计算机图形学大作业 A1 报告

赵泳豪 朱浩泽

摘要

本文基于计算机图形学课程提供的渲染框架，完成了对渲染框架的改进以及功能扩展，本文的主要贡献集中于针对核心渲染算法的改进与功能扩展，主要工作包括，实现 Whitted-style 光线追踪算法、实现 KdTree 加速光线-物体求交、实现对光源采样的路径追踪算法、实现基于光子映射的路径追踪算法、实现 KdTree 加速光子映射算法、实现基于菲涅尔方程的绝缘体材质渲染。

1. 引言

本次实验选取大作业 A1 作为最终目标，根据作业要求，对课程给定的框架进行了一定程度上的改进与功能扩展，并且最终有效的在原框架的基础上提升了渲染质量与渲染性能，本文的主要工作有：

- 实现 Whitted-style 光线追踪算法
- 实现 KdTree 加速光线-物体求交
- 实现对光源采样的路径追踪算法
- 实现基于光子映射的路径追踪算法
- 实现 KdTree 加速光子映射算法
- 实现基于菲涅尔方程的绝缘体材质渲染

接下来将对本文所做的工作进行详细介绍与总结。

2. 相关工作

光线跟踪 (Ray tracing)，又称为光迹追踪或光线追迹，来自于几何光学的一项通用技术，它通过跟踪与光学表面发生交互作用的光线从而得到光线经过路径的模型。在图形学领域光线追踪用于表示三维计算机图形学中的特殊渲染算法，跟踪从眼睛发出的光线而不是光源发出的光线，通过这样一项技术生成编排好的场景的数学模型显现出来。这种方法有更好的光学效果，例如对于反射与折射有更准确的模拟效果，并且效率非常高，所以当追求这样高质量结果时候经常使用这种方法。

相对于传统光栅化渲染算法，光线追踪算法渲染得到的结果往往更加接近真实情景，在游戏领域，光线追踪常作为高画质的代名词。学界公认此算法最早由 Turner Whitted 在 1980 年提出 [4]，也就是最基本的 Whitted-style 光线追踪算法，本次实验将从 Whitted-style 光线追踪算法出发，对光线追踪算法进行探究与实验。

相对于光栅化渲染算法，光线追踪算法最大的劣势就是性能较差，在一些情景下无法做到实时的效果。NVIDIA 新一代 RTX20 系显卡带来了一种全新的 GPU 光线追踪技术，该技术利用 GPU 的高并行计算的特性，使得光线追踪算法实时渲染游戏画面变的可行，使得游戏画质进一步得到了提升，在本文中我们也尝试利用 OpenMP 多线程、KdTree 等技术对光线追踪算法进行加速。

3. 方法实现

3.1. Whitted-style 光线追踪

3.1.1 背景知识

我们首先了解 Whitted-style 光线追踪算法的流程，由于 Whitted-style 光线追踪算法相对简单，根据流程以及框架中提供的相应函数，就可以按步骤完成 Whitted-style 光线追踪算法。

Whitted-style 光线追踪算法中忽略了漫反射间接光照问题，这也是该算法比较简单的原因之一，

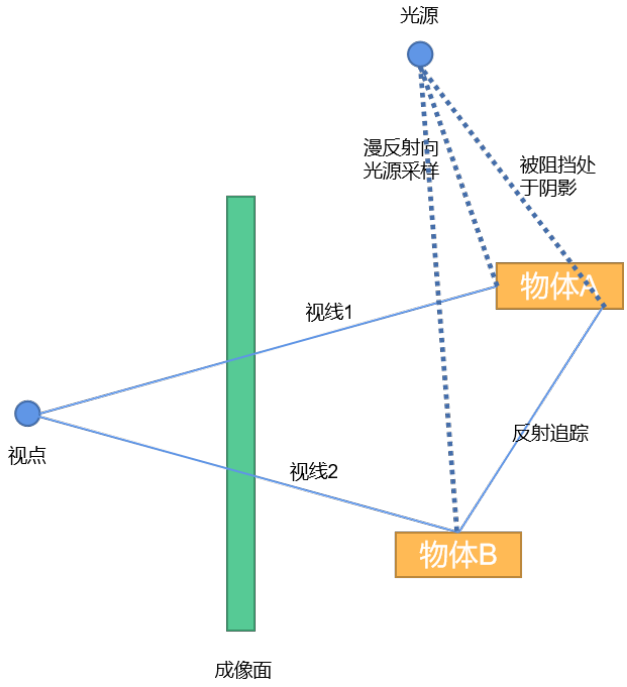

图 1. Whitted-style 光线追踪意图

Whitted-style 光线追踪中对于正常的光线，由光源发出，经过反射和折射过程，最终汇入人眼，但从光源发出的光线，能否最终射入人眼是个不确定的问题，因此实际 Whitted-style 光线追踪算法的跟踪方向与光传播的方向是相反的，更准确的来说是视线跟踪。

Whitted-style 光线跟踪算法图1，由视点与成像面上的像素构成的一条射线，射线与物体相交后，若为漫反射表明停止追踪，并向光源采样确定阴影和直接光照，反之若其为反射面或者折射面则在其反射与折射方向上进行跟踪，在理想情况下，光线可以在物体之间进行无限的反射和折射，但是在实际的算法进行过程中，我们不可能进行无穷的光线跟踪，因而需要给出一些跟踪的终止条件。在算法应用的意义上，可以有以下几种终止条件：

- 该光线未碰到任何物体
- 该光线碰到了漫反射表明
- 光线反射或折射次数即跟踪深度大于一定值

3.1.2 实现

Whitted-style 光线追踪算法的实现比较简单，因为框架中已经提供了基础的 Ray Casting 算法[1]，该算法其实就是光线追踪算法的第一步，该算法投射出去的视线与物体相交，无论相交表明的类型 (漫反射、反射、折射) 如何，均不进行追踪，直接向光源采样确定其阴影与直接光照，我们要做的工作就是在 Ray Casting 算法的基础上，对物体表明类型进行判断，若为反射 or 折射类型则继续递归在其反射与折射方向上进行跟踪即可。

图2中给出了两种算法的对比效果，可以发现对于反射类型的球体 Ray Cast 算法由于没有考虑反射，所以渲染出的画面并没有体现出反射效果，而 Ray Tracing 算法考虑到反射效果，渲染结果可以体现镜面反射。

Whitted-style 光线追踪算法的一个明显的弊端就是，虽然能渲染出镜面反射的效果，但该算法没有考虑漫反射，追踪遇到漫反射表明会直接终止，所以渲染出的画面很 “假”，比如由于漫反射，现实中不会出现结果图中的硬阴影。

3.2. KdTree 加速光线-物体求交

3.2.1 背景知识

Whitted-style 光线追踪算法中极其关键的一步操作，就是计算路径 (光线或视线) 与场景中物体是否相交并且给出第一个相交的物体，在框架中计算相交使用的是遍历的方式，即遍历场景内所有物体找出最近的相交物体，在框架中提供的场景中，由于物体较少，因此遍历算法的弊端没有很明显的体现，但试想对于极复杂的场景，场景中可能存在数千万个物体，这时候遍历算法的性能会极其低效，严重限制了整个光线追踪算法的性能。

3.2.2 实现

针对性能问题，一个普遍的接近方案便是使用加速数据结构，本次实验中我们采用 KdTree[3]对光线-物体求交算法进行加速，KdTree 的思想与我们熟知的二叉树类似，KdTree 根据物体的包围和 (能

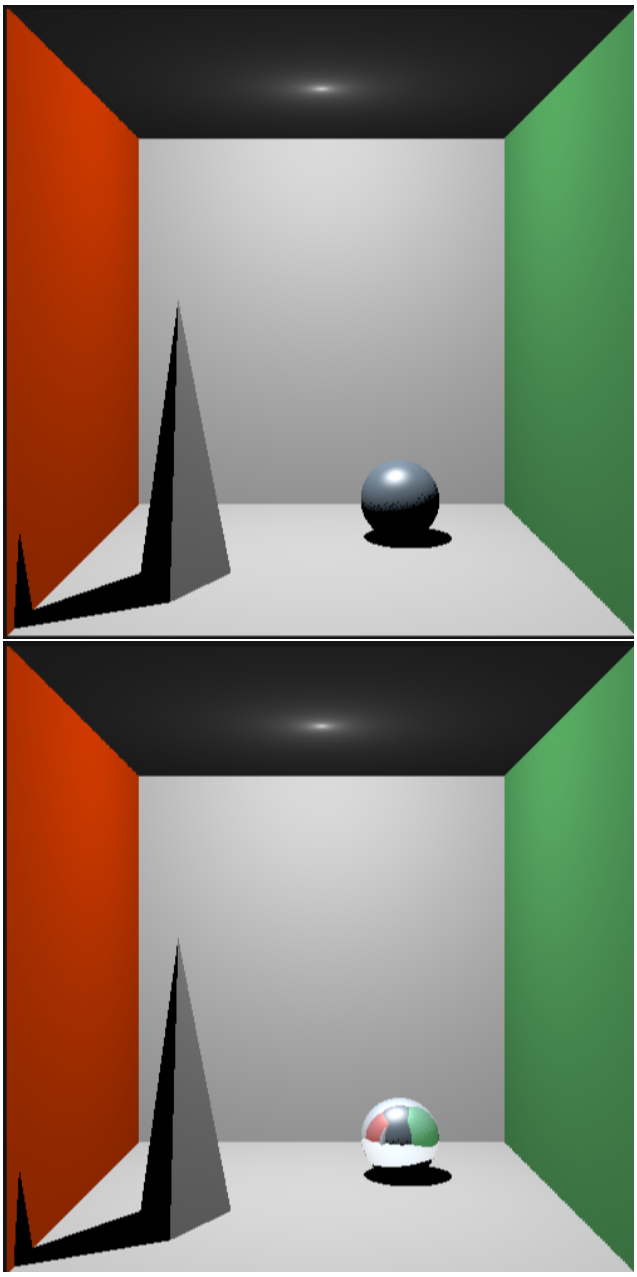

图 2. Ray Traceing(下图) 与 Ray Casting(上图) 对比

包围整个物体的最小的立方体) 的空间位置，将所有物体进行分割。

图3中给出了二维平面 KdTree 的示意图，三维空间同理，将物体根据空间位置进行分割，光线只有与空间范围 (图中的大方框) 相交，才能与其中的物体相交，若不相交则肯定与其中物体不相交，总结来说还是使用的是二分查找的思想，只是将其应用于三维空间。

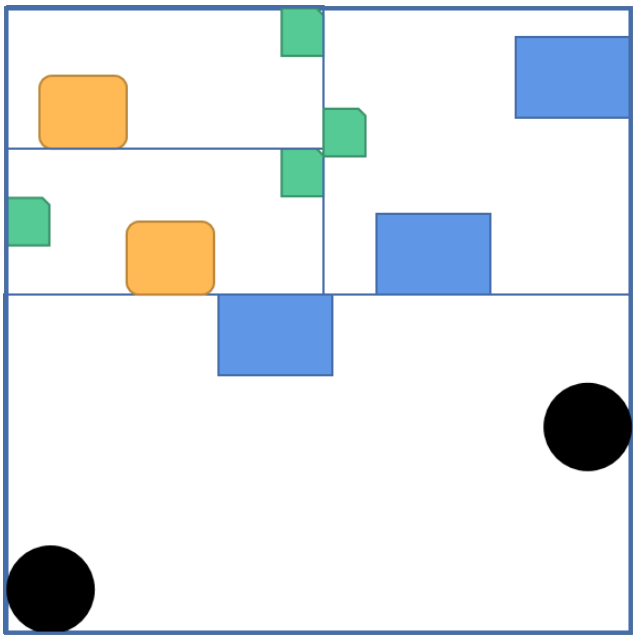

图 3. KdTree 示意图

KdTree 的编程实现部分主要注意合理的选取分裂的维度，对于三维空间我们按照 x->y->z->x->... 的顺序周而复始的进行空间分裂，尽量保证物体平均分布，这样能保证 KdTree 的平衡性保证查找效率。

由于场景中物体较少，无法体现 KdTree 对于算法性能提升的贡献，KdTree 加速空间查找的特性将在 KdTree 加速光子映射部分详细说明。

3.3. 实现对光源采样的路径追踪算法

3.3.1 **背景知识**

路径追踪是对传统光线追踪技术的升级，相对于传统光线追踪技术，路径追踪进一步根据辐射度量学理论采用更科学的着色方程，并且充分的考虑了漫反射问题，使得相对于传统光线追踪算法路径追踪算法能渲染出更真实的画面。

路径追踪的核心理论在于基于物理辐射度量学的渲染方程，以及利用蒙特卡洛积分法来对渲染方程进行求解，由于篇幅问题，在本文中不再对基于物理辐射度量学的渲染方程进行详细介绍，我们只

需要知道对于物体上一点，其渲染方程为：

$$L_r(x,\omega) = \int_{\Omega} L_i(x,\omega_i) f_r(x, w_i \to \omega) cos(\theta) \mathrm{d}\omega_i$$

对于该积分方程，我们在不清楚其解析式的情况下采用蒙特卡洛积分的方式对该方程求解。蒙特卡洛积分方程为：

$$\int f(x)\mathrm{d}x = \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} \frac{f(x_j)}{pdf(x_j)}$$

该方程涉及概率论的知识，在此便不再赘述，可以简单理解为蒙特卡洛积分是对原积分在数值上的无偏估计。在采样足够多的情况下，蒙特卡洛积分在数值上无限接近原积分。

3.3.2 实现

路径追踪算法便是将上述理论应用于渲染过程，和传统光线追踪一样，路径追踪也会首先从视点射出一条射线，与传统光线追踪的不同点在于，这条射线的方向是随机采样得到的，改射线与物体相交后，交点表面如果为反射或者折射类型，则和传统光线追踪算法一样，继续向着反射或折射方向射出一条射线递归计算，交点表面如果为漫反射类型，传统光线追踪算法会终止，路径追踪算法不同，考虑到漫反射路径追踪会继续随机采样一个方向射出射线 (蒙特卡洛采样)，直至递归结束 (达到最大递归深度或射线与光源相交)，至于为何采样数为 1(只随机射出一条射线)，原因是防止计算量指数级增长，如图4中所示，采样率大于 1 时，光线数量会随着递归次数呈指数增长。

当然，采样率为 1 的会导致我们得到的结果随机性强，与真实结果相差很大，路径追踪解决这个问题的方式就是在发出射线时 (第一次发出射线)，发射 n 条射线进行采样，从而在保证计算量可接受的情况下降低随机性。图5中给出了路径追踪的简单示意图，如同我们上文描述的，从每个像素点发出多个射线进行采样，最后返回多个结果的平均值为该像素的估计值。

这样做法的弊端很明显，如果没有足够的射线进行采样，我们得到的结果随机性很强，通俗来说就是得到的结果噪声很大，因此就有了在原始路径追踪基础上的改进方法，对于每个相交点我们在随机射出射线的基础上也向着光源方向采样。

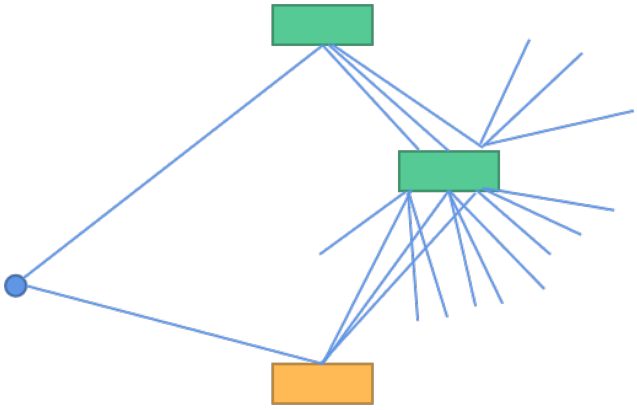

图 4. 计算量指数级增长示意图

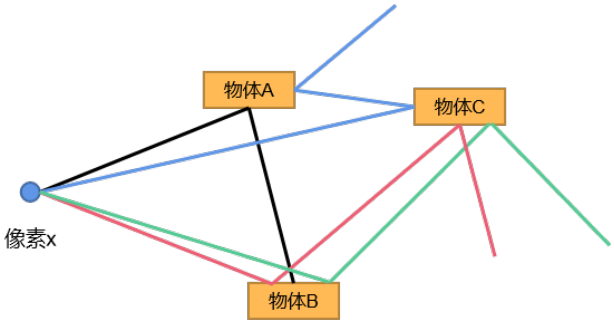

图 5. 路径追踪示意图

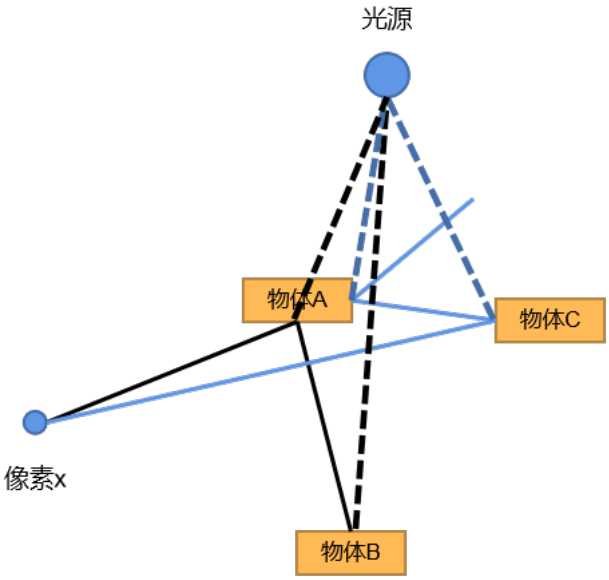

图 6. 对光源采样的路径追踪示意图

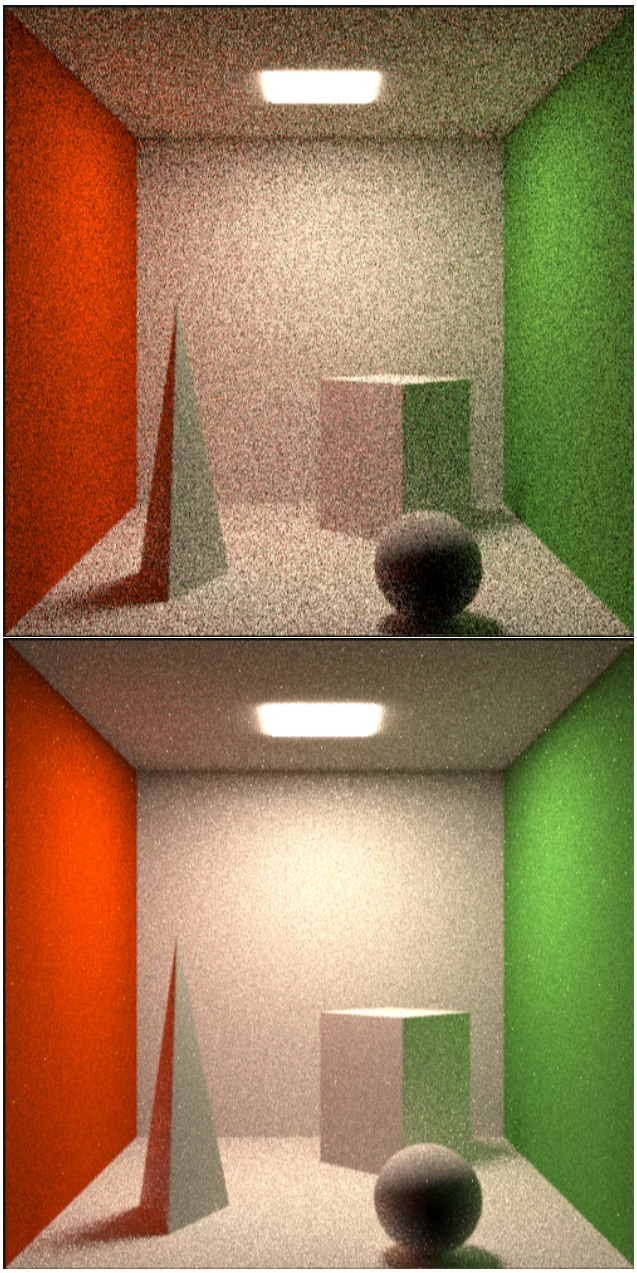

图 7. 传统路径追踪 (上图) 与对光源采样的路径追踪 (下图) 结果对比

图6中给出了向光源采样的路径追踪算法，算法也很好理解，就是在随机采样的基础上向光源采样，将两次采样的结果结合作为该点渲染方程的估计值，这样保证了直接光照，可以有效的减少渲染结果的噪声。图7中给出了框架中路径追踪算法和我们改进后路径追踪算法的渲染结果对比，同样在最大深度为 4，初始采样数为 512 的情况下，改进后对光源采样的路径追踪算法的噪声明显低于传统路径追踪算法。

3.4. 基于光子映射的路径追踪算法

3.4.1 背景知识

上文提到的路径追踪算法均是单过程的算法，从像素点射出 n(初始采样数) 条射线进行采样，每次与漫反射表面相交都会采用蒙特卡洛积分的方式随机射出一条射线进行随机采样。采样的不确定性导致在初始采样数很小的情况下算法得到的结果噪声很大。

光子映射算法 [2]继续对路径追踪算法进行了改进，光子映射算法是一个两阶段的算法。

3.4.2 实现

第一阶段从光源随机发射 n 个带有能量的光子，光子遇到反射或折射表明会根据向着反射或折射方向继续发射，遇到漫反射表面就将光子信息 (能量、位置、入射方向) 记录到全局光子表中，表示光子的部分能量被漫反射表明吸收，接下来光子会以一定概率被完全吸收，或更新能量和方向后继续向着某个随机方向发射。第一阶段不断进行，直到 n 个光子均被漫反射表明吸收，此时我们便得到了一个存有光子信息的光子表。

然后进行第二阶段，光子映射的第二阶段与路径追踪类似，也是从单个像素射出 n 条射线进行追踪，与路径追踪算法不同在于，对于某个交点光子映射使用该交点附近 k 个光子的能量和方向来对光照进行估计。该方法使用交点附近光子信息来估计光照，相对于利用蒙特卡洛积分的随机采样有着更好的准确度。图8中给出了光子映射的示意图，第一阶段光子从光源发出，在物体间反射 (镜面反射和漫反射)，然后被吸收的光子最终用于第二阶段的光照估计。

理解了原理，最终落实到程序实现也就比较简单了，只需要添加保存光子位置、能量、方向等信息的数据结构，并且模仿随机采样路径的方式实现随机发射光子即可，第二阶段依然沿用路径追踪的

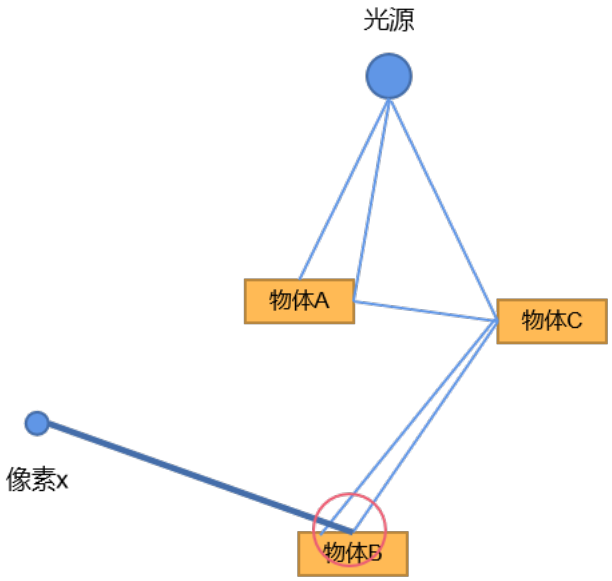

图 8. 光子映射示意图

算法即可，与之不同的是此时路径追踪不再需要递归实现，只需要根据最近的 k 个光子的信息估计光照即可。图9给出了光子映射的结果 (第二阶段采样数为 16，取最近的 50 个光子来评估光照)，可以看出由于光照估计是有偏的范围估计，结果会比较模糊，而且对于诸如墙角等边缘区域的渲染并不是很好，并且在光子数量较少的情况下图片中噪声也很严重。

在没有加速结构的支持下，1000000 个光子的映射耗费了 7594s 的时间完成一张图片的渲染，接下来我们进一步实现 KdTree 加速结构来加速光子映射算法。

3.5. KdTree 加速光子映射算法

3.5.1 背景知识

由于光子映射算法中第二阶段我们需要根据最近的 k 个光子的信息估计光照，在总光子数为 m 的情况下求最近的 k 个光子在不加速的情况下是一个时间复杂度为 $O(k*m)$ 的算法考虑到一般 $k << m$ 因此算法的时间复杂度为 $O(m)$ 对于每个像素我们都需要进行 n 次采样这样渲染算法总的时间复杂度为 $O(w*h*n*m)$。在光子数量 m 特别大的情况下算法的性能会非常低，因此我们需要 KdTree 加速结构将算法时间复杂度降低为 $O(w*h*n*log(m))$.

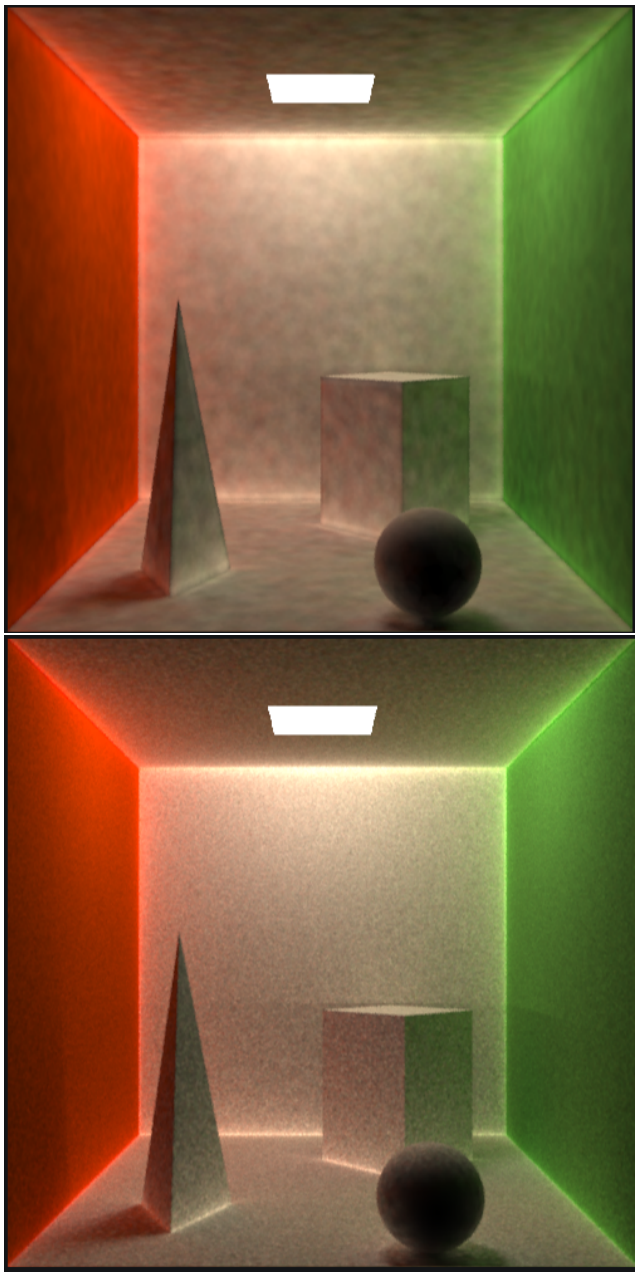

图 9. (第二阶段采样数为 16 的情况下) 光子映射取最近的 50 个光子，100000 个光子 (上图)1000000 个光子 (下图) 结果

3.5.2 实现

KdTree 的原理我们上文已经简单进行了介绍，在本节我们主要对 KdTree 加速效果进行测试。设置光子数量为 1e4，1e5，1e6 分别进行实验，根据渲染

photon num	1e4	1e5	1e6
cost(s)with KdTree	20.22	36.65	85.65
cost(s)without KdTree	83.57	635.27	7594.54

表 1. 性能测试

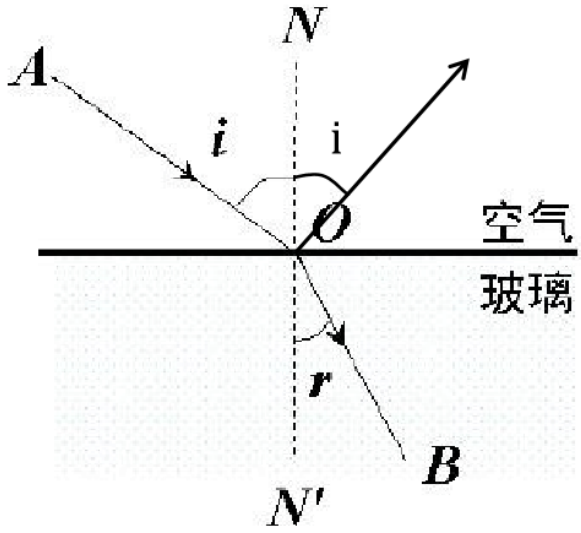

图 10. 折射 and 反射示意图 (空气-> 玻璃)

总时间来估计算法性能。表1中给出了性能测试的结果，由于没有 KdTree 加速的情况下算法运行过慢，因此仅进行一次实验，即使仅进行了一次实验，也不难从结果看出在光子数量增加的情况下，KdTree 结构对算法性能的贡献。

3.6. 实现基于菲涅尔方程的绝缘体材质渲染

本小节主要是在框架内提供的 Lambertian 针对漫反射表明着色的着色器的基础上进一步利用菲涅尔方程完成对绝缘体 (玻璃) 材质的 Glass 类型着色器。

3.6.1 **背景知识**

实现绝缘体材质着色器的核心在于光线邻域反射和折射的相关知识以及菲涅尔方程。图10中给出了常见的反射折射示意图，由于其中相关的物理知识很浅层，在此便不再赘述，我们这里主要讨论在图形学中如何处理折射和反射。

我们以玻璃和图10为例，对于一束光打到玻璃表明，首先会有部分能量被玻璃表明吸收，这取决于玻璃的材质，体现在视觉上是玻璃的颜色，然后剩余的光线，一部分会发生反射，反射角为 i，另一部分光线会发生折射，折射角为 r，这两者之间保持着 $n = \frac{sin(i)}{sin(r)}$ 其中 n 为物体的折射率，由物体材质决定。路径追踪算法中，对于物体与视线相交点，我们已经知道该点的入射角度以及法线方向，就可以利用公式和物体材质参数求得出射方向 out 以及折射角 r。

$$
\begin{aligned}
out &= \frac{N + in}{|N + in|} \\
sin(r) &= \frac{sin(i)}{n} \\
cos(r) &= \sqrt{1 - cos(r)^2} \\
cos(i) &= \frac{N \cdot in}{|N||in|} \\
sin(i) &= \sqrt{1 - cos(i)^2}
\end{aligned}
$$

根据公式可以计算出出射方向 out 以及折射角 r，再根据折射角 r 又可以计算出折射方向，至此关于折射反射的方向我们已经可以确定，但根据能量守恒定律，总能量 = 折射能量 + 反射能量，那么对于一束光，其能量该如何在折射与反射之间分配，这就涉及到了菲涅尔方程。

根据光学理论，光的反射率不是固定的，而是跟随入射角变化而变化的，观察表明，入射角 i 的值越大，反射率越大，这里就需要引入菲涅尔方程来帮助我们对反射率进行量化，菲涅耳方程是由法国物理学家奥古斯丁 · 菲涅耳推导出的一组光学方程，用于描述光在两种不同折射率的介质中传播时的反射和折射。方程中所描述的反射因此还被称作“菲涅耳反射”。

$$
\begin{aligned}
R_s &= \left| \frac{n_1 \cos\theta_i - n2\sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}}{n_1 \cos\theta_i + n2\sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}} \right|^2 \\
R_p &= \left| \frac{n_1\sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} - n2\cos\theta_i}{n_1\sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} + n2\cos\theta_i} \right|^2 \\
R &= \frac{R_s + R_p}{2}
\end{aligned}
$$

上文公式中给出了菲涅尔方程的解析式，在我们已知材质 1 的折射率 $n1$ 和材质 2 的折射率 $n2$ 以及入射角 θ_i 的情况下可见根据公式求得反射率 R，由于方程很复杂，因此在不追求高精度渲染的情况下，人们一般使用 Schlick 方程来近似菲涅尔方程。

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$
$$R = R_0(1 - R_0)(1 - cos(i))^5$$

本文中也将使用 Schlick 方程来近似求解菲涅尔方程。

至此我们具体的理论已经解释完成，接下来我们将实现对于非绝缘体的着色器，并且应用于路径追踪和光子映射渲染器，观察最终对于玻璃材质的渲染效果。

3.6.2 实现

框架中已经为我们提供了完善的添加着色器的接口，我们只需要声明一个针对绝缘体材质的 Glass 着色器并使其继承 Shader 抽象基类即可，在 Glass 中我们已经知道物体的材质，材质对应的吸收率 (颜色相关) 和折射率以及入射光线的交点和法线，根据上一小节提到的几何关系首先计算出反射光以及折射光的方向，再根据 Schlick 方程来求得反射率 R，R 是一个比例参数，根据 R 我们可以计算出反射方向和折射方向能量比例，从而进行分配。需要注意的是物体穿过玻璃材质射出时的法线方向需要进行矫正 (方向取负)，而且物体从玻璃射出时的折射率是射入时的倒数。

上述过程实现后我们将其植入路径追踪渲染器和光子映射渲染器，这部分工作主要是添加代码，与图形学无关，在此便不再赘述，接下来我们渲染带有玻璃材质物体的场景来观察效果。设置路径追踪递归深度为 4，像素采样数为 2048，光子映射设置光子数 1e7，像素采样数 64，进行渲染。图11展示了最终渲染效果，没有其他正确实现的图片来进行对照，不过单纯目测对于玻璃球的渲染还是令人满意的。个人认为相对于路径追踪，光子映射对于玻璃球聚焦、焦散的渲染效果更加真实，但光子映射依然在边缘处由于范围估计的原因会出现溢色的问题、如果有时间会考虑优化光子映射算法来解决该问题。

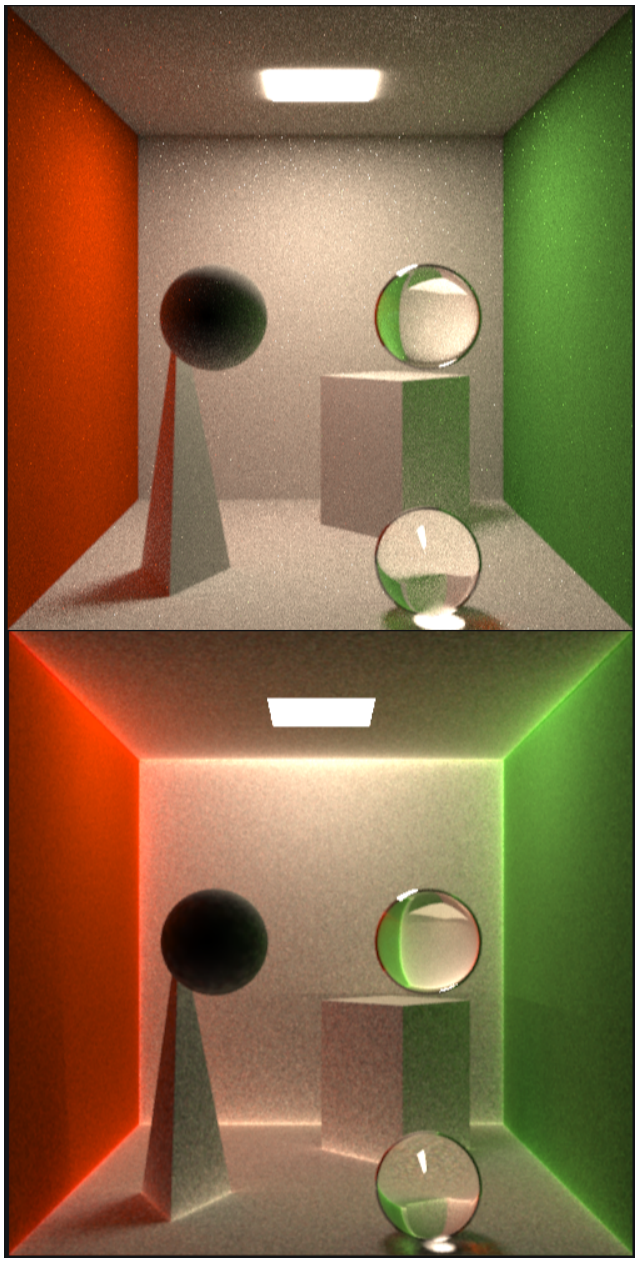

图 11. 路径追踪 (上图) 和光子映射 (下图) 渲染玻璃材质结果

4. 总结

通过本次大作业的实现，我对整个光线追踪的系列技术有个更加深入的理解，也加深了我对计算机图形学的兴趣，代码实现的过程提升了自己的编

result size	500*500	1000*1000	1500*1500
OpenMp cost(s)	0.0435	0.1321	0.2949
serial cost(s)	0.1428	0.6136	1.2940

表 2. 性能测试

程能力，资料查询的过程使得自己对模型-> 渲染-> 图片的过程有了更清晰的了解，通过自己的努力框架的功能越来越丰富，虽然大多数实现都是最简单，最基础的算法，但看着场景被渲染然后显示在屏幕上，心中还是会有不小的成就感。

在完成图像学作业的过程中我还顺带完成了 Kdtree 数据结构来加速光子映射算法，通过实践实现 KdTree 数据结构进一步提升了我将数据结构与算法知识应用于实际应用场景的能力，将光子映射算法的计算复杂度从不可接受下降到可接受的时长。本次图形学作业绝对物超所值，从中我学到了很多知识，不只是图形学的知识。

5. **附录** A

在正文中提到的光子映射算法在第二阶段光照信息的估计是利用了点附近 k 个光子的光照信息，实现光子映射的过程中还测试了利用了点附近距离为 r 的范围内的光子的光照信息进行光照信息估计，得到图12中的结果，总体来看画面更加平滑但画面显得很“脏”，而且物体边缘的渲染结果依然不是很好。

6. **附录** B

本次大作业还利用了 OpenMp 对渲染程序进行了多线程并行优化，虽然加速效果很明显，不过由于这部分工作量很少，因此放在附录中进行介绍，针对框架中原有的以及后来实现的渲染算法，其实并行很简单，考虑到最终渲染结果的图片中像素与像素之间的渲染没有数据相关，因此可以很简单的利用 OpenMp 针对 for 循环的优化，仅需要一条 OpenMp 编译宏和支持 OpenMp 的编译器，即可完成渲染算法的多线程并行优化。我们以 Ray Cast 算法为例，来展示并行优化的效果。从表2中的结果来看 OpenMp 并行优化的效果还是很明显的，基本可以保持加速比为 4，即比串行程序快 4 倍。

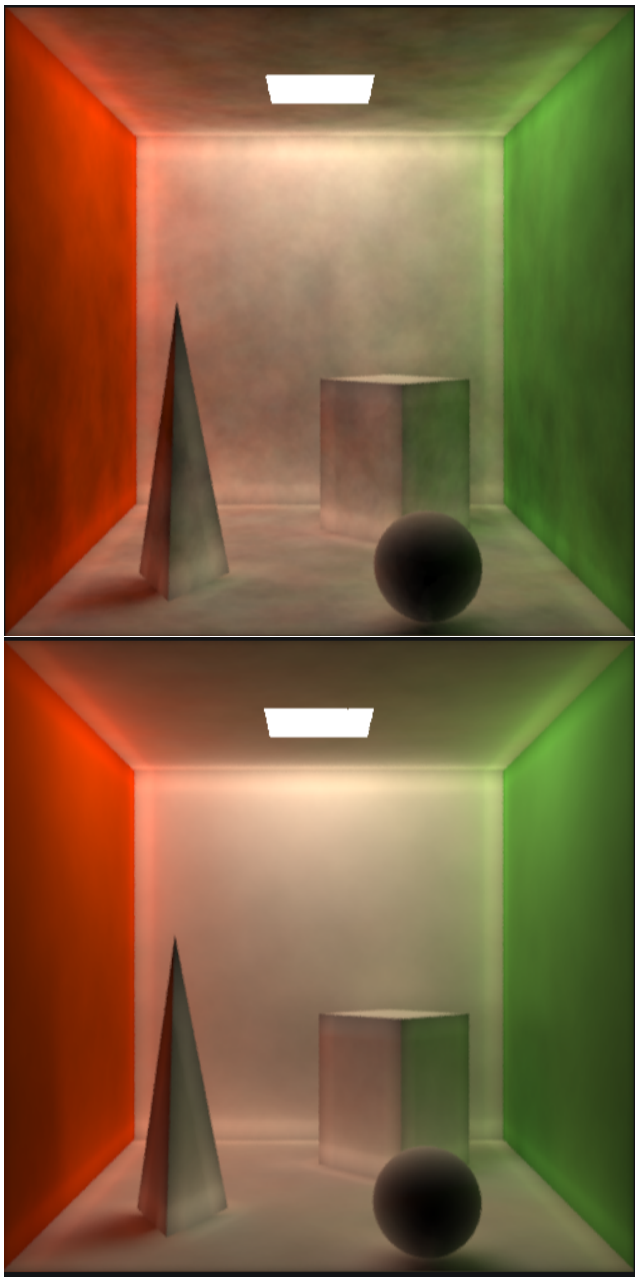

图 12. 限定范围 r 的情况下 (第二阶段采样数为 16 的情况下) 光子映射 100000 个光子 (上图)1000000 个光子 (下图) 结果

7. **附录** C

考虑到光子映射算法引入了新的参数“光子数量”，原框架内并没有 UI 模块来对光子数量进行调整，因此在框架内添加光子数量调整器 UI。图13展示

图 13. 添加光子数量选项

了配置器在框架中添加的位置，设置默认值为 1000，如果追求渲染效果，建议将光子数量设置为至少 1000000。在实现该功能前，调整光子数量通过宏定义来进行，每次调整光子数量都需要重编译整个项目，实现该功能后可以更加方便的在运行时动态调整光子数量，从而对比不同光子数量对渲染效果的影响。

参考文献

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In Proceedings of the April 30–May 2, 1968, spring joint computer conference, pages 37–45, 1968. 2

[2] Henrik Wann Jensen. Global illumination using photon maps. In Eurographics workshop on Rendering techniques, pages 21–30. Springer, 1996. 5

[3] Andrew W Moore. An intoductory tutorial on kd-trees. 1991. 2

[4] Turner Whitted. An improved illumination model for shaded display. In Proceedings of the 6th annual conference on Computer graphics and interactive techniques, page 14, 1979. 1