Fx3x3]
    self.init_VN()
```

图 9: 更新存储的三角形网格的顶点信息

在算出来新的三角形网格的顶点信息后，需要在 trimesh 中的 RayMeshIntersector 对象来更新原来的三角形网格信息。

4.2 实验环境搭建

此次复现使用 Python 基于 pytorch 框架实现，主要用到 trimesh，numpy，OpenCV 等库。

在配置好环境后，按照 config.py 的提示，更改相应目录即可。

值得注意的是独立显卡必须是 NVIDIA 的，且显存要在 12GB 以上。

4.3 创新点

1) 改进了平滑损失函数，提升重建质量。

2) 用 trimesh 完美代替 Optix，降低编译难度，提升代码迁移能力，保留足够的可拓展空间。

5 实验结果分析

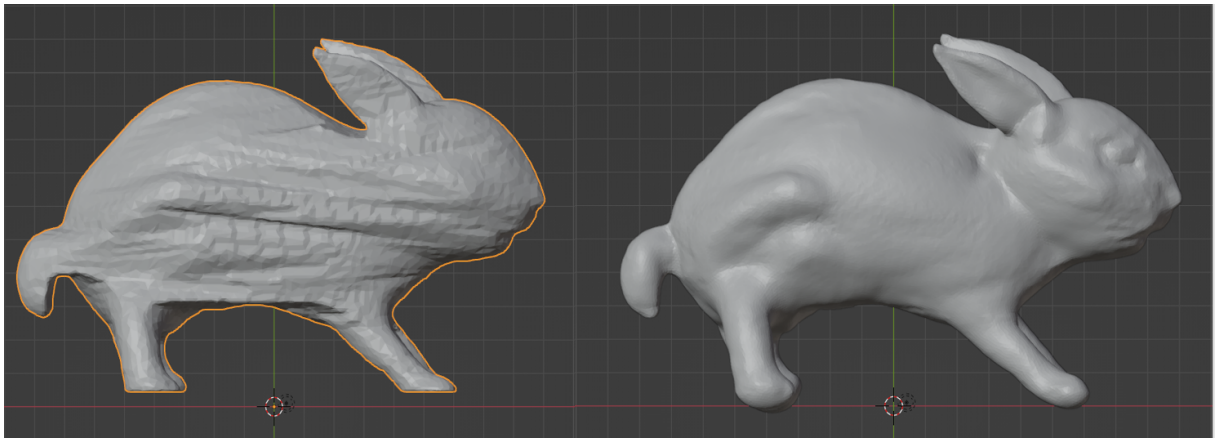

图 10: 源代码实验结果

左图是原始模型，右图是原论文代码所得到的实验结果，可以看到在还原透明物体几何形状的同时也保留了较好的细节信息。

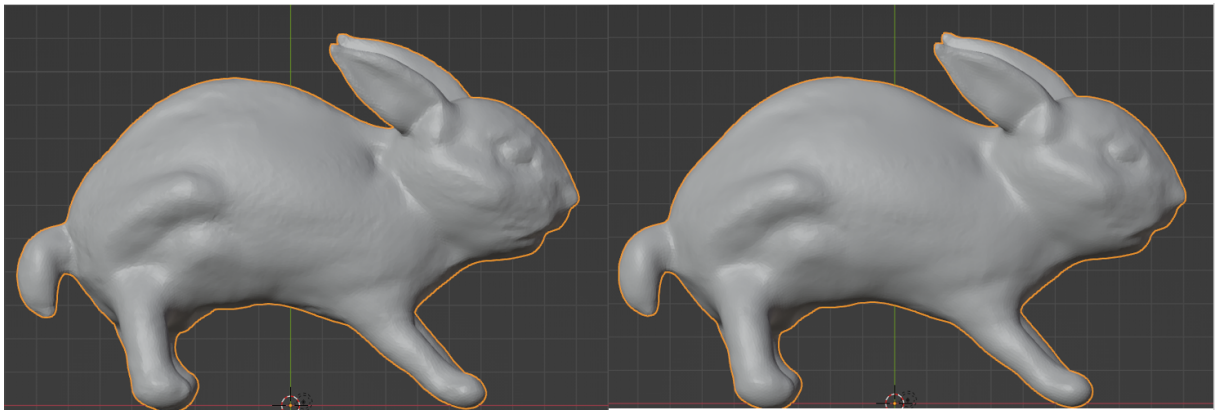

图 11: 初步改进平滑损失的实验结果

左图是源代码结果，右图是初步改进平滑损失所得到的实验结果，可以看到在该平滑的地方的确是更平滑了，但同时也有过度平滑的地方出现。

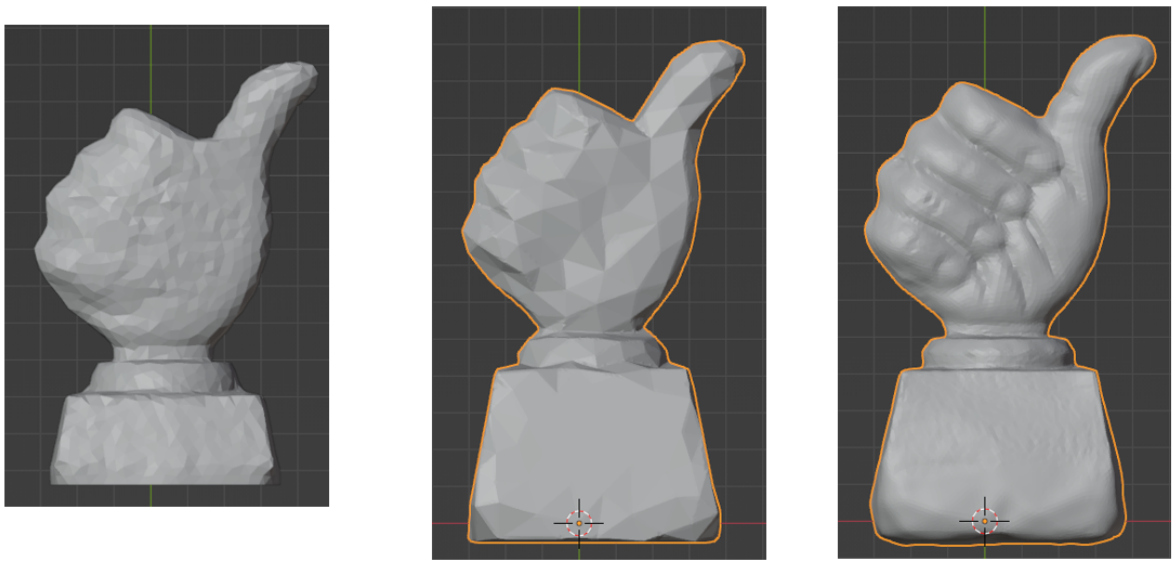

图 12: 用 trimesh 实现的实验结果

从左往右，重建质量越来越高，说明 trimesh 完美代替了 Optix。

6 总结与展望

传统的图形 pineline 还是以 C++ 为主的，但也逐步向着以 Python 为代表的深度学习未来靠近。作为一名研究图形学的人，还是要继往开来，既要掌握传统知识，高效快速地实现具体图形任务；也要向深度学习的未来学习，如何突破理论限制，更进一步地提升性能极限。

参考文献

[1] LIU S, LI T, CHEN W, et al. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning [J]. Cornell University - arXiv, 2019.