

# 基于光线追踪算法的真实感图像渲染器

姓名: 肖光煊

学号: 2018011271

班级: 计 81

计算机图形学基础

(春季, 2020)

清华大学

计算机科学与技术系

2020 年 6 月 21 日

## 摘要

基于路径跟踪 (Path Tracing) 和随机渐进式光子映射 (Stochastic Progressive Photon Mapping) 两种光线跟踪算法，实现了带有反射、折射、纹理、支持三角网格模型导入的光线追踪引擎。除基本功能外，基于牛顿迭代法实现了 Bezier 旋转曲面的解析求交，根据凸透镜成像原理实现了景深效果，通过 UV 展开和法线扰动算法实现了贴图纹理和凹凸纹理，通过像素超采样实现了抗锯齿效果，通过 RGB 三通道独立折射实现色散效果。在加速方面，算法上实现了轴对齐包围盒 (AABB) 和层次 KD-Tree 结构，硬件上采用了 OpenMP 加速，并实现了支持简单场景绘制的 CUDA 版本。

# 目 录

<b>1 项目架构</b>	<b>4</b>
<b>2 算法选型</b>	<b>4</b>
2.1 路径跟踪算法 . . . . .	4
2.2 随机渐进式光子映射 . . . . .	5
<b>3 光线求交加速</b>	<b>7</b>
3.1 算法加速 . . . . .	8
3.1.1 轴对齐包围盒 . . . . .	8
3.1.2 KD 树 . . . . .	9
3.2 硬件加速 . . . . .	10
3.2.1 CUDA . . . . .	10
3.2.2 OpenMP . . . . .	10
<b>4 参数曲面解析求交</b>	<b>11</b>
<b>5 实现效果</b>	<b>12</b>
5.1 景深 . . . . .	12
5.2 软阴影 . . . . .	13
5.3 抗锯齿 . . . . .	13
5.4 贴图纹理 . . . . .	13
5.5 凹凸纹理 . . . . .	14
5.6 重心插值 . . . . .	15
5.7 色散 . . . . .	16
<b>6 复杂场景</b>	<b>17</b>
6.1 教堂 . . . . .	17
6.2 客厅 . . . . .	17
<b>7 网格文件解析</b>	<b>18</b>

# 1 项目架构

本项目沿用 PA1 的文件结构和编译方法，使用提供的 run\_\*.sh 脚本即可复现实验结果。

- include/: 各种物体和算法的实现
- src/: 主函数代码和纹理、场景、图片加载和保存等函数的实现，mergeRGB.py 是用来实现色散，合并 RGB 通道需要的 python 脚本。
- texture/: 各种贴图和凹凸纹理。
- testcase/: 各种场景的描述文件。
- mesh/: 各种 3D obj 文件。
- output/: PT、RC 算法的输出目录。
- sppm\_output/: SPPM 算法的输出目录。

# 2 算法选型

本次实验我实现了两种光线追踪算法：路径跟踪（Path Tracing，下称 PT）和随机渐进式光子映射（Stochastic Progressive Photon Mapping，下称 SPPM），均取得了较好的渲染效果，代码均在 render.hpp 文件中。

## 2.1 路径跟踪算法

路径跟踪算法的思路较为直接：从相机向像素发出射线，经过场景中物体不断反射、折射，如果能够达到光源，则将光源发光强度乘上一路经过物体的反射率（衰减率），作为该光源对像素颜色的贡献；如果无法打到物体，则将背景光强乘上一路物体的反射率作为该像素的颜色；如果超过预先设定好的迭代深度，则将之前所有积累的光照强度算作该像素的颜色，并退出。由于存在漫反射面反射、折射面折射与反射导致光线分叉、部分镜面反射等不确定情况，因此采用蒙特卡洛模拟方法，对每一个像素多次发出射线，射线接触物体过程中的漫反射、折射面光线分叉、部分镜面反射的情况都根据物理定律计算相应概率，根据概率进行决策，将每次发出射线的色彩贡献取平均作为该像素点的颜色。路径跟踪算法能够实现较好的漫反射面、镜面和折射光线直接进入相机的成像效果，并且具有较好的 Color Bleeding 效果，如图1所示。但是由于路径追踪算法是单向的，因此从漫反射面很难追踪到通过折射和反射投射在上面的强光，即焦散现象。如图 2 所示，两个



图 1: 路径跟踪效果示意图

光源位于右上角和右下角，经过玻璃兔子之后应该在左下地面和左上墙面形成相应的光斑和影子，但是由于地面和墙面都是漫反射面，采样的时候只有很少比例能够取到这个折射方向，所以无法形成真实的焦散光斑，这一缺点在 SPPM 算法中得到了解决。

## 2.2 随机渐进式光子映射

随机渐进式光子映射则是一种双向算法，分为两个步骤。首先相机先按照类似路径追踪的思路，通过像素向场景中发射射线，直到射线接触到漫发射面为止，该接触点称之为“可见点”，这样图片上的每一个像素就对应于至多一个可见点，这些可见点记录其空间中的坐标，并组织成一棵 kd 树便于查找。第二个步骤也类似路径追踪，不过是从发光物表面随机选点选方



图 2: 路径跟踪的缺点在于不能呈现焦散现象

向作为大量光子发射的方向，随后光子给之前追踪到的“可见点”送光，如果光子确实在“可见点”的半径范围内，则将该点光通量和管辖半径进行更新。每次做完这两个步骤之后图片都可以在之前的基础上输出，我的代码中也实现了这个 checkpoint 功能，可以更方便看出 SPPM 算法渲染图像收敛的过程。

图3是和图2场景完全相同，只更改了渲染算法生成出来的图像。可以看到相比于 PT 算法在左下角的一片模糊的光斑，SPPM 渲染出了更为清晰真实的焦散光斑。



图 3: SPPM 算法生成相同场景的兔子，可见焦散效果

### 3 光线求交加速

为了能够渲染规模更大，物体更多的场景，原始的逐一枚举求交显然是不行的。我从算法和硬件两方面做了光线求交加速。

### 3.1 算法加速

#### 3.1.1 轴对齐包围盒

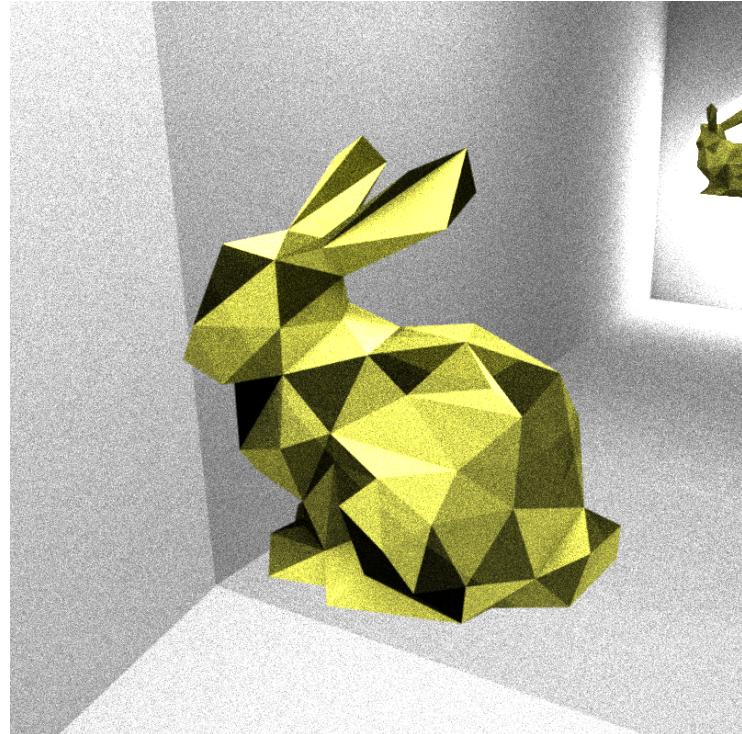


图 4: 轴对齐包围盒加速测试图

最易想到的就是轴对齐包围盒，代码为 Bound.hpp 中的 AABB 类，只需用一个与 x,y,z 轴对齐的长方体盒子将物体包住，求交的过程先跟包围盒求交，如果没有交点则一定不会与其中包围的物体相交，可以减少复杂网格的求交计算。轴对齐包围盒的求交算法也十分简单，只需计算三个维度光线射入和射出的 t，取三个  $t_{max}$  中的最小者和三个  $t_{min}$  中的最大者比较即可。图4中的兔子一共有 200 个三角面片，我将这 200 个面片用 AABB 包围起来。使用 PT 算法，spp=8 进行渲染。不使用 AABB 渲染时间为 28.337s，使用 AABB 渲染时间缩短到 13.334s。可见轴对齐包围盒确实能够大幅加速复杂面片的求交过程。

### 3.1.2 KD 树



图 5: KD 树加速测试图

在本项目中有两个地方需要用到 KD 树来进行查找加速，一个是光线与物体求交，一个是 SPPM 算法中可见点的查找，其代码分别位于 `object_kdtree.hpp` 和 `hit_kdtree.hpp` 中。我使用图5测试了 KD 树的加速效果，图中的兔子由 1000 个三角面片构成，使用算法为 PT，`spp=100`。在只使用 AABB 加速时渲染时间为 `3m32.993s`，KD 树和 AABB 同时使用渲染时间缩短到 `1m34.201s`，可见 KD 树的加速效果是十分显著的。由于本框架将场景中所有物体都组织成一个 Group，因此可以将 Group 内所有的物体也组织成一棵 KD-Tree，形成一种树套树的查找结构。通过这样组织，物体繁多的场景会有更大的加速比。

## 3.2 硬件加速

除了算法加速外，我还在硬件层面上做出了加速的尝试。

### 3.2.1 CUDA

通过自学，我初步了解了 CUDA 的写法，并且将 PT 算法的项目搬运到了 CUDA 代码上，整体代码位于 final\_cuda 文件夹内，可以通过 make output/sample.png 指令运行。但十分可惜由于我对 CUDA 的运作方式了解不深，我对内存的管理十分糟糕，并且 OOP 本身的设计模式也与 CUDA 的编程风格格格不入，因此运行速度仅比 CPU 单核快约 50 倍。

### 3.2.2 OpenMP

虽然说 OpenMP 不在加分范围内，但还是值得提一句。我在 64 核的服务器上使用 OpenMP 将 for 循环并行跑起来，加速还是很明显的。

## 4 参数曲面解析求交



图 6: 参数曲面解析求交效果图

我实现了教材上描述的牛顿法求 Bezier 曲面与射线交点的算法，代码位于 `revsurface.hpp` 中。首先我利用 AABB 包围盒与射线的交点当做迭代的初值，之后进入牛顿法迭代。之前我怎么修改阈值范围都会导致牛顿法迭代不收敛，我百思不得其解直到阅读了翁家翌学长的报告。报告中说 Bezier 曲线在  $t \in [0, 1]$  内十分平滑，但是在这个范围之外就极其陡峭。这启发了我，我将牛顿法迭代过程中每一步得到的值打印出来，发现果然是每当  $t$  越出 0 至 1 这个范围，迭代就再也不能收敛了。于是我每一步都加了一个检测：如果  $t$  小于 0，则将  $t$  设为 0；如果  $t$  大于 1，则将  $t$  设为 1。仅仅加了这一个约束，我的牛顿法有如神助，普遍只需迭代 4 次就可以将误差降到  $10^{-6}$  之下，相比之前的三角面片求交有了极大的速度上的提高，并且也

更能显示曲面的完整性。

图6所显示的是解析法求参数曲面与射线交点得到的花瓶，这种方法的另一好处是可以直接准确无误得到旋转曲面 UV 展开中的  $u$ ,  $v$  坐标，方便进行纹理映射。

## 5 实现效果

### 5.1 景深

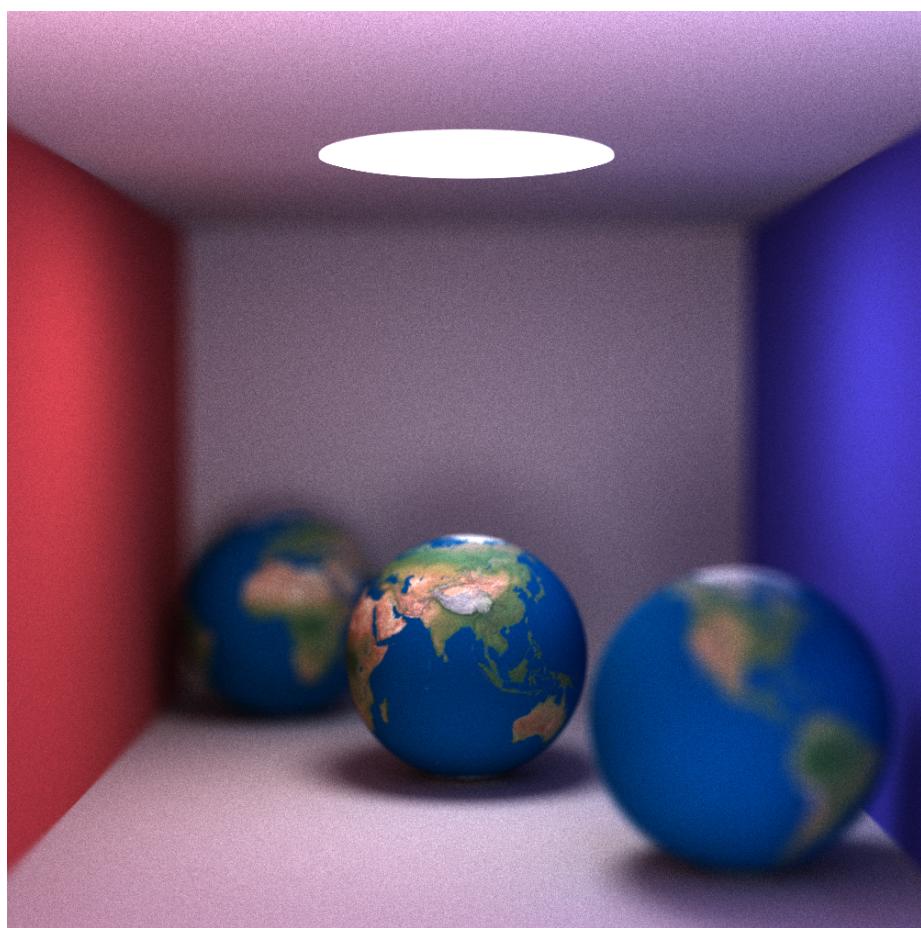


图 7: 景深效果图

景深的概念看似复杂，但实际上只要理解了现在的 PerspectiveCamera 与真实相机之间的差异就可以做出景深效果，代码在 camera.hpp 中。目前的 PerspectiveCamera 中照片上的每一个像素都是与场景中物体上的点一

一对对应的，这是小孔成像的原理，并且要求小孔的直径无穷小。在现实中不可能做到无限小的小孔，因为这样进光量也会无限小导致无法成像，因此现实的相机都是有光圈的，其直径为参数 `aperture`。

有了光圈之后，现实相机遇到的另一大问题就是无法成像。因为如果进光孔有范围，场景中任何一个点都将对应于照片上的一个范围，这会导致图像糊掉。因此人们会在进光孔上加一个凸透镜使光线汇聚，引入一个新的参数焦距 `focalLength`。这样所有焦平面上的物点都可以成像到相片上的一个点，形成清晰地成像效果。

然而不在焦平面上的物点就没有那么幸运了，必然对应于相片上的一块区域，因此真实相机会将焦平面前后一段范围以外的物体模糊掉，这段距离就称为景深。

理解了景深的形成原理之后，实现景深就不难了。只要对每一个像素先找到相机与像素连线与焦平面的交点，然后将相机中心在直径为 `aperture` 的圆盘上随机采样，最后再将焦平面上的交点与新的相机射线发射点连接作为相机射线，就可以得到景深效果了。

如图7所示，相机的焦平面位于中间地球的位置，因此只有中间地球较为清晰，其前后的地球都被虚化了。

## 5.2 软阴影

PT 算法和 SPPM 算法跑出来的图自带软阴影，因此这一效果也实现了。

## 5.3 抗锯齿

PT 算法和 SPPM 算法都需要对一个像素进行多次采样，我在每次采样的基础上引入了一个  $\pm 1$  的随机扰动，相当于之前只用某个像素中心点的颜色作为该像素的颜色，必然会导致不够平滑的现象。现在将像素周围  $[-1, 1] \times [-1, 1]$  区域内的平均颜色作为像素的颜色，这样就会做出柔和的物体边缘，减少锯齿。

## 5.4 贴图纹理

贴图纹理和凹凸纹理的本质都在于找一个  $f : R^3 \rightarrow R^2$  的 UV 展开函数，将三维的曲面映射到平面上，随后用载入的纹理图相应坐标的颜色表示此位置的颜色。因此对于每种曲面实现方法是不同的，我实现了球体，平面、旋转曲面的 UV 展开算法，三角网格的 UV 展开由于过于复杂且情况

繁多，因此使用载入 obj 文件中的 uv 坐标更好，不宜自行实现 UV 展开。之前图6、图7展现了球体和旋转曲面加贴图纹理的效果。

## 5.5 凹凸纹理

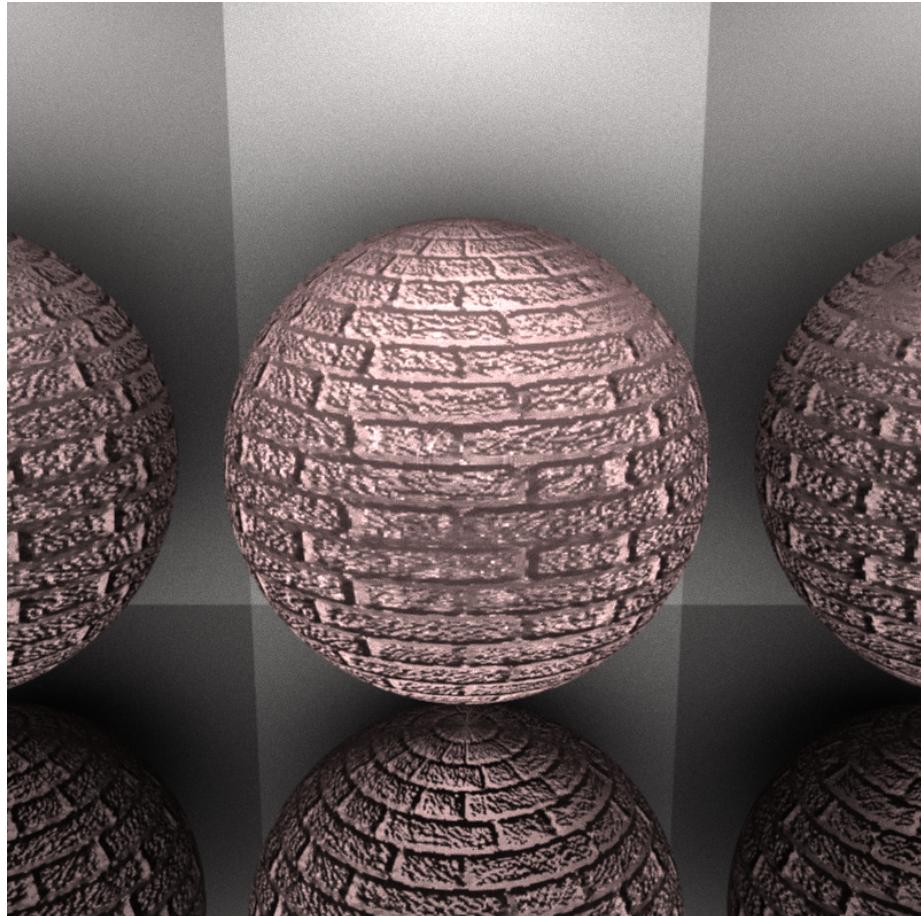


图 8: 凹凸纹理效果图

与贴图纹理类似，凹凸纹理实现也需要获得三维曲面的 UV 展开。不同的是得到 uv 坐标之后修改的不是物体表面的颜色而是法向量。载入的灰度图表示物体表面的起伏高度  $H$ 。设起伏高度方程为  $H(u, v)$ ，曲面参数方程为  $F(u, v)$ ，原法向量为  $n(u, v)$ ，则新曲面参数方程为：

$$S(u, v) = F(u, v) + H(u, v)n(u, v)$$

为了得到新的法向量，我们需要求新曲面对两个方向的偏导  $\frac{\partial S}{\partial u}$  和  $\frac{\partial S}{\partial v}$ ，取二者的叉积即为新的法向量。

$$\frac{\partial S}{\partial u} \approx \frac{\partial F}{\partial u} + \frac{\partial H}{\partial u} n(u, v)$$

$$\frac{\partial S}{\partial v} \approx \frac{\partial F}{\partial v} + \frac{\partial H}{\partial v} n(u, v)$$

而  $\frac{\partial H}{\partial u}$ 、 $\frac{\partial H}{\partial v}$  都可以线性逼近：

$$\frac{\partial H}{\partial u} \approx \frac{H(u + \Delta u, v) - H(u - \Delta u, v)}{2\Delta u}$$

$$\frac{\partial H}{\partial v} \approx \frac{H(u, v + \Delta v) - H(u, v - \Delta v)}{2\Delta v}$$

由此凹凸纹理即全部实现完毕，其代码与贴图纹理一样均在 `texture.hpp` 文件中，效果如图8所示。

## 5.6 重心插值

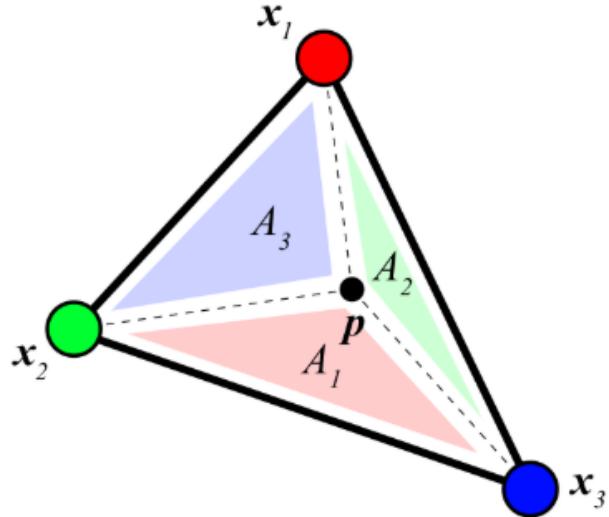


图 9：重心插值示意图

为了用较少的三角面片就获得柔和的曲面效果，以及获得连续的三角面片纹理，在三角面片表面必然要做重心插值。如图9所示，用顶点所对的三

角形内三角形面积比例表示该点的权重，对三个点的法向量/颜色作加权平均，即可得到连续变化的法向量/纹理颜色，代码位于 triangle.hpp 中。

## 5.7 色散



图 10: 色散示意图

我实现色散的思路较为简单，首先获得红、绿、蓝三种颜色的折射率，用这三种颜色跑三张图，然后用 mergeRGB.py 脚本将三张图片的 RGB 通道相应拼接，即可得到粗略的色散效果。如图10所示，场景中所有的光线和物体均为黑白二色，但是通过色散效果实现了色散效果，形成了真实钻石独特的彩色光辉。如果没有色散效果形成的钻石将十分单调乏味，而且难以看出前后层次。

## 6 复杂场景

### 6.1 教堂



图 11: Sibenik 大教堂与天使 Lucy

这张图我采用 SPPM 算法耗时 15 小时渲染得到，总共包含 50 多万个三角面片，墙和地面用了各种贴图纹理和凹凸纹理，做出古建筑的效果。Sibenik 教堂的 Mesh 来自 MIT 作业的样例，天使 Lucy 的 Mesh 来自 Stanford 公开的 3D Archive，还是很不错做出了那种显圣的感觉。但是初始半径设的有点太大了，图片效果略微有些脏，很遗憾没有时间提交一张减小初始半径的图片了，那张图片更加清晰，光线也更正常。

### 6.2 客厅

这张图我采用 SPPM 算法耗时 10 小时渲染得到，总共包含 58 万多个三角面片。还是初始半径设的有点大导致物体有些模糊，但是细节还是能看到的。左面的墙上挂了贝多芬头像、皇马队徽和清华校徽以及我的微信头像，分别为我最喜欢的音乐家、俱乐部和大学。右侧的画是二校门，表达了我这学期对校园的深深思念。中间茶几上摆放着这学期都没摸过的《计算机图形学》教科书，象征着虽然没能到课堂亲自聆听胡事民老师教诲但依然珍惜图形学课程，顽强完成大作业的乐观精神。



图 12: 客厅

## 7 网格文件解析

最后，由于我整个作业采用的是 PA1 的框架，所以读入场景以及材质信息都需要沿用 PA1 的编码格式。我搜索到 PA1 实际上是 MIT 计算机图形学的一项作业，此外那项作业又附送了一个 Java 代码，能够将网上常见的大型 obj 和 mtl 材质文件划分为只有一个材质的多个小 obj 文件，并自动生成场景材质描述信息。我发现他的代码不支持发光物体，而且对于有些含有四边形的 obj 文件会有 bug，我给他略加调试，使其鲁棒性增强。这样网上公开的 obj 文件都可以自动生成并导入到现有框架之下了。我想这个工具应该流传下去，因为我在最后一天才了解到许多同学因为不知道如何处理网上现成的 3D obj 文件而不会绘制复杂场景，使得他们原本不错的算法没有表现的空间。Java 代码为 FileUtil.java 和 ObjSplit.java，首先在目录下新建一个名为 tex 的文件夹，再在命令行先输入 javac ObjSplit.java 编译代码，随后输入 java ObjSplit ObjFile.obj MtlFile.mtl 即可将现成 ObjFile.obj 和 MtlFile.mtl 文件转化为多个单一材质的小 \*.obj，并在目录下生成相应的场景材质信息 s.txt。以后的助教可以在第一次作业就把这个工具交给同学们，让同学们自行导入公开的 Mesh，让这项作业越办越好！