光线追踪项目报告

姚嘉豪 521021911060

摘　要：光线追踪 (Ray Tracing) 算法是计算机图形学领域中一种重要的三维图形渲染算法，通过模拟光线在场景中的传播来生成高质量的图像。目前，业界和学术界产出了 OpenGL、Direct3D 等工具，在光线追踪实时渲染上做出了十分瞩目的效果。而本文采取不使用任何 API 的方法，参考 Ray Tracing In One Weekend、GAMES101 等教程自行实现了简单的光线追踪渲染器，并增加了多线程加速、OBJ 文件的读取等算法来实现更好的渲染速度和效果。最终在 cornel box 中渲染出了上海交通大学庙门的模型，并取得了不错的效果。

Ray Tracing Report

Jiahao Yao

Abstract: The Ray Tracing algorithm is a crucial 3D graphics rendering technique in the field of computer graphics. It generates high-quality images by simulating the propagation of rays of light within a scene. Currently, industry and academia have developed tools such as OpenGL and Direct3D, achieving remarkable results in real-time ray tracing rendering. This paper, however, takes a different approach by implementing a simple ray tracing renderer without relying on any API. Drawing inspiration from tutorials like "Ray Tracing In One Weekend" and "GAMES101," it incorporates enhancements such as multi-threading acceleration and OBJ file parsing algorithms to achieve improved rendering speed and quality. Ultimately, the renderer successfully produces impressive results, rendering a model of the Shanghai Jiao Tong University's main gate in the Cornell Box scene.

Key word：Ray Tracing, no-API, acceleration

1　简介与意义/Introduction

光线追踪（Ray Tracing）是计算机图形学中一项关键的渲染技术，其基本原理是通过模拟光线在场景中的传播来生成逼真的图像。相较于传统的光栅化渲染技术，光线追踪能够更准确地模拟光的行为，包括反射、折射、阴影和光影等效果，因而能够产生更真实、细致的视觉效果。下面是光线追踪核心算法的伪代码：

```
遍历屏幕的每个像素{
    创建从视点通过该像素的光线
    初始化最近 T 为无限大，最近物体为空值
    遍历场景中的每个物体{
        如果光线与物体相交{
            如果交点处的 t 比最近 T 小{
                设置最近 T 为交点的 t 值
                设置最近的物体为该物体
```

```
}   }   }
如果最近物体为空值{
    用背景色填充该像素
}否则{
    对每个光源射出一条光线来检测是否处在阴影中
    如果表面是反射面，生成反射光，并递归
    如果表面透明，生成折射光，并递归
    使用最近物体和最近 T 来计算着色函数
    以着色函数的结果填充该像素
 }  }
```

作为一种物理模拟为基础的渲染方法，光线追踪技术通过准确地模拟光的反射、折射和散射等现象，提供更真实感的图像效果。光线追踪遵循三个基本原理：光沿直线传播、光线不会相互碰撞、光路是可逆的。从光源发出的光线经过连续的反射或直射进入眼睛，同时这个过程也可以反向表示，即光线从眼睛发出，经过一系列反射和折射到达光源。在场景中模拟光线的投射过程，通过每个像素发射一条光线，确定该像素能够“看到”场景中的哪些物体。然后，从光线击中物体的点发射一条朝向光源的光线，根据是否能到达光源来判断该点是否处于阴影中。换言之，光线追踪（这里指的是反向光线追踪）的基本原理是逆向追踪与相机镜头相交的光线路径，并计算光线在场景中与物体交互时的各种物理效应。

在实际的渲染场景中，不同材质对光线路径的影响至关重要。举例来说，需要准确计算光线在某一点的反射和折射。基于这些计算，可以进一步确定阴影的位置和形状。在渲染过程中，每条光线在不同物体表面上的所有反射和折射点都被累加，以确定每个像素的最终颜色值。这个过程有效地模拟了光线在场景中的行为。

近年来，光线追踪在电影、游戏和虚拟现实等领域取得了巨大成功。引入实时光线追踪技术的游戏，如《Minecraft》和《Cyberpunk 2077》，呈现出令人惊叹的视觉效果，使得虚拟世界更加细腻生动。同时，光线追踪在电影制作中的运用，如《模拟游戏》和《猫鼠游戏》，为影片注入了更为真实的光影表现，提升了观众的沉浸感。

本文在 Ray Tracing in One Weekend [1] 和 Ray Tracing Next Week [2] 两本书籍的基础上，自行设计了光线追踪器，并进行优化创新，获得了更好的渲染质量与效率。

2 相关工作/Related Works

2.1 经典光线追踪算法/Classic Ray Tracing

光线追踪算法的起源可以早至 1968 年，Arthur Appel 在一篇名为《Ray-tracing and other Rendering Approaches》[3]的论文中提出的 Ray Casting 的概念，称为光线投射。光线投射是一根单一的从一个点向一个方向发射出光线，它与场景中的物体相交时停止，计算出的是光照方程中直接光照的部分。1986 年，Kajiya 统一了光照公式，并推导出了光照公式的路径表述形式，使得光照公式由一个递归的结构，变成一个路径函数的积分，因此蒙特卡洛的每个随机数只要产生一条路径即可，这些路径不需要是递归的，因此每条路径可以随机生成，然后

每个路径的值作为一个随机数用于计算最终的光照结果。这种新的形式称为路径追踪（Path tracing），如下图所示。在路径追踪算法中，首先在场景中物体的表面随机产生一些点，然后将这些点和光源以及摄像机链接起来形成一条路径，每个路径就是一个路径函数的随机值。这样的路径，根据场景的复杂度，每帧可能包括上亿条光线，因此传统的路径追踪算法很难运用到实时渲染领域。

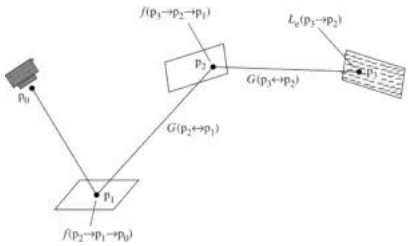

图 1 路径追踪示意图

2.2 现代光线追踪优化/Ray Tracing Optimization

从历史上看,光线追踪技术一直只能在非实时应用中使用,视频游戏只能依靠光栅化进行渲染。而 GPU（图形处理器）是针对并行计算而设计的硬件，适用于大规模数据并行处理。在光线追踪中，GPU 可以同时处理多个像素的计算，每个像素的光线追踪过程独立进行。NVIDIA RTX 的出现促进了计算机图形学的新发展,实现了对光照,阴影,反射作出反应的交互式图像渲染。

重要性采样是一种在采样过程中更注重对场景中重要区域的采样技术。通过分配更多的采样样本给对最终图像有更大影响的区域，提高渲染效率和质量。重要性采样，可以更有效地捕捉光照的变化，减少渲染图像中的噪音，提高最终图像的真实感。

3 研究内容与方法(或算法)/Contnts and Methods(or Algorithm)

3.1 算法框架/Framework of Algorithm

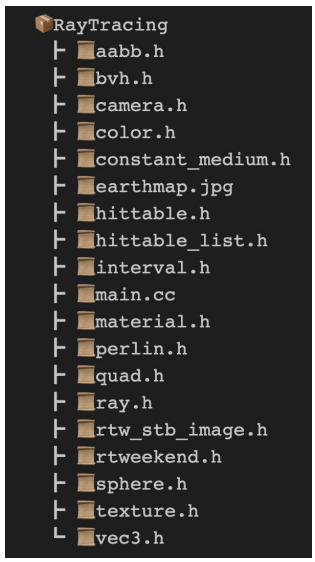

图 2 RayTracing 文件树

本文的光线追踪算法的框架由一系列头文件和主程序文件组成，每个文件负责不同的功能：

1. aabb.h 包含轴对齐包围盒(AABB)的定义和相关操作，是一种用于提高光线追踪效率的空间加速结构。
2. bvh.h 实现了包围盒层次结构（Bounding Volume Hierarchy，BVH），通过递归划分场景中的物体，降低射线相交测试次数。
3. camera.h 包含相机类的定义，负责生成光线以进行场景渲染，设置相机参数、视图矩阵等。
4. color.h 定义了颜色类和相关操作，用于表示和处理光线追踪中的颜色信息。
5. constant_medium.h 实现了常介质（Constant Medium）类，用于模拟光线穿过各向同性介质时的效果。
6. hittable.h 定义了可被光线击中的物体的抽象基类，包含了射线与物体相交的接口。
7. hittable_list.h 实现了包含多个物体的列表类，用于管理场景中的物体。
8. interval.h 包含区间类的定义，用于表示光线追踪中的数值区间。
9. material.h 包含光线追踪中材质的定义，如漫反射、金属、电介质等。
10. perlin.h 实现了 Perlin 噪声生成的类，用于纹理和材质的复杂表现。
11. quad.h 定义了四边形类，用于创建场景中的平面。
12. ray.h 包含射线类的定义和相关操作，用于描述光线在场景中的传播。
13. rtweekend.h 包含一些通用的常量、工具函数和头文件引用，提供方便的数学和随机数生成函数。
14. sphere.h 包含球体类的定义，是光线追踪场景中常见的几何体之一。
15. texture.h 包含纹理类的定义，用于赋予物体表面更复杂的外观。
16. vec3.h 包含三维向量类的定义和相关操作，用于表示光线追踪中的位置、方向等信息。
17. main.cc 是主程序入口，设置场景、相机参数，调用光线追踪算法进行渲染。

3.2 漫反射模型/Diffuse Reflection Model

本文采用 Lambertian 分布作为漫反射模型。该分布以与表面法线之间的角度Φ成比例地散射反射光线，其中Φ是反射光线与表面法线之间的角度。这意味着反射光线最有可能在接近表面法线的方向上散射，而在远离法线的方向上散射的可能性较小。与均匀散射相比，这种非均匀的 Lambertian 分布更好地模拟了真实世界中材料的反射。

将一个随机单位向量添加到法线向量上来创建这个分布。在表面上的交点处，有一个命中点 **P** 和表面的法线 **n**。在交点处，该表面有两个唯一的接触点，所以对于任何交点，只能有两个与之相切的单位球（每个表面的一个唯一的球）。这两个单位球将通过其半径的长度从表面上移开，对于单位球来说，半径恰好为 1。一个球将沿着表面法线的方向 **n** 移开，另一个球将沿着相反的方向**-n** 移开。这样就得到了两个单位尺寸的球，它们只在交点处“刚好”接触表面。其中一个球的中心位于 **P-n**，另一个球的中心位于 **P+n**。中心位于 **P-n** 的球被认为是在表面“内部”，而中心位于 **P+n** 的球被认为是在表面“外部”。

选择与光线起点在表面同一侧的切线单位球。在这个单位半径球上选择一个随机点 S，并从命中点 P 向随机点 **S** 发送一条光线（向量 **S-P** ）：

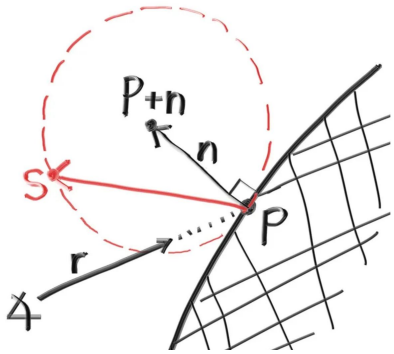

图 3 根据 Lambertian 分布随机生成一个向量

3.3 边界体积层次结构/Bounding Volume Hierarchy(BVH)

文件bvh.h中的bvh_node 类是一个包围盒层次结构Bounding Volume Hierarchy(BVH)的实现，用于加速光线追踪算法。该类通过构建一种树形结构，将场景中的物体划分为包围盒，以减少光线和物体的相交测试，从而提高渲染效率。

类的成员变量包括 **left** 和 **right**，分别表示 **BVH** 节点的左右子节点，以及 **bbox**，表示该节点的包围盒。这些成员变量在判断光线和节点相交时起到关键作用。

hit 函数用于判断给定射线是否与当前 **BVH** 节点相交。首先，检查射线是否与节点的包围盒相交，如果不相交，则直接返回 **false**。如果相交，继续递归调用左右子节点的 **hit** 函数，更新击中信息。这种递归判断方式能够有效地排除大量的无关物体，提高渲染效率。

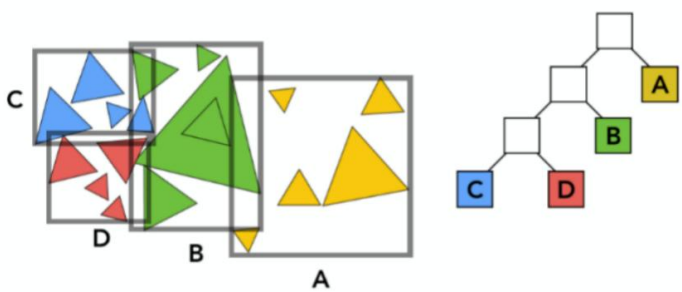

图 4 BVH 划分示意图

3.4 恒定密度介质/Constant Density Medium

常密介质是一种用于模拟雾、烟雾或其他透明散射材料的技术一束光线穿过常密介质时，可以在体积内部散射或者直接穿过介质。介质密度和体积也决定了光线直接穿过的难易程度。当光线穿过体积时，它可能在任何点散射。体积越密，这种可能性就越大。光线在任何小距离ΔL内散射的概率是：

$$\text{probability} = C \cdot \Delta L$$

material.h 中的 isotropic 类实现了
各向同性材质的散射行为，其中散射方向是随机的，漫反射系数由给定的颜色或纹理确定。

在下文的最终效果图中可以看到左下角的白色烟雾块体现了恒定密度介质。

3.5 多线程加速/Multi-thread acceleration

在 Ray Tracing In One Weekend [1]的渲染器实现中，整个渲染过程由单一线程完成，严重影响其效率。而经典的多线程加速算法，通常只是简单地将图片均分为和线程数量相同的等比例子区块，使用每个线程渲染不同区块。这种渲染方法的不足之处在于：图片的某些区域相对而言更为复杂，计算时间与光线散射次数更多，在渲染的后期，一些线程可能会很早的完成任务，导致线程的空置和计算资源的浪费。

为了提升效率，本项目使用了 C++的线程库<thread>来实现多线程加速。具体而言，本项目创建一个包含多个线程的线程池，选择按行分割，保证待分配行数变量的原子性，每个线程负责渲染图像的一部分，从而在并行处理的情况下提高渲染速度。在渲染高分辨率图像时，多线程可以更有效地利用多核处理器的性能。

3.6 更优的光源采样算法/Light Source Sampling

在光线追踪过程中，光源的渲染是一个关键环节。单纯依赖反射光线的渲染，在使用蒙特卡洛方法渲染场景时，需要大量的采样以减少渲染图像中的噪点。为解决这一问题，先前的研究提出了一种直接向光源进行采样的算法。这种方法在考虑反射光线的同时，加入了对光源直接采样的概率，以此在保持概率分布函数归一性的前提下，实现了更优的采样概率分布。该算法通过减少所需的采样次数，能生成噪点更少的场景图像。

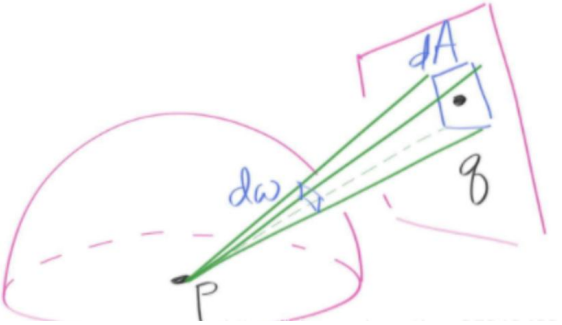

图 5 直接光源采样示意图

4 实验结果与分析/Experiment Results and Analysis

测试场景引入具有交大元素的模型。将上海交通大学东大门的照片贴于 Cornell Box 的内侧墙面，增加玻璃球和地球纹理展示反射、折射与阴影效果。

测试平台为 MacOS 14.1.2　Apple M1。

测试分辨率为 600×600，每像素光照采样次数（SPP）为 4096，光线弹射次数上限为 6。渲染耗时 57 分钟。可见纹理渲染全部正确，阴影、玻璃球等效果较逼真，同时保持了较高的渲染效率。

图 5 最终渲染效果图

5 特色与创新/ Distinctive or Innovation Points

这项目采用了无 API 调用的方式实现了简单的 CPU 光线追踪器,基于经典光线追踪算法。在这个基础上，引入了一些创新和特色，其中包括:

1. 引入了包围体层次结构（BVH）作为渲染的加速手段，通过构建树形结构对场景进行层次划分，以提高整体渲染效率。

2. 采用 C++的多线程库实现了对图像的并行渲染，充分利用多核处理器，以更快的速度完成渲染任务，为用户提供更高效的渲染体验。

3. 通过优化的光源采样算法，进一步改进了光线追踪的渲染效果，得到更真实和高质量的图像输出。

综合而言，本项目在传统光线追踪算法的基础上，通过引入 BVH、多线程加速和优化的光源采样算法，实现了更高效的渲染速度和更真实的渲染效果。

6 项目心得/Self-Experience

光线追踪项目是一个充满挑战与乐趣的学习过程。通过这个项目，我深入理解了计算机图形学的核心原理，尤其是光线追踪算法的实现细节。在这个过程中，我深刻体会到了 C++

中面向对象的理念，并使用了多线程库进行加速。

最令人满足的时候是看到最终渲染出的图像，感受到项目的成果。这次经历让我对计算机图形学产生了更深远的兴趣，也激励我继续追求更高层次的图形学技术。

References:

[1] Ray Tracing in One Weekend. raytracing.github.io/books/RayTracingInOneWeekend.html, Accessed 11,30,2023.

[2] Ray Tracing Next Week. raytracing.github.io/books/RayTracingNextWeek.html, Accessed 11,30,2023.

[3] Appel, A. (1968, April). Some techniques for shading machine renderings of solids. In Proceedings of the April 30--May 2, 1968, spring joint computer conference (pp. 37-45).