大作业-基于光线追踪的全局光照场景渲染

1. OpenGL 环境

大作业采用 Freeglut+Glew 可编程管线，使用 CLion+CMake(MinGW)进行编译。

2. 程序使用说明

启动“程序包”文件夹下的 final.exe 文件，将会打开一个默认大小为 500 x 500 的 OpenGL 窗口，自动开始光线追踪渲染。由于本程序采用蒙特卡洛采样方法，需要将若干帧画面进行融合，因此渲染过程存在亮度由低到高、噪点由多变少的变化过程，平均帧率与机器的计算性能有关。共渲染 8 个迭代，每个迭代 512 帧，输出“iter [n] finished”时表示当前迭代结束，单击鼠标右键开始下一次迭代。随着迭代次数的增长，图像中的噪声逐渐减少。每次迭代结束时输出的平均帧率仅供参考，需要注意的是，若在渲染时移动窗口或更改窗口大小，会导致渲染暂停，使平均帧率的计算结果偏低。

场景中除 5 个墙面以及 2 个矩形面光源外，共有 7 个模型，分别为：1 个具有地球纹理的塑料球，1 个光滑金属球，1 个透明玻璃球，3 个塑料圆柱体，以及 1 个来自作业 2 的花瓶模型，均按照作业要求设置材质与纹理。使用鼠标左键分别点击以上 7 个模型，可以使其成为光源或恢复原状，进而观察到不同的场景渲染效果。

3. 基本思路

基于光线追踪的全局光照渲染过程与普通光照不同，普通光照的片元着色器仅需考虑当前片元与光源的关系，而光线追踪需要综合考虑当前像素与光源以及场景中其他片元的关系；普通光照的插值计算由程序管线自动完成，而光线追踪将一部分插值计算工作交给片元着色器，体现在计算光线与面片的交点中；此外，光线追踪对每个像素都需要递归求解每一条光路，基于以上三点，全局光照的渲染过程更为复杂，所需的计算量也更大。

在全局光照场景中，一个点的光照由两部分组成：

1. 该点的自发光
2. 来自其他方向的光在该点产生的累积光强

其中，其他方向的光是指该点法向半球中可以接受到的光线。因此，计算一个点的光照强度，需要进行一个半球积分，其简化形式如下：

$$L(p) = E(p) + \int L(q) \cdot cos(\theta)\, d\omega \tag{1}$$

其中p为需要计算光照强度的点，$L(p)$为p点的自发光，q为自p点出发，在方向ω上的光线命中的点，θ为方向ω与 p 点法矢量的夹角。在这种模型中，光强即为一种能量的积累。

3.1. 蒙特卡洛方法

显然，在代码中无法完成公式(1)中的积分，因此引入了一种基于采样的方法：蒙特卡洛方法。蒙特卡洛方法是一种用于估计积分值的方法，最直观的例子是通过“丢豆子”的方法估计圆周率 π 的值。

3.1.1. 积分估计原理

对于平面直角坐标系中的函数$f(x)$，用蒙特卡洛方法对区间$[a, b]$上的积分进行估计：

$$\int_a^b f(x)dx = \frac{b-a}{N}\sum_i f(x_i) = \frac{1}{N}\sum_i \frac{f(x_i)}{pdf(x_i)}$$

其中$pdf(\cdot)$为某一点处的概率密度，对于函数$f(x)$而言为：

$$pdf(x) = \frac{1}{b-a}$$

图 1 直观地显示了上述公式的几何意义，取样越密集，越接近理想积分值。

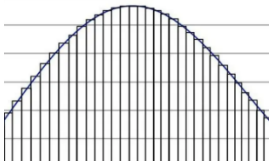


图 1 二维函数蒙特卡洛积分的几何意义

类似地，光照计算公式(1)可用蒙特卡洛方法表示为：

$$L(p) = E(p) + \frac{1}{N}\sum_{i}\frac{L(q)\cdot cos(\theta)}{pdf(\omega)}$$

对于半球积分而言，概率密度函数为：

$$pdf(\omega) = \frac{1}{2\pi}$$

其中2π即单位半球面面积，类似于二维积分中的区间长度$b-a$.

3.1.2. 蒙特卡洛方法在 OpenGL 管线中的实现

那么，如何利用 OpenGL 的渲染管线进行蒙特卡洛采样与混合呢？经过查阅资料发现，可以借助 OpenGL 的帧缓存与延迟渲染机制来完成。每一帧的处理思路为：

1. 绑定自定义帧缓存；
2. 进行当前帧的渲染，并与将上一帧步骤 3 产生的纹理进行混合；
3. 将自定义帧缓存中的像素数据读取到纹理中；
4. 重新绑定默认帧缓存；
5. 将这一帧步骤 3 中的纹理绘制到屏幕中进行显示。

因此，需要定义两组着色器管线，第一个着色器管线负责主要的光线追踪渲染工作（步骤 2），而第二个着色器管线只需要将渲染结果进行适当的处理之后绘制到屏幕中（步骤 5）。两组着色器管线可以共用一个顶点着色器，而使用不同的片元着色器。

3.2. 光线追踪

有了蒙特卡洛方法作为基础，复杂的积分计算已经转化为了随机取样与混合，此时需要进一步考虑如何进行光线追踪与渲染，即完成 3.1.2.小节步骤 2 中的渲染。

3.2.1. 原理

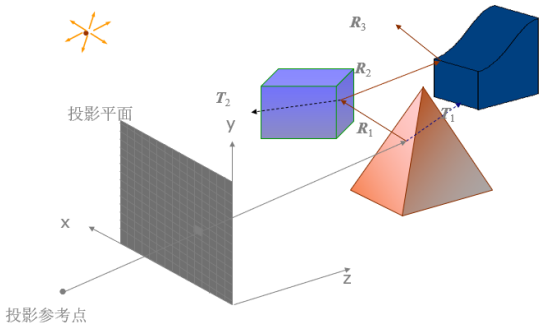


图 2 光线追踪原理图[1]

根据课上学到的知识，光线追踪的本质是对屏幕上的每个像素进行全局光照计算，上图

[1] 来自课程 PPT：《基础光照计算》

2 直观地展示了其原理。其中投影参考点即可看作视点，而投影平面则是覆盖整个屏幕的“画布”。与传统管线的光栅化绘制模式不同，此处只需要绘制一个正方形“画布”铺满整个屏幕，经过 OpenGL 管线的插值后，由片元着色器完成逐像素计算。对于每一个像素，初始光线由视点指向对应的“画布”上的位置，然后对该光线在每一次击中点的反射、折射进行追踪，叠加得到最终的颜色值。

3.2.2. 线性化递归

GLSL 不支持函数的递归调用，因此不能直接使用递归的方式进行光线追踪。为此，可以使用一种“线性化”的递归方式，即使用循环，每一次循环对应一次击中点的计算，使用数组保存每一次击中点的颜色数据，循环结束（未击中、击中光源或达到最大循环次数）后从数组的末尾向前遍历，得到最终积累的颜色值。

在使用线性化递归之后，实际上 3.1.中的蒙特卡洛方法仅对第一个击中点的光照进行了半球面的均匀混合，且对于每一帧渲染来说，每一层递归的取样次数仅为 1. 虽如此，最后可以发现这种方式已经取得了较好的渲染效果。一种较为合理且直观的解释是：第一个击中点才是视觉效果的主要承担者，这种线性化的递归方式相当于将随机取样的视觉效果通过多次随机采样集中到第一个击中点处，再由第一个击中点呈现给屏幕。当然，这种方式也存在一定的局限性，表现为渲染时场景中始终存在噪点，难以达到收敛。

4. 开始光线追踪

第 3.小节介绍了项目开发前的思考与准备，构建出了完整的思路架构，接下来就是一步步用代码实现。

4.1. 场景建模

4.1.1. 模型定义

考虑大作业要求以及代码实现的便利性，定义 4 种模型：四边形模型、球体模型、圆柱体模型以及自定义模型（即作业 2 中的旋转扫描模型）。选择四边形模型而非三角形模型的原因是：本次作业的光源采用面光源，5 个墙面以及 2 个面光源可直接用四边形模型表示，且四边形模型的相交计算也可以直接应用到自定义模型四边形面片的相交计算中。另外，球体模型和圆柱体模型的相交计算算法十分直观，对于反射、折射也具有良好的视觉效果。定义模型的代码位于 model.h 中。4 种模型的属性如下表所示：

表 1 模型属性

<table>
<tr><th>名称</th><th colspan="4">几何属性</th><th colspan="2">共有属性</th></tr>
<tr><td>四边形（Quad）</td><td colspan="2">顶点（vec3[4]）</td><td colspan="2">法矢量（vec3）</td><td rowspan="4">材质</td><td rowspan="4">纹理</td></tr>
<tr><td>球体（Sphere）</td><td colspan="2">球心（vec3）</td><td colspan="2">半径（float）</td></tr>
<tr><td>圆柱体（Cylinder）</td><td>下底面圆心（vec3）</td><td>底面半径（float）</td><td colspan="2">高（float）</td></tr>
<tr><td>自定义（Customized）</td><td colspan="4">面片（Patch[n]）</td></tr>
</table>

在模型定义基本完善的基础上，定义一个场景类（Scene）作为管理模型的上层类。该类的任务是在初始化时完成模型的导入与构建，在每一帧渲染时实现模型数据与着色器的交互，以及渲染逻辑的组织。定义场景类的代码位于 scene.h 中。

4.1.2. 表面材质

根据渲染的需要，定义表 2 所示的表面材质属性：

表 2 表面材质属性

条目	解释	范围
lighting	是否为光源	True/False
color	表面基础颜色/发光颜色	

specularRate	镜面反射概率 [1]	[0.0, 1.0]
specularTint	镜面反射光在表面基础颜色 [2] 与光照颜色之间的插值	[0.0, 1.0]
specularRoughness	镜面反射的粗糙度：反射光在标准方向上的扰动程度 [3]	[0.0, 1.0]
refractRate	折射概率	[0.0, 1.0]
refractTint	折射光在表面基础颜色与光照颜色之间的插值	[0.0, 1.0]
refractIndex	折射率	
refractRoughness	折射的粗糙度：折射光在标准方向上的扰动程度	[0.0, 1.0]

[1] 反射概率与折射概率的关系为：$0.0 \le specularRate + refractRate \le 1.0$，漫反射概率为：$1 - specularRate - refractRate$.

[2] 此处的基础颜色需要乘以光照的绝对强度，即：$color \cdot \|light\|$。

[3] 扰动程度的实现方式为从标准方向到随机方向按照粗糙度进行插值。

考虑到光线追踪为随机采样，镜面反射强度、折射强度、漫反射强度均以概率的形式表示，在着色器中根据以上概率随机采样下一条光线的方向。根据作业要求，程序中定义了一些需要用到的材质，包括：金属、塑料、墙面、玻璃、光滑木面、光滑陶面，各项属性均参考实际情况设置，详细内容请参考 material.cpp.

4.1.3. 数据交互

在计算每一个屏幕像素时需要获取场景中所有面片的数据，因此，需要以恰当的方式将所有模型的数据传入片元着色器中。四边形模型、球体模型、圆柱体模型的属性较为简单，可以直接通过 Uniform 变量传递；自定义模型的面片数据规模较大，无法直接通过 Uniform 变量传递，因此以 TextureBuffer 纹理的形式传入着色器，在着色器中通过 texelFetch 方法获取数据，具体过程在这里不做详细讨论，详细代码请参考 scene.cpp/tracer.frag.

4.2. 相交判断

相交判断是进行光线追踪的关键，用以判断光线与模型是否相交，同时返回交点坐标、交点与光线起点的距离、交点处法矢量等必要信息。交点与光线起点的距离用以进行遮挡判断。为简化程序设计，本程序仅支持球体模型的透明材质，因此只有球体模型进行相交判断时需要考虑与前后表面的两种相交情况。另外，在进行相交判断时需要剔除与自身相交的情况，比较简单的方法是判断交点到光线起点的距离是否为负数。详细代码请参考 tracer.frag.

4.2.1. 四边形模型相交判断思路

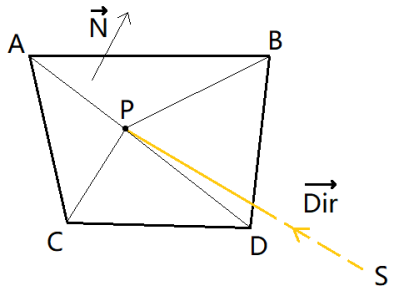


图 3 四边形相交判断

如图 3 所示，根据光线与四边形的位置关系可以得到方程组(2)，据此可以求出光线与四边形的交点P以及光线源点S到交点P的距离$|SP|$：

$$\begin{cases} \overrightarrow{AP} \cdot \vec{N} = 0 \text{（P 在平面内的条件）} \\ P = S + |SP| \cdot \overrightarrow{Dir} \text{（P 与光线的关系）} \end{cases} \tag{2}$$

根据向量之间的叉积与法矢量方向之间关系，当不等式组(3)全部成立时，交点 P 在四边形内部。

$$\begin{cases} (\overrightarrow{AC} \times \overrightarrow{AP}) \cdot \vec{N} > 0 \\ (\overrightarrow{AP} \times \overrightarrow{AB}) \cdot \vec{N} > 0 \\ (\overrightarrow{CD} \times \overrightarrow{CP}) \cdot \vec{N} > 0 \\ (\overrightarrow{BP} \times \overrightarrow{bd}) \cdot \vec{N} > 0 \end{cases} \tag{3}$$

4.2.2. 球体模型相交判断思路

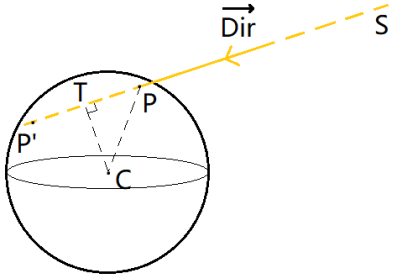


图 4 球体相交判断

如图 4 所示：根据向量$\overrightarrow{ST}$与向量$\overrightarrow{Dir}$之间的关系可以得到方程组(4)，据此求出$|CT|$与$|ST|$

$$\begin{cases} \overrightarrow{CT} \cdot \overrightarrow{Dir} = 0 \text{（垂直关系）} \\ T = S + |ST| \cdot \overrightarrow{Dir} \text{（T 与光线的关系）} \end{cases} \tag{4}$$

若$|ST| < Radius$说明光线与球体相交，进而可以根据勾股定理求出$|PT|$，得到P与P'坐标：

$$P = S + (|ST| - |PT|) \cdot \overrightarrow{Dir}$$
$$P = S + (|ST| + |PT|) \cdot \overrightarrow{Dir}$$

当$|SP| = 0$并且$|SP'| > 0$时，说明此时的光线为折射光线。

4.2.3. 圆柱体模型相交判断思路

圆柱体模型相交判断相对更为复杂，原因是需要同时考虑光线与圆柱体侧面和底面相交的情况。

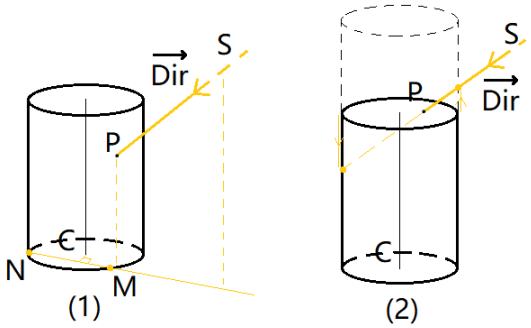


图 5 圆柱体相交判断

如图 5(1)所示，由于本程序中的圆柱体模型中轴均垂直于xz平面，可将光线向量投影至圆柱体底面所在平面后，计算与圆柱中轴的最近距离，求出与底面圆周的交点M, N，然后便可根据光线的方向向量$\overrightarrow{Dir}$计算与圆柱体侧面的两个交点。比较两个交点与圆柱体上底面、下底面所在高度的关系即可判断交点是否存在、是否在侧面或上下底面。图 5(2)展示了光线与圆柱体上底面相交的情况。具体计算过程此处不再赘述。

4.2.4. 自定义模型相交判断思路

自定义模型（旋转扫描模型）由大量四边形面片构成，最简单直接的相交判断思路是：暴力遍历所有四边形面片，按照四边形的相交判断方法判断是否存在交点，同时剔除背向面。在此基础上，由于已有圆柱体相交判断算法，一种简单的优化思路是使用一个圆柱体包围盒进行预计算，若存在交点再遍历所有面片，否则表明不会相交。这种方法可以在一定程度上

优化性能，但并不明显。根据课上所学知识以及查阅资料，可以使用层次包围盒（Bounding Volumn Hierarchy，BVH）的方法进行进一步优化。详细代码请参考 scene.cpp/tracer.frag.

4.2.4.1. BVH 方法思路

顾名思义，BVH 方法使用多重包围盒对模型面片进行细分，每个包围盒中可能包含若干个面片或更深层次的子包围盒，形成一棵搜索树。在进行相交判断时逐层搜索相交的包围盒直至到达包含面片信息的叶子结点，然后对叶子结点中的面片进行遍历求交。以二叉树为例，每次搜索都有机会剔除一半的节点，可以在很大程度上地优化性能。（由于同一层级的包围盒在空间上可能存在交叉，因此可能需要同时遍历两个子包围盒）

本程序采用 AABB（即长方体）包围盒，每个包围盒用体对角线上两个相对的顶点表示，这两个顶点分别位于坐标值最大和最小的位置。采用 AABB 盒构建 BVH 树的思路[1]为：

1. 导入自定义模型时用可排序数组保存面片数据；
2. 遍历数组中索引在$[l,r]$范围内的面片，分别计算x,y,z坐标的最大最小值，得到这部分面片的 AABB 包围盒；（初次遍历的范围为 $[0,size-1]$）
3. 如果这部分节点数目小于可以包含的最大面片数目，则构建叶子结点，停止递归，否则进行步骤 4，递归建树；
4. 选择当前 AABB 包围盒最长轴，按照四边形面片位置进行排序；
5. 分别对面片数组的前半部分$[l,mid]$和右半部分[mid+1,r]递归建树。

4.2.4.2. 线性化 BVH 树并传入着色器

同样地，由于 GLSL 不支持递归调用，无法使用递归的方式遍历 BVH 树，必须进行线性化处理。递归搜索 BVH 树的节点时，首先与根节点求交，然后分别与左右子树递归求交，因此，应该按照前序方式进行线性化。这部分算法与二叉树的前序遍历一致，因此不进行详细叙述。线性 BVH 节点以数组索引的形式储存左右节点指针。

线性 BVH 树可以按照 4.1.3.小节中介绍的方法传入片元着色器。在着色器中，同样通过与二叉树前序遍历类似的方式进行搜索。不同之处在于 GLSL 没有内置的栈数据结构，需要使用数组与索引代替栈的功能。

保证在本机相同的 GPU 环境下渲染（不降频），下表 3 展示了三种不同搜索方式的平均帧率，可以看到拥有大量面片的自定义模型确实是造成低帧率的原因，而采用 BVH 方法使性能有了质的提升。

表 3 不同搜索方式的平均帧率

方法	直接遍历	圆柱体包围盒	BVH	不渲染自定义模型
平均帧率/FPS	4.8	8.3	92	146

4.2.5. 遮挡剔除

遮挡剔除的实现方式十分简单，即每次都遍历场景中的所有模型，在遍历每一个模型或模型中的面片时，与之前已经击中的最近的距离进行比较，只有在距离更小时才可以判断为击中。

4.2.6. 纹理映射

光线追踪场景中无法通过管线自动插值计算纹理坐标，需要根据击中点信息自行计算。本程序支持四边形模型、球体模型、自定义模型和圆柱体模型的表面纹理映射，为简少代码工作量，圆柱体模型只在侧面实现纹理映射。具体而言，四边形模型需要通过击中点与四个顶点的关系计算纹理坐标，而另外三种模型均为旋转对称模型，根据击中点与旋转对称轴以及模型高度的关系可以方便地计算出归一化的纹理坐标。

4.3. 实现光线追踪

[1] 参考博客：https://blog.csdn.net/weixin_44176696/article/details/118655688

4.3.1. 法矢量半球采样

进行递归光线计算时，需要在当前击中点法矢量的周围随机生成下一条光线。本程序采用一种经过调整的随机半球采样方法，即越靠近法矢量，采样率越高。在不考虑蒙特卡洛概率密度变化的情况下，这种方式有利于降低对比度和噪声。具体代码请参考 tracer.frag.

4.3.2. 递归光线追踪

按照 3.2.2.小节中描述的“线性化”递归方式进行光线追踪，增加了部分算法细节的光线追踪伪代码如下：

Algorithm 1 片元着色器中的光线追踪算法

Input *rayDir*（光线方向：视点指向屏幕像素点，已经归一化），
rayStart（光线起点：即视点），
maxDepth（递归的最大层数，即最大循环次数）
Output *color*（颜色值）

```
for depth = 0 to maxDepth do
    if 当前光线没有击中任何模型 then
        color[depth] = vec3(0)
        退出循环
    end if

    color[depth] = 从当前击中点获取的颜色值 (1)
    cosine[depth] = 当前光线与击中点处法矢量的夹角余弦 [1]

    if 击中光源 then
        退出循环
    end if

    根据物体材质随机生成下一条光线：镜面反射/折射/漫反射 [2]
end for

for i = depth - 1 down to 0 do
    根据 color[depth]计算 color[depth - 1] [3]
end for

return color[0]
```

[1] 可以进行开方或平方等处理调整明暗对比度。

[2] 不同光线类型的概率参考 4.1.2.小节表面材质，方向计算参考 4.3.1.小节。

[3] 需要考虑每一层光线的类型，具体实现时单独使用一个数组保存类型。当光线为镜面反射光或折射光时，将$depth+1$层的颜色值与$depth$层的颜色值按照击中点材质的 Tint 属性进行插值混合，光线为漫反射光时直接使用 Lambert 算法计算。

片元着色器负责从纹理中取出上一帧该像素的颜色值，使用 Algorithm 1 计算出当前帧的颜色值除以 3.1.1.小节中的概率密度$\frac{1}{2\pi}$，然后进行混合。实际上，渲染第N帧时，上一帧的画面已经包含了前$N-1$帧的混合值，因此混合时上一帧与当前帧的比例为：

$$1.0-\frac{1}{N}:\frac{1}{N} \tag{5}$$

需要通过一个 Uniform 变量实时传入当前正在渲染的帧次N. 具体代码请参考 tracer.frag.

5. 其他处理

到此为止，基于光线追踪的全局光照场景渲染功能已经全部实现，图 6(2)展示了最终的渲染效果。本小节为一些细节方面的内容。

5.1. 旋转扫描模型导入

根据作业要求，在场景中需要添加一个来自作业 2 的旋转扫描模型。关于模型的储存有多种方式可以选择，第一种直观的方式就是直接以原始面片数据的形式储存，本程序最终选择采用这种方式；另一种方式是储存模型的旋转轴和扫描线样本点，并据此设计击中判断算法，而这种算法难以实现，始终没有找到合适的思路，因此并未选择这种方式。

5.2. 伽马校正

关于伽马校正的原理此处不详细叙述，其基本目的是修正颜色值与实际视觉效果的非线性差别，可以起到调整色差、对比度的作用。伽马校正的基本公式如下，γ值一般取$\frac{1}{2.2}$:

$$Output = Input^{\gamma}$$

本程序在 3.1.2.小节中提到的第二个着色器管线中进行伽马校正。与此同时，为防止在递归光线追踪时颜色值曝光过度，在 Algorithm 1 中的语句(1)处进行反伽马校正。使用这种方式的理由是：通过表面材质设置的颜色值是人眼看到的颜色值，通过反伽马校正转换为储存与计算的 RGB 值，最后再通过伽马校正转换为人眼看到的色调绘制到屏幕上。图 6 展示了是否进行伽马校正的区别，可以明显观察到进行了伽马校正的渲染场景对比度更小，视觉效果更加合理。但这种求幂的方式也存在一个缺点，即加重了图像暗处的噪声。

(1)无伽马校正　　(2)有伽马校正

图 6 有无伽马校正效果对比

5.3. 反走样

本程序在片元着色器中通过自定义多重采样方式实现抗锯齿，思路为对屏幕像素值加上适当的随机偏移量。偏移量不宜过大，否则容易出现模糊，也不宜过小，否则达不到反走样效果。经过尝试确定的最佳偏移是在 1 个像素单位圆内随机分布，具体调整方法如下：

$$\begin{cases} pixel.x = pixel.x + \dfrac{2.0 \cdot (d \cdot sin(\theta) - 0.5)}{width} \\ pixel.y = pixel.y + \dfrac{2.0 \cdot (\mathrm{d} \cdot \cos(\theta) - 0.5)}{height} \end{cases}$$

式中$pixel$为当前像素的坐标，$width$与$height$分别为渲染画面的宽度与高度（非屏幕宽高），d与θ分别为范围在$[0,1]$的随机半径与范围在$[0,2\pi]$的随机角度。

图 7 展示了是否使用以上反走样方法的区别，可以明显观察到使用了反走样的渲染效果更加平滑。与此同时，这种多重采样方法也可以在一定程度上降低图像噪声。

(1)无反走样　　(2)有反走样

图 7 有无反走样效果对比

5.4. 模型在光源与非光源之间的切换

这部分内容即第 2.小节中提到的鼠标左键操作。其实现方式与着色器中的击中判断算法基本一致，但由于此处无需达到渲染时的高精确度，并且只需要返回是否击中，因此对算法进行了一些简化。点击鼠标左键时，根据点击的位置计算出相对于渲染场景的像素坐标，然后按照相同的方式进行击中判断，切换击中的最近的模型材质的光源属性，并重新开始场景渲染。图 7 即将场景中的玻璃球切换为光源时的局部截图。

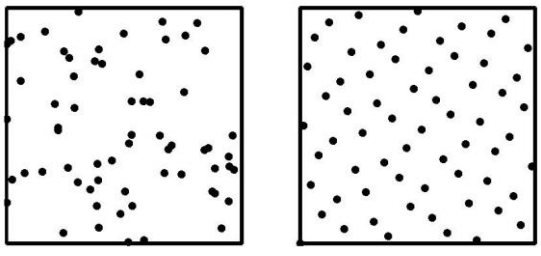
5.5. 关于 GLSL 随机数

由于涉及到随机采样，因此需要在片元着色器中使用随机数。这种随机数的实现方式必须考虑两重随机性：对不同像素随机、对不同帧次随机，因此需要根据像素位置以及帧次设置随机种子以及合适的哈希函数。本程序中的随机数生成方法参考网络资料[1]中的代码，并根据实际使用情况进行了一些简化。其大致原理是利用并放大浮点数在计算中的不精确性。

5.6. 降噪

5.6.1. 低差异序列加速收敛

造成画面中存在噪点的一个原因是伪随机数的取样不均匀。查阅资料[2]发现，存在一种低差异序列，可以通过一些数学方式使得取样更为均匀，进而在一定程度上减少噪点。图 8 直观地显示了伪随机取样与低差异序列取样的区别，可以看到，低差异序列生成的取样点比较均匀地覆盖了整个取样面。



(1)伪随机　　(2)低差异序列

图 8 两种取样方式对比

Sobol 序列是一种简单的低差异序列，此处不详细讨论其数学原理，其大致思路是：为

[1] 参考博客：https://blog.csdn.net/weixin_44176696/article/details/119044396
[2] 参考博客：https://blog.csdn.net/weixin_44176696/article/details/119988866

每一维度设置一组生成矩阵，按照特殊的按位异或方式计算每一维度第 n 帧的取样点。由于随机半球采样需要 2 个维度的随机数，因此需要 2 个维度的 Sobol 生成矩阵，代码参考网络资料实现。

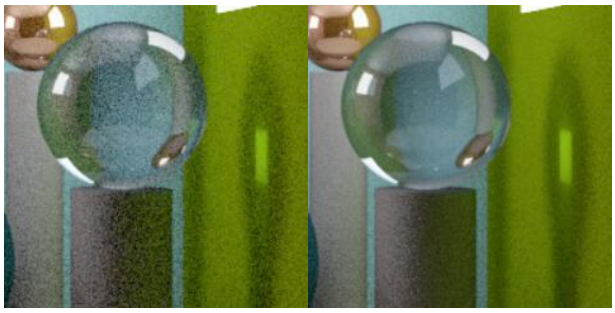
图 9 展示了是否使用 Sobol 序列在 1024 帧时的局部渲染效果，可以看出使用 Sobol 序列时的画面噪点更少；但是这种方式在场景中光照较暗的地方并不能明显减少噪点。

(1)不使用 Sobol　　(2)使用 Sobol

图 9 是否使用 Sobol 序列效果对比

5.6.2. 调整混合方式

4.3.2.小节式(5)的混合比例看似没有问题，实际上随着帧次N的增大，第N帧的混合比例越来越小，在进行浮点数计算时的误差也会越来越大。因此，可以将渲染过程分为I个迭代，每个迭代混合M帧，最后将I个迭代的渲染结果混合。由于每个迭代混合帧数减小，计算误差减小，在多个迭代混合时进一步将误差平均，可以有效地降低图像噪声。图 10 展示了是否使用迭代混合方法的局部渲染结果：



(1)常规混合　　(2)迭代混合

图 10 是否使用迭代混合方法降噪效果对比

为方便查看不同迭代的降噪效果以及平均帧率，需要通过鼠标点击开始下一次迭代，具体操作方法请参考第 2.小节。