光线追踪

王智烨(521021911051)

摘　要: 在计算机图形学中，光线追踪是一种模拟光在三维场景中传播和交互的关键方法，能够获得逼真的渲染效果。如今，伴随着 GPU 性能的提升，许多研究者在深度学习的帮助下进行实时的光线追踪渲染，并在游戏引擎、影视制作上获得了不错的效果。该项目结合 CS3310 课程内容并参考 Ray tracing in one weekend 等相关资料，使用 C++语言实现了一个基本的光线追踪程序以及一些基本的物体材质和纹理。通过引入基于物体的包围盒层次结构算法以及多线程处理，能够更加迅速地处理复杂的模型场景。除此之外，该项目支持对于 OBJ 文件的读入以及材质设置，能够渲染更加复杂的物体，完成了对于上海交通大学校徽雕像以及一些 OBJ 模型的渲染，展现了光线的直射、反射和折射等效果，实现了效率和渲染质量较好的平衡。

Ray Tracing

Zhiye Wang

Abstract:

In computer graphics, ray tracing is a key technique for simulating the propagation and interaction of light in three-dimensional scenes, providing realistic rendering effects. Today, with the improvement of GPU performance, many researchers leverage deep learning to achieve real-time ray tracing rendering, obtaining impressive results in game engines and film production. This project, combining content from the CS3310 course and referencing materials like "Ray tracing in one weekend," implemented a basic ray tracing program using the C++ language, featuring fundamental object materials and textures. By incorporating an object-based bounding volume hierarchy (BVH) algorithm and multithreading, the program efficiently handles complex model scenes. Furthermore, the project supports loading OBJ files and material settings, enabling the rendering of more intricate objects. It successfully rendered the statue of the SJTU emblem and other OBJ models, showcasing effects such as direct, reflective, and refractive rays, achieving a balance between efficiency and rendering quality.

Key word：Ray tracing, multithread, bounding volume hierarchy

1　简介与意义/Introduction

1.1 项目意义和依据/Significance

光线追踪是一种用于模拟光在三维场景中传播的图形学技术。它从观察者的视点出发，通过追踪光线在场景中的传播路径，计算光线与物体之间的交点和光的交互，从而生成最终的图像。与传统的图像渲染技术相比，光线追踪在模拟光的真实行为上更为精确，因此常常能收获更加逼真的效果。

目前，伴随着硬件艺术的不断发展尤其是图形处理单元（GPU）性能的提升，实时光线追

踪正在成为图形学领域的热点，许多游
戏引擎和图形库正在引入实时光线追踪技术；除此之外，也有许多研究人员试图利用神经网络来加速光线追踪的计算过程，提高渲染速度，并用深度学习生成更真实的场景和材质。

本项目使用 C++语言，结合课程教学内容以及 ray tarcing in one week[6] ，完成了一个基本的光线追踪程序。支持 obj 模型文件的读入，并设计了三角形，四边形和球体三种几何形状。为了能够达到更加逼真的渲染效果，实现了金属、非金属和基本的 Lambertian 反射等材质以及从图片的纹理载入。为了提高运行速度，本项目实现了多线程的逐行渲染以及包围体层次结构（BVH）[5] ，以加速光线与物体场景的相交检测。

1.2 本方法/系统框架/Article Structure

该项目的框架包含以下几个部分：光线形成、光线与物体的相交检测、光的传播与反射、光线递归追踪和图像输出。

在光线形成阶段，程序会从相机视点发出光线，这些光线沿着像素的方向穿过图像平面；在物体相交检测时，该项目利用 BVH 优化检测步骤，确定光线与物体表面的交点；在相较之后，与真实的光线相同，框架需要计算光线在物体表面的反射、折射、漫反射等光学效应；在折射等情况下，程序需要递归地追踪光线路径达到更好的效果。

2 相关工作/Related Works

2.1 光线追踪算法

1968 年，Arthur Appel[1] 首次引入了用于渲染的光线投射（Ray Casting）算法。该研究通过从观察点对每一个像素发射一条光线并在待渲染的世界场景中找到第一个阻挡光线路径的物体来渲染场景。为了解决场景中没有间接光的问题，Turner Whitted[2] 引入了反射和折射光线。通过这种方法，光线的投射过程被延长。反射光线在物体表面沿镜面反射方向继续照射。而反射的颜色则由反射光线与场景中的对象的交点决定。然而，Whitted 光线追踪只实现了完美的镜面反射，对于粗糙的镜面显示便不够准确。

针对上述的问题，Robert L. Cook[3] 等人在 1984 年提出了分布式光线追踪算法（Distributed Ray Tracing）。在该算法中，一条可视光线的辐射亮度是来自多条入射光的蒙特卡洛估计（见公式 1）,对于反射和折射光是来自在该光束各自对应的发现方向的半空间内对方向的采样，而阴影光束则是在光源范围内的方向采样。

$$\mathrm{In} = \frac{1}{n} * \Sigma_{i=1}^{n} \frac{g(x_i)}{p(x_i)} \tag{1}$$

Kajiya[4] 通过提出渲染方程，成功地物理学和统计学意义上建立了光线追踪的科学模型，精确地在程序中定义了光和物体的作用以及光的传播方式。

2.2 光线追踪的优化与加速

光线追踪算法的很大一部分计算开销来源于计算光线与物体碰撞的计算，包围盒通过用一个相对简单的形状将一个复杂的形状包围起来。通过物体进行划分空间结构，将不同的物体防

止在不同的包围盒内，省去了三角形和包围盒求交的过程。

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

3.1 光线追踪框架

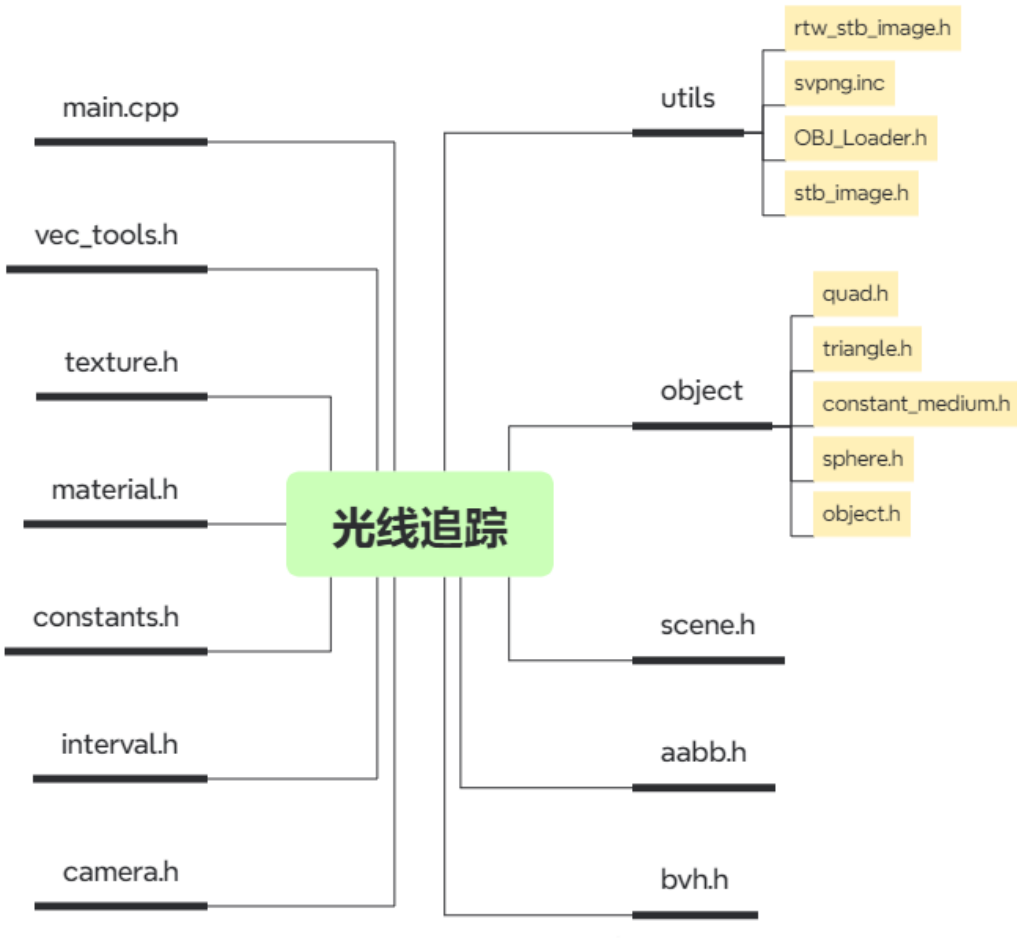

图 1 光线追踪框架

光线追踪的框架如上图所示，向量实现借助于 OpenGL 的 glm 库，其提供了向量的叉乘、点成、单位化等常用的操作。向量工具类 vec_tools.h 则实现了 glm 向量库的扩展，提供随机生成向量、折射、反射等工具；obj 的文件的读取借助 OBJ_Loade.h，获取文件中网格的顶点信息。

光追的核心部分为相机类 camera.h 和光线类 ray.h。前者记载了待渲染图像的信息以及相机的参数，并包含了渲染等关键步骤；光线类的作用是产生光线，并且通过成员变量记录光线

的起点和方向信息，并且提供相关函数
支持计算光线在指定时间 t 的位置。

物体实现的基类是 object.h，定义了一个抽象的可被光线击中的物体，包含了其包围盒以及碰撞函数。为了保持的物体类本身的简介以及可扩展性，光线与物体相交的信息则被储存在 record.h 类中，包括了焦点的位置、法向位置、碰撞点所处的物体材质以及光线相交时的位置参数、纹理坐标等等信息。在材质中，本项目实现了基础的漫反射材质、金属材质、透明材质、发光材质和气体。纹理的实现则是借助 texture 基类，提供了单一颜色纹理以及从图片加载纹理。

渲染器会对每个像素执行 get_ray 获取光线，为了保证光线的反射与折射会最终停止，每条光线都会设置递归深度。在计算像素颜色的过程中，会使用 intersection 函数检查是否与物体相交，如果不相交则会返回黑色。若相交，渲染器会调用材质的 scatter 计算光线的散射方向以及对应的衰减系数，进行递归计算。若材质不会发生散射，则会返回材质的自发光颜色。为了防止图片走样，渲染器会对每个像素进行多次采样（samples per pixel）并求平均值，得到最后像素的真正颜色。

3.2 OBJ模型的读入以及triangle的实现

在 Ray tracing in one weekend[6] 中，作者实现了球体、平面的物体渲染。然而在实际的光线追踪应用中，模型文件常常通过 OBJ 导入进行实现。因此，本项目针对 OBJ 文件，实现了 triangle 三角形类，提供了该类文件的读取方法。

在 OBJ 文件中，其产生的 mesh 常常由一系列的顶点、发现、纹理坐标和面（三角形或四边形）构成。因此，该项目首先实现了三角形类。不同于球体类需要一个顶点和一个半径进行定义，三角形类的定义依赖于三个顶点的坐标。光线的求交采用了 Moller-Trumbore 方法（详见算法 1）。三角形类的包围盒的生成则是选取三个顶点在 x、y、z 三个坐标中各自最小和最大的点组成。

算法 1：Moller-Trumbore

```
Input: v0, v1, v2, orig, dir
Output: tnear, u, v
S = orig - v0
E1 = v1 - v0
E2 = v2 - v0
S1 = crossProduct(dir, E2)
S2 = crossProduct(S, E1)
S1E1 = dotProduct(S1, E1)
t = dotProduct(S2, E2) / S1E1
b1 = dotProduct(S1, S) / S1E1
b2 = dotProduct(S2, dir) / S1E1

if t >= 0 and b1 >= 0 and b2 >= 0 and (1 - b1 - b2) >= 0
```

```
11.      tnear = t
12.      u = b1
13.      v = b2
14.      return true
15.  return false
```

本项目借助了 OBJ_Loader 加载 OBJ 文件，其能够读取出 OBJ 文件中的 Mesh 网格信息。随后，通过遍历网格的所有顶点，为每三个顶点构造一个三角形，并指定相应的材质，添加到场景容器中。

3.3 包围盒层次结构

在光线追踪中，为了减少光线与场景中物体的相交测试次数以提程序运行效率，可以利用包围盒层次结构（Bounding Volume Hierarchy）[5] 层次化地组织场景中的物体达到该效果。

该项目设计了继承自 object 的 bvh_node 类来表示包围盒层次结构的节点。在 BVH 的构建过程中，程序会先选择一个轴，根据该轴上对场景中的物体进行排序。随后递归地将物体划分到左右子树中构建 BVH。在这个过程直到每个叶子节点都只包含一个或者两个物体。最终，每个节点都会有一个包围盒，用于进行加速光线相交测试。需要注意的是，BVH 作为物体划分空间而不是使用包围盒将物体进行划分，并不追求包围盒的严格分开，包围盒之间的重叠在程序中也是允许的。

当需要使用 BVH 时，只需要将场景物体集合 scene 重新以 BVH 的形式进行组织，算法将直接与 BVH 进行交互，而不是与场景中的物体列表。

算法 2：Intersect-BVH

```
Input: ray, bvh_node
Output: hittable
1.  if ( ray misses node.bbox )
2.      return
3.  if ( node is a leaf node )
4.      test intersection with all objs
5.      return closest intersection;

6.  hit1 = Intersect-BVH( ray, node.child1 )
7.  hit2 = Intersect-BVH( ray, node.child2 )

8.  return the closer of hit1, hit2
```

3.4 多线程加速

本项目最初使用单线程进行渲染，但实际上在渲染的过程中，每个像素点的渲染都是独立的，因此可以通过使用多线程并行渲染不同的像素点完成渲染阶段的加速。

结合到项目原先的设计结构，即先对行进行循环，再对不同列的像素进行渲染的顺序，为

了尽可能避免不同进程间的竞争，该项
目安排每个线程处理图像的不同行。因此，线程函数通
过 Lamda 表达式进行定义，每个线程通过不断自增 next_row 变量来获取需要处理地行数。

在多线程的实现中，由于每个线程的执行需要了解当前待执行的行数，会频繁访问待渲染

行的这一变量。为了避免线程之间的竞争，用于表示待渲染的下一行 next_row 变量通过 C++ 的 atomic 库进行声明，保证对于该变量的操作是原子的，有效提高多线程运行的安全性和稳定性。

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

图 2 光追渲染结果

该场景（图 2）使用 11 代 Intel i7 CPU 的 15 线程进行渲染，分辨率大小设置为 800*800，SSP（Samples per pixel）设置为 4096。其中交大雄校徽模型拥有 70604 面，4 颗星星共有 24000 面左右，画面右下角的草与石头共 7000 面，整个场景一共在 100000 面左右，总渲染时间在 5 小时左右。

在整个场景中，除去四周的背景墙，所有模型皆通过 OBJ 文件进行读入。四颗星星的材质被分别设置为金属和玻璃，而石头的纹理则是通过读入纹理照片进行实现的。可以从图像中观察到明显的反射、折射等光学现象。例如从金属星星的表面可以看到一定的墙壁颜色，这是由于光线的反射形成的。而透过玻璃材质的星星可以看到背景墙的深蓝色，这则是光线折射效果的体现。观察场景中物体的阴影以及和光源的相对位置，符合现实生活中的物理规律。因此，本项目较好地实现了一个基本地光线追踪程序，能够支持 OBJ 文件的读入以及贴图，并通过多线程和 BVH 获得了一定的加速效果。

5 特色与创新/ Distinctive or Innovation Points

本项目使用 C++语言实现了一个简单的光线追踪程序。并且使用了 BVH 技术加快了光线与物体的碰撞检测过程。除此之外，本项目通过三角形类的设计实现了对于 OBJ 文件的读入，扩大了程序可渲染的模型范围。并且通过基于 thread 库实现了光线追踪程序的多线程渲染，获得了明显的加速效果。

通过本次项目的实践，同学们完成了从基础的向量类编写开始，逐渐搭建出一个基本的光线追踪渲染程序，并基于自己的理解和需求添加了一些课程以及教程之外的内容。这种亲手慢慢构建出一个整个工程项目的经历让同学们受益匪浅，也学习了如何发现问题以及解决问题。除此之外，该项目展现了计算机图形学神秘而又有趣的一面，激发了同学们在该领域内不断探索的兴趣。最后，也十分感谢老师们的指导以及助教对于本小组问题的耐心解答。

References:

[1] 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)

[2] Whitted, T. (1979, August). An improved illumination model for shaded display. In Proceedings of the 6th annual conference on Computer graphics and interactive techniques (p. 14)..

[3] Cook, R. L., Porter, T., & Carpenter, L. (1984, January). Distributed ray tracing. In Proceedings of the 11th annual conference on Computer graphics and interactive techniques (pp. 137-145).

[4] Kajiya, J. T. (1986, August). The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (pp. 143-150).

[5] Arvo, J., & Kirk, D. (1989). A Survey of Ray Tracing in. An Introduction to Ray Tracing",(ed. Press, 201-262.

[6] Shirley, P., Black, T. D., & Hollasch, S. (2023). _Ray Tracing in One Weekend_. Retrieved from https://raytracing.github.io/books/RayTracingInOneWeekend.html