学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301002	曾令腾	8	25	16	36	85

计算机图形学结课论文

基于 Whitted 的光线追踪渲染器设计与实现

Design and Implementation of a Ray Tracing Renderer Based on Whitted's Model

学　　院：软件学院

专　　业：软件工程

学生姓名：曾令腾

学　　号：21301002

指导教师：吴雨婷

北京交通大学

2024 年 6 月

中文摘要

本文实现了基于 Whitted 的光线追踪渲染器，主要使用 Python 编程语言和 Pillow、NumPy 库进行开发。光线追踪渲染器能够生成具有漫反射、反射和高光效果的三维场景并可用鼠标键盘控制视角，通过模拟光线从观察者位置到屏幕上每个像素点的传播路径，计算光线与场景中物体的交点及其光照效果。光线追踪渲染器的运行结果展示了三个球体在不同光照条件下的光影效果，验证了算法的正确性。本文介绍了算法的实现原理、数学基础、实现细节及其在简单场景中的应用。

关键词： 光线追踪；反射；高光

Abstract

This article implements a Whitted-style ray tracing renderer, primarily developed using the Python programming language and the NumPy mathematical library. The ray tracing renderer is capable of generating 3D scenes with diffuse reflection, reflection, and specular highlights and can be controlled using mouse and keyboard to change the viewpoint. By simulating the path of rays from the observer's position to each pixel on the screen, the renderer calculates the intersection points of the rays with objects in the scene and their lighting effects. The output of the ray tracing renderer demonstrates the lighting effects on three spheres under different lighting conditions, validating the correctness of the algorithm. This article covers the implementation principles of the algorithm, the mathematical foundations, the implementation details, and its application in a simple scene.

目录

1 引言

在这学期的计算机图形学课程中，通过前两次作业我初步掌握了在 VS 中基于 OpenGL 绘制几何体以及创建简单的集合体材质和光源。但是光源、光线以及由此产生的光影效果较为生硬，并不符合物理实际，所以我考虑自己研究光线追踪技术。光线追踪作为计算机图形学中重要的渲染技术，近年来随着硬件计算能力的增强和算法优化的发展，逐渐成为生成高质量真实感图像的主流方法之一。其基本原理是模拟光线从观察者位置出发，经过屏幕上每个像素点，与场景中的物体相交并计算光照效果。相较于传统的光栅化技术，光线追踪能够更精确地模拟光的传播和反射，因此在处理阴影、反射和折射等光学效果时表现更为出色。

因为我对 VS 下的 C++语言开发 OpenGL 并不熟悉，所以我基于 Python 和 NumPy、Pillow 库进行开发，参照 Whitted 经典光线追踪模型[1][3]。我构建了一个包含 3 个球体和上下部背景的简单场景，研究了光线追踪算法在不同光照条件下的表现，并考虑了漫反射、反射和高光等光照效果在图像生成中的应用。通过实验验证了光线追踪 raytrace 算法的正确性和在简单场景下的实际效果。同时，支持通过键盘鼠标移动视角，在同类 Python 光线追踪渲染器程序中有一定创新性[4]。

接下来我将分别介绍光线追踪的基本原理和数学基础、raytrace 算法的实现细节和场景构建的过程、实验结果分析并总结全文。

2 相关工作介绍

光线追踪作为计算机图形学中的重要技术，自上世纪 80 年代开始得到广泛的研究和应用。本节将对光线追踪的发展历程和代表性模型进行简要介绍，

2.1 经典光线追踪算法

Whitted 风格光线追踪：由 Whitted 于 1980 年提出，是光线追踪的经典算法之一[1]。该算法不仅考虑了漫反射和镜面反射，还增加了阴影、透明度和折射等光学效果的计算，使得生成的图像更加真实[5]。

2.2 蒙特卡罗光线追踪

路径追踪：由 Cook 等人在 1984 年提出，是一种基于蒙特卡罗方法的光线追踪技术。路径追踪通过随机选择光线路径的方式来模拟光的传播，能够处理复杂的光照效果，如全局光照、柔和阴影和次表面散射等。

2.3 加速技术

包围盒层次结构：用于加速光线与场景物体相交检测的技术。BVH 将场景中的物体组织成层次结构，通过空间分割和快速相交测试，减少了光线与物体相交的计算量，大大提高了渲染速度。

2.4 开源光线追踪引擎

LuxCoreRender：一个开源的物理渲染引擎，支持光线追踪和路径追踪等算法，提供高质量的图像渲染和灵活的渲染管线设置。

2.5 本章小结

通过对上述相关工作的回顾和分析，可以看出光线追踪技术由简单到复杂的发展历程。我旨在模仿 Whitted 光线追踪模型，基于 Python 实现一个简单的光线追踪器，建立起我对光线追踪的初步理解。

3 基于 Whitted 的光线追踪渲染器算法的 Python 实现

本章将介绍使用 Python 实现的 Whitted 光线追踪图形界面的实现细节包括方法描述、实验设置、实验结果与分析。

3.1 方法描述

基于 Whitted 光线追踪技术，利用 Python 编程语言的 Pillow 以及 NumPy 库实现了一个基本的三维场景并支持光线追踪效果渲染。

3.1.1 光线追踪算法

我使用 raytrace 函数实现了光线追踪的核心功能，包括光源照射、光线与球体交点判定与确定、光照计算以及交点颜色的叠加四个阶段[1]。

（1） 光源照射与追踪光线阶段。光线从观察者的位置沿着视线方向出发，逐像素计算光线在场景中传播的路径。在每次光线传播过程中，分别计算光线与场景中各个物体（三个球体）的交点。

（2） 光线与物体交点的判定阶段。经过 raytrace 函数计算后，使用 reduce(np.minimum, distances) 找到最近的交点，即光线与最近物体的距离，为之后的光照计算做准备。

（3） 光照计算阶段。为了模拟物体表面以及临近表面的光照效果，需要经过 light 函数计算每个物体对该交点的光照贡献，包括漫反射、镜面反射和高亮效果。漫反射负责模拟光线在粗糙（也就是非理想表面）表面上的扩散，镜面反射负责模拟光线在光滑表面（上下部背景）上的反射，而高光则负责模拟强光照射下的反射亮点。

漫反射模拟光线在粗糙表面上的扩散。根据 Whitted 的光线追踪模型漫反射的计算公式为[1][5]：

$$I_{diffus} = k_d * (L * N)$$

其中，k_d是漫反射系数，L是光线方向，N是法向量。

镜面反射负责模拟光线在光滑表面上的反射。镜面反射的计算公式为：

$$I_{specular} = k_s * (\max(0, R * V))^n$$

其中，k_{ds}是镜面射系数，R是反射光线方向，V是观察者方向（本文中为 E），n 是高光系数。

在计算漫反射光线和镜面反射光线时，反射光线和折射光线都会不断递归追踪，直到达到一定的递归深度为止。Whitted 光线追踪中折射光的计算根据斯涅耳定律[1]：

$$T = \eta D + \left(\eta(D * N) - \sqrt{1 - \eta^2(1 - (D * N)^2)}\right) * N$$

其中，T 是折射光向量，η是折射率。

（4） 颜色叠加阶段。最后，在 reduce 确定好交点后，通过 extract 和 place 函数计算光线与物体交点的光照度，将符合条件的颜色数据叠加得到该点处经过光线追踪处理的颜色 RGB 值[2]。颜色叠加时使用本学期数字图像处理的 RGB 颜色模型进行加法混色即可。

```
def raytrace(O, scene):
    """光线追踪函数"""
    distances = [s.intersect(O, D) for s in scene]
    nearest = reduce(np.minimum, distances)

    for (s, d) in zip(scene, distances):
        hit = (nearest != FARAWAY) & (d == nearest)
        if np.any(hit):
            dc = extract(hit, d)
              Oc = O.extract(hit)
              Dc = D.extract(hit)
```

```
                cc = s.light(Oc, Dc, dc, scene, bounce)
                color += cc.place(hit)

        return color
```

3.1.2 数学基础原理

实现光线追踪的数学部分主要是要处理向量的运算，就必然包括向量的合成与分解、向量的模运算等，然后将计算好的向量储存起来便于后续使用[2]。我在 vec3 类中定义了三维向量的基本运算和函数，包括向量的加减法、向量乘法和模运算等。向量的加减法主要用于光线传播路径的计算。例如，光线从观察者位置出发，经过空间中球体边缘上的某一点，我们需要计算该点的具体坐标，这就需要向量的加减法。向量的乘法通常用于光照计算，如光线强度的变化、漫反射与镜面反射光线的方向等。在 extract 函数中实现从向量中提取符合条件的像素，作为球体与光线交点的计算来源。使用 place 函数存放参与计算的向量，用于计算光线与物体交点处的颜色 RGB 叠加[3]。

```
class vec3():
    """表示三维向量的类"""

    def init(x, y, z):
        (this.x, this.y, this.z) = (x, y, z)

    def mul(self, other):
        """向量与标量乘法"""
        return vec3(self.x * other, self.y * other, self.z * other)

    def add(self, other):
        """向量加法"""
        return vec3(self.x + other.x, self.y + other.y, self.z + other.z)

    def sub(self, other):
        """向量减法"""
```

```
17.             return vec3(self.x - other.x, self.y - othe
    r.y, self.z - other.z)
18.
19.         def dot(self, other):
20.             """向量点乘"""
21.             return (self.x * other.x) + (self.y * other.
    y) + (self.z * other.z)
22.
23.         def components(self):
24.             """返回向量的各分量"""
25.             return (self.x, self.y, self.z)
26.
27.         def extract(self, cond):
28.             """根据条件 cond 从向量中提取元素"""
29.             return vec3(extract(cond, self.x),
30.                         extract(cond, self.y),
31.                         extract(cond, self.z))
32.
33.         def place(self, cond):
34.             """根据条件 cond 在数组中放置向量"""
35.             r = vec3(np.zeros(cond.shape), np.zeros(con
    d.shape), np.zeros(cond.shape))
36.             np.place(r.x, cond, self.x)
37.             np.place(r.y, cond, self.y)
38.             np.place(r.z, cond, self.z)
39.             return r
```

3.1.3 光线追踪物体设置

由于我只是对光线追踪的实现进行初步探索，所以从自己的能力和渲染正确性考量只定义 3 个球体的简单 scene，对每个球体分别设置其大小、位置、初始颜色和反射参数。我将三个球体分别设置为亮蓝色、库克紫以及灰白色，这三个颜色是漫反射状态下的颜色。

```
1. # 场景中的物体列表
2. scene = [
```

```
    Sphere(vec3(1.0, .2, 2.0), .6, rgb(0, 0, 1)),  # 蓝色球体
    Sphere(vec3(-.95, .31, 1.25), .6, rgb(.5, .0, .5)), # 库克紫球体
    Sphere(vec3(-.175, .51, 7.5), .6, rgb(1, .99, .99)), # 白色球体
]
```

3.1.4 观察视角设置

为了延续之前 OpenGL 实验中的传统，我还是加入了键鼠控制观察视角的功能。键盘的上下左右键控制 y 轴（垂直视线的平面），鼠标滚轮控制 z 轴上的远近。从而更直观地观察光线追踪算法在不同角度和距离下的效果，不仅可以展示出静态的渲染结果，还可以动态地调整视角，观察光线在不同角度下的光线追踪效果。

```
# 视角控制参数
E = vec3(0, 0.35, -1)  # 观察者（Eye）位置初始设定

# 键盘和鼠标操作可以改变观察者的视角角度
def key_pressed(event):
    global E
    if event.key == 'up':
        E.y += 0.1
    elif event.key == 'down':
        E.y -= 0.1
    elif event.key == 'left':
        E.x -= 0.1
    elif event.key == 'right':
        E.x += 0.1
    # 重新渲染图像
    render_scene()

def mouse_pressed(event):
    global E
    if event.button == 'up':
        E.z += 0.1
```

```
        elif event.button == 'down':
            E.z -= 0.1
        # 重新渲染图像
        render_scene()
```

3.2 实验设置

3.2.1 实验环境

Pycharm IDE，Python 3.12 编译器，依赖库包括 NumPy、Pillow 等。

3.2.2 场景参数

基本场景参数包括设置光源位置 L 为 vec3(5, 5, -10)，观察者位置 E=vec3(0, 0.35, -1)，使用 FARAWAY=1.0e39 表示远处的一个极大值，三个球体、四分棋盘格底部背景以及接近纯黑的上部背景，从而便于观察两种反射以及阴影是否正确。

3.3 实验结果与分析

3.3.1 实验操作过程

首先，我构建了用于光线追踪渲染的基本物体场景，包括 3 个球体和底部、上部背景色。然后直接运行结果，检查场景是否正确渲染，结果如图 3.1.

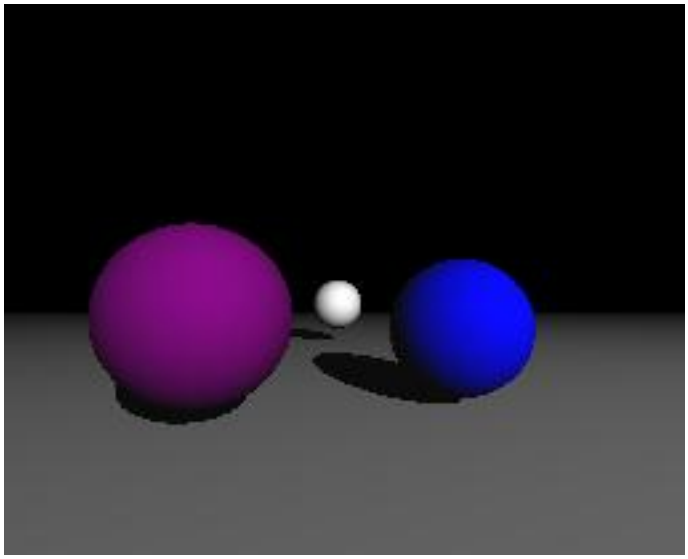

图 3.1

接下来，raytrace 函数进行光线追踪渲染光线从观察者位置 E 出发，经过屏幕上每个像素点，计算每条光线与场景中球体的交点，确定最近的交点并计算光照效果（计算包括球体的漫反射、反射和高光效应，以及阴影像素）。

3.3.2 实验结果展示与分析

（1）实验生成了一幅 scene 的基本图像，展示了场景中球体的颜色和基本光照效果，如图 3.1

（2）在基础 scene 上加入光线追踪后可以观察到 3 个球体自身的漫反射、镜面反射和与下部背景的阴影效果， 如图 3.2

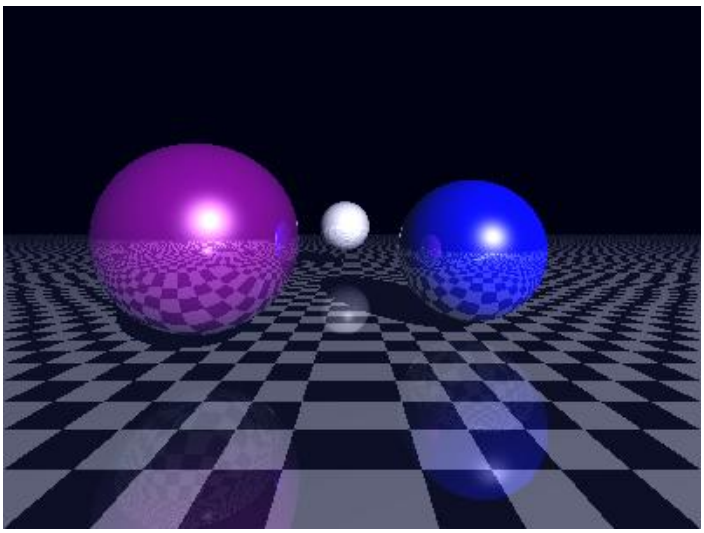

图 3.2

（3）移动观察视角，观察检验不同角度下的光线追踪效果如图 3.3

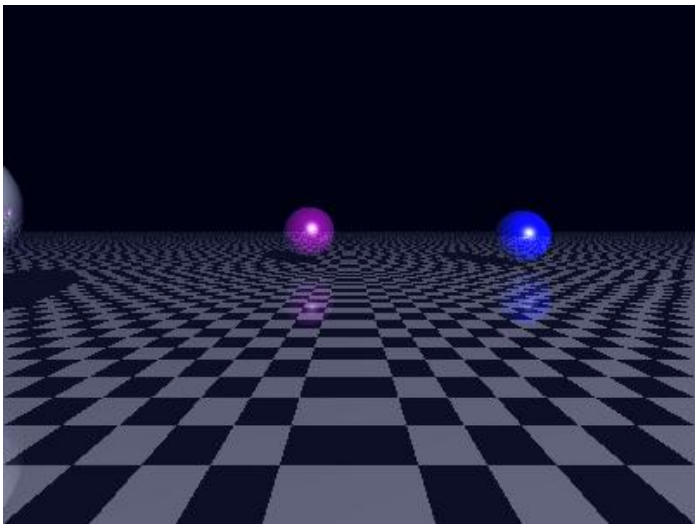

图 3.3

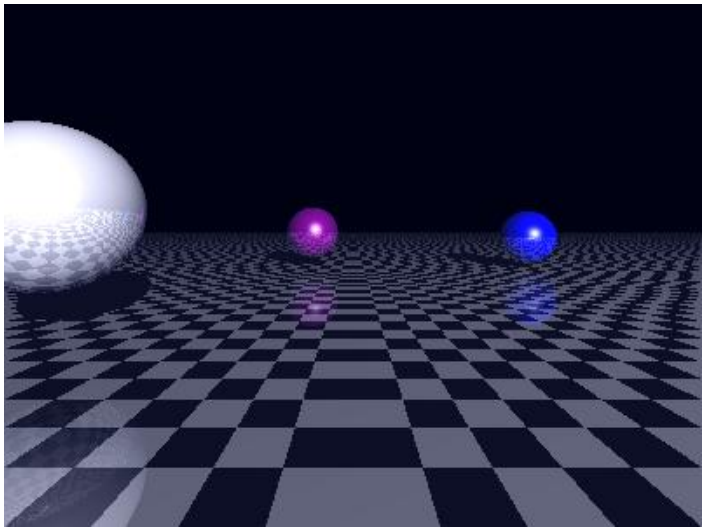

图 3.4

实验通过构建一个包含多个球体和背景色的基本场景，验证了光线追踪渲染器算法的正确性有效性。首先，通过生成基础场景图像，确保场景正确渲染。然后，通过使用（加入）光线追踪算法，计算光线与物体的交点和光照效果，展示了光线追踪下的光照现象包括漫反射、镜面反射和阴影。

图 3.1 展示了基本场景中球体的颜色和光照效果，验证了场景构建的正确性。图 3.2 展示了加入光线追踪后的效果。图 3.3 展示了不同视角下的光线追踪效果，通过移动观察视角，可以看到场景中物体在不同光线条件下的变化，可以观察到在正常视角下（y 不小于 0），三个球体之间、球体与上下部背景之间的光线（漫反射、镜面反射、阴影等）均是正常正确的，体现出了较好的基础光线追踪效果。

实验结果证明，本文的光线追踪算法可以正确地模拟较复杂的光照效果，实现了基于 Whitted 的光线追踪渲染。

4 总结与展望

我成功实现了一个基于 Whitted 光线追踪的三维场景渲染器，在 Python 环境下利用基础的数学建模和向量运算实现 Whitted 光线追踪。通过实验验证了光线追踪算法的正确性和光线追踪的效果，未来可以进一步优化算法，将非纯色背景考虑在内，从而增加场景的真实性，同时可以利用 CUDA 加速进一步提升渲染速度和图像质量，以满足更高要求的实时渲染需求。

参考文献

[1] Roger D , Assarsson U , Holzschuch N .Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU[J].Eurographics Association, 2007.DOI:10.2312/EGWR/EGSR07/099-110.

[2] CSU_XIJI, Python 手写光线追踪教程 https://blog.csdn.net/xiji333/article/details/108613678?ops_request_misc=%257B%2522request%255Fid%2522%253A%252217191431841680022746077 5%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=171914318416800227460775&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-108613678-null-null.142^v100^pc_search_result_base5&utm_term=whitted&spm=1018.2226.3001.4187

[3] Ray Tracing 1（Whitted）笔记 https://blog.csdn.net/m0_64596020/article/details/137509875?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-8-137509875-blog-108613678.235^v43^pc_blog_bottom_relevance_base9&spm=1001.2101.3001.4242.5&utm_relevant_index=11

[4] python 手写光线追踪（不使用图形学 API） https://blog.csdn.net/PythonKiki/article/details/115379049?ops_request_misc=&request_id=&biz_id=102&utm_term=python%20%E5%85%89%E7%BA%BF%E8%BF%BD%E8%B8%AA&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-1-115379049.142^v100^pc_search_result_base5&spm=1018.2226.3001.4187

[5] Whitted, T. (1980). An Improved Illumination Model for Shaded Display. Communications of the ACM, 23(6), 343-349