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）
21301010	李阳阳	9	27	17	34	87

论文规范性较好，但实验结果分析有些简单

《计算机图形学》
（24 春）期末课程论文

学　　院：软件学院

专　　业：软件工程

学生姓名：李阳阳

学　　号：21301010

光线追踪在计算机图形学中的应用

李阳阳 [1]

（1.北京交通大学软件学院，北京 100044；学号 21301010）

摘　要： 本文作为《计算机图形学》的期末课程论文，结合课程所学与文献调研，探讨了光线追踪技术在计算机图形学中的应用，阐述了光线追踪的历史与发展、基本原理以及优化算法，并自行进行了实验。着重通过实验设置和结果分析，实现了运用光线追踪算法模拟光线在不同表面间的反射和折射，验证了光线追踪算法在生成逼真图像方面的出色效果。实验结果表明，光线追踪技术能有效模拟光线在不同表面间的反射和折射，从而产生逼真的阴影和高光效果，在生成高质量渲染图像方面具有显著优势。

关键词： 图像渲染，全局光照，光线追踪

The Application of Ray Tracing in Computer Graphics

Li Yangyang[1]

（1. School of Software, Beijing Jiaotong University, Beijing 100044; Student ID 21301010)

Abstract: This article, as the final course paper of Computer Graphics, combines the course knowledge and literature research to explore the application of ray tracing technology in computer graphics. It elaborates on the history and development, basic principles, and optimization algorithms of ray tracing, and conducts experiment. By focusing on experimental settings and result analysis, we have achieved the use of ray tracing algorithms to simulate the reflection and refraction of light between different surfaces, verifying the excellent performance of ray tracing algorithms in generating realistic images. The experimental results show that ray tracing technology can effectively simulate the reflection and refraction of light between different surfaces, resulting in realistic shadows and highlights, and has significant advantages in generating high-quality rendered images.

Keywords: Image rendering, global lighting, ray tracing

目 录

1 引言

光线追踪（Ray Tracing）是一种计算机图形学中的图像渲染技术，通过模拟光线与物体的交互过程生成逼真的图像。自 1968 年 Arthur Appel 首次提出射线投射（Ray Casting）技术以来，光线追踪经历了显著的发展，特别是在 1980 年代 Turner Whitted 引入递归算法之后，光线追踪能够模拟复杂的光照效果。近年来，随着 GPU 等硬件技术的进步，实时光线追踪逐渐成为可能，广泛应用于电影、动画、游戏以及虚拟现实等领域[1]。

本文作为《计算机图形学》的期末课程论文，结合课程所学与文献调研，了解光线追踪技术，阐述光线追踪的历史与发展、基本原理，并进行了实验，实现了光线追踪算法来模拟光线在不同表面间的反射和折射，验证了光线追踪算法在生成逼真图像方面的出色效果。

2 方法描述

2.1 光线追踪的历史发展

光线追踪是一种计算机图形学中的图像渲染技术，旨在模拟光线与物体的交互过程，从而生成逼真的图像。这项技术最早由 Arthur Appel 于 1968 年提出，他开发了一种为实体线框模型渲染阴影的方法，称为“射线投射”，用于生成三维图像的阴影效果[2]。

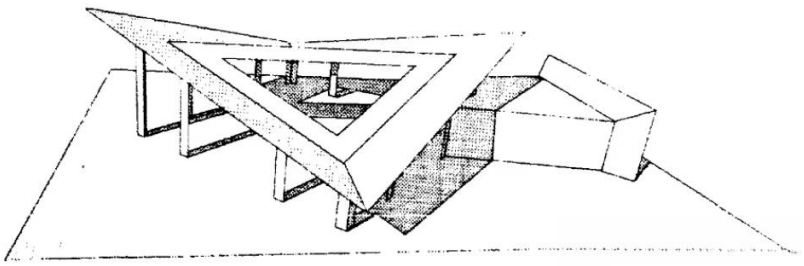

图 1 Appel 第一次为计算机渲染三维模型添加了阴影

到了 1980 年代，Turner Whitted 进一步发展了光线追踪技术，引入了递归算法，使得光线追踪能够模拟反射、折射和阴影等复杂的光照效果[3]。尽管计算复杂度高，但生成的图像更具真实感，这项研究开启了光线追踪的发展之路。

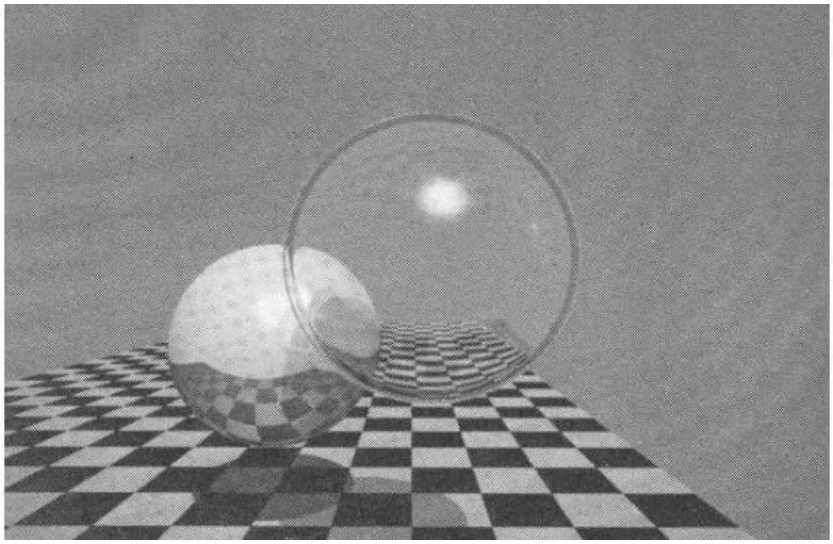

图 2 Whitted 经典递归光线追踪渲染结果[3]

1984 年，Robert L. Cook 等人[4]提出了分布式光线追踪算法(Distributed Ray Tracing)，针对运动模糊、景深、光泽反射等特殊效果的渲染提出了解决方案。

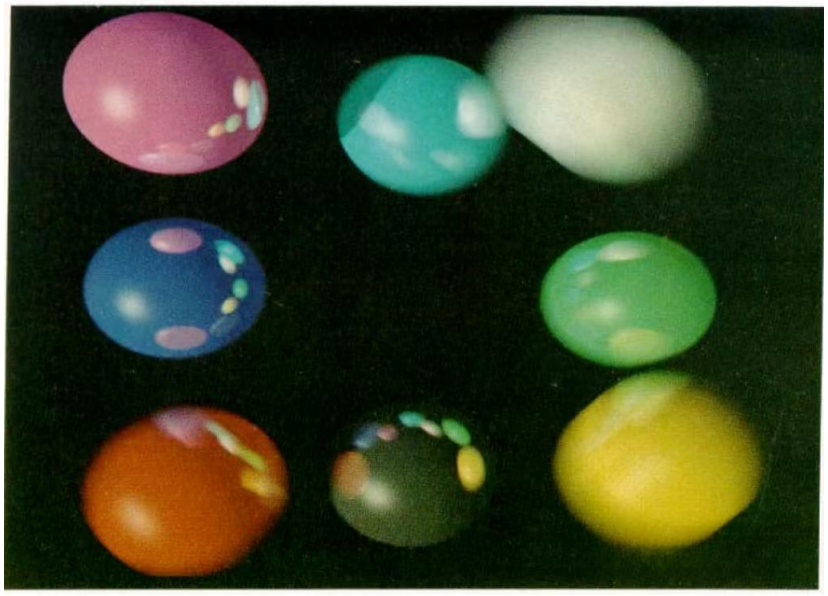

图 3 Cook 等人模拟出运动模糊等镜头效果

1986 年，Kajiya 提出了路径追踪（Path Tracing）算法[5]，通过随机采样和蒙特卡罗积分解决了全局光照问题，使得光线追踪技术在模拟复杂光照效果方面更加灵活。并提出了渲染方程，从物理学和统计学意义上建立了光线追踪渲染的科学模型，为直至今日的研究人员提供了重要的指导意义。

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int \rho(x,x',x'')I(x,x')dx''\right]$$

近年来，随着硬件技术的发展，特别是 GPU（图形处理单元）的强大并行计算能力，实时光线追踪技术得到了迅速发展。由于光线追踪能够生成高度真实感

的图像，因此被广泛应用于电影、动画、游戏以及虚拟现实等领域。

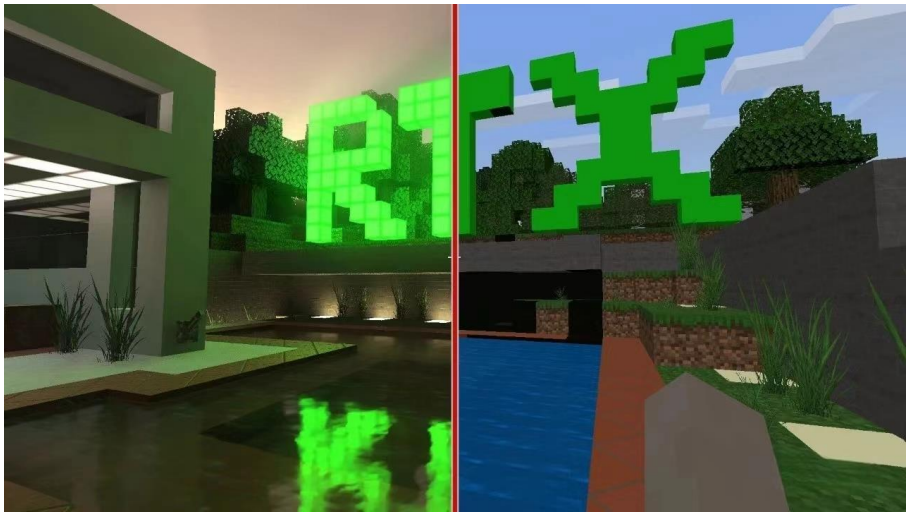

图 4 光线追踪在《我的世界》游戏中的应用

2.2 光线追踪的基本原理

光线追踪是最早的全局光照算法，是以光线投射算法[6]为基础，逐渐发展而来的。区别于光栅化渲染采用的局部光照，光线追踪则采用全局光照模型，通过物理原理对光线和物质之间的交互行为进行建模，考虑物体间相互光照影响，比传统的光栅化渲染效果更加立体，色彩更柔和更逼真。

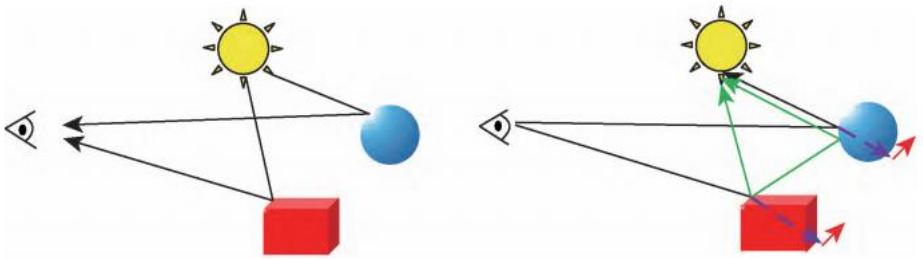

图 5 光栅化（左）与光线追踪（右）

光线追踪技术基于几何光学原理，其核心思想是模拟光线在场景中的传播路径，并计算它们在与物体交互时产生的反射、折射和散射等现象，生成逼真的图像。光线追踪通常包括以下几个主要步骤：

（1）光线发射

光线追踪过程从摄像机位置开始，沿不同方向发射光线。每条光线代表一个像素，通过计算光线在场景中的传播路径，可以确定该像素的颜色值。光线的初始方向通常根据摄像机的位置和视角进行计算。

（2）光线与物体交互

当光线与场景中的物体相交时，需要计算交点的位置和法线方向。根据物体

的材质属性，光线在交点处可能会发生反射、折射或吸收。对于反射光线，新的光线方向可以根据反射定律计算；对于折射光线，则需要根据折射定律计算新方向。

（3）递归处理

光线追踪的一个重要特性是递归处理。每条光线在与物体交互后，可能会生成新的二次光线。这个过程不断递归，直到光线的能量衰减到一定阈值，或者达到设定的最大递归深度。

（4）颜色计算

每条光线最终会到达一个光源，或者在多次反射和折射后消耗掉能量。通过累积所有光线在每个像素上的贡献，可以计算出该像素的最终颜色值。这个过程通常需要考虑光线在传播过程中经历的光照衰减、阴影、全局光照等因素。

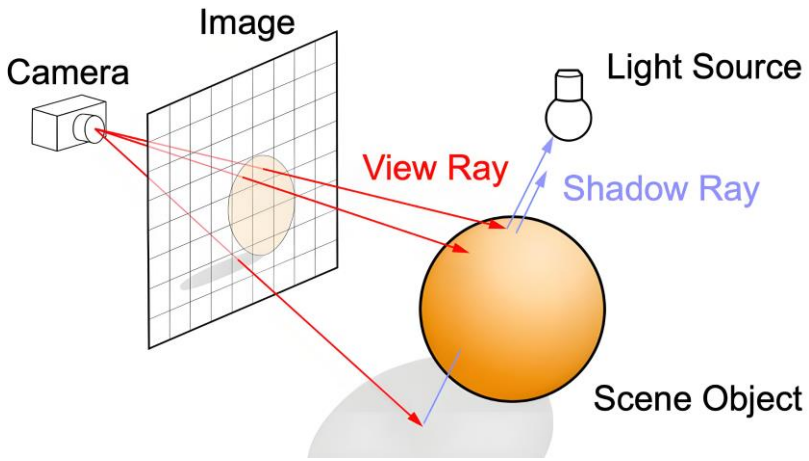

图 6 光线追踪原理示意图

用经典光线追踪算法模拟真实场景中的光线传播，需要的光线样本数量大，且每根光线的传播路径都需要大量的计算。该算法的缺点是性能消耗大，难以实现复杂场景的快速计算。随着计算机硬件和算法的不断进步与计算机图形学研究的多元化，更多优化技术也相继提出，不断提高图像渲染的质量和效率[7]。

3 实验设置

3.1 材质

首先需要设置材质，材质类型包括粗糙材质、反射材质和折射材质，材质参数包括环境光照、漫反射、镜面反射系数等。在实验中，反射材质采用 Fresnel 公式[8]计算光的反射和折射率。实现代码如图 7 所示。

```
struct Material
{
    vec3 ka, kd, ks; // 环境光照(Ambient)，漫反射(Diffuse)，镜面反射(Specular)系数，用于phong模型计算
    float shininess; // 镜面反射强度（或者理解为物体表面的平整程度）
    vec3 F0;         // 反射比。计算公式需要折射率n和消光系数k：[(n-1)^2+k^2]/[(n+1)^2+k^2]
    float ior;       // 折射率（index of refraction）
    MaterialType type;
    Material(MaterialType t) { type = t; }
};
```

图 7 定义材质

3.2 光源、光线和交点

随后，实验设置了光源、光线与交点。光线的定义包含光照强度、起点与方向，实现代码如图 8 所示。

```
struct Light
{ // 定义光源
    vec3 direction;
    vec3 Le; // 光照强度
    Light(vec3 _direction, vec3 _Le)
    {
        direction = _direction;
        Le = _Le;
    }
};
```

```
struct Ray // 射线（从视角出发追踪的射线）
{
    vec3 start, dir; // 起点，方向
    Ray(vec3 _start, vec3 _dir)
    {
        start = _start;
        dir = normalize(_dir); // 方向要进行归一化
    }
};
```

图 8 定义光源与光线

接着是关于光线与交点的实验设置。首先要判断光线和物体是否有交点，将射线方程设置为 V(t)=S+td , $0 \le t < \infty$。平面方程为: $(p - p') \cdot N = 0$。判断射线是否与平面相交，即判断是否有存在 t1，$0 \leqslant t1 \leqslant \infty$，使得点 p =V (t1)在平面上。假设 p = V (t)， 对下面方程求解 t。

联立射线方程与平面方程：$\begin{cases} V(t) = S + td \\ p = V(t) \\ (p - p') \cdot N = 0 \end{cases}$

得 $t = \frac{(p'-S)\cdot N}{d\cdot N}$　　Check：$0 \le t < \infty$

因此，当 t 大于 0 时表示相交，默认取-1 表示无交点。确定射线与平面相交后，再求得其交点 p 坐标、法线，并判断该点是否在三角形内，最后设置交点处

表面的材质。实现代码如图 8 所示。

```
struct Hit // 光线和物体表面交点
{
    float t;                // 直线方程v=s+td。当t大于0时表示相交，默认取-1表示无交点
    vec3 position, normal; // 交点坐标，法线
    Material *material;    // 交点处表面的材质
    Hit() { t = -1; }
};
```

图 8 定义光源、光线和交点

3.3 球体

在实验中，使用球体作为反射光线的物体。定义球体的中心位置和半径以及材质。函数 intersect 用来计算一条光线与球体的交点，通过联立光线方程和球体方程求解交点参数 t。当判别式小于 0，表示光线没有与球体相交，否则计算两个交点 t1 和 t2，并选择距离较近的交点，设置交点对象的属性并返回。实现代码如图 9 所示。

```
class Sphere : public Intersectable // 定义球体
{
    vec3 center;
    float radius;

public:
    Sphere(const vec3 &_center, float _radius, Material *_material)
    {
        center = _center;
        radius = _radius;
        material = _material;
    }
    ~Sphere() {}

    Hit intersect(const Ray &ray)
    {
        Hit hit;
        vec3 dist = ray.start - center;  // 距离
        float a = dot(ray.dir, ray.dir); // dot表示点乘，这里是联立光线与球面方程
        float b = dot(dist, ray.dir) * 2.0f;
        float c = dot(dist, dist) - radius * radius;
        float discr = b * b - 4.0f * a * c; // b^2-4ac
        if (discr < 0)                      // 无交点
            return hit;
        float sqrt_discr = sqrtf(discr);
        float t1 = (-b + sqrt_discr) / 2.0f / a; // 求得两个交点，t1 >= t2
        float t2 = (-b - sqrt_discr) / 2.0f / a;
        if (t1 <= 0)
            return hit;
        hit.t = (t2 > 0) ? t2 : t1; // 取近的那个交点
        hit.position = ray.start + ray.dir * hit.t;
        hit.normal = (hit.position - center) / radius;
        hit.material = material;
        return hit;
    }
```

图 9 定义球体并计算光线与球体的交点

3.4 圆柱体

同时在场景中还使用了圆柱体。定义圆柱体类 Cylinder，包含不同表示圆柱体的参数。和球体类似，函数 intersect 用于计算光线与圆柱的交点。对于无限长圆柱，选择最近的交点。对于有限长圆柱，还需验证交点是否在上下底面之间，同时检查光线与上下底面的交点，最终选择最近的有效交点，设置交点对象的属性并返回。实现代码如图 10 所示。

```
class Cylinder : public Intersectable
{ // 无(有)限长圆柱面，(q-pa-(va,q-pa)va)^2-r^2=0，q是面上一点
    // (va,q-pa)，是点乘
    vec3 va;      // 转轴方向
    vec3 pa;      // 转轴中心点（或者理解成某一截面的中心）
    float radius; // 旋转半径
    vec3 p1;      // 下底面圆心，有(va,q-p1)>0，这个参数不给就是无限长圆柱
    vec3 p2;      // 上底面圆心，有(va,q-p2)<0，这个参数不给就是无限长圆柱

public:
    Cylinder(vec3 _pa, vec3 _va, float _radius, Material *_material, vec3 _p1 = vec3(0, 0, 0), vec3 _p2 = vec3(0, 0, 0))
    {
        pa = _pa;
        va = normalize(_va);
        radius = _radius;
        material = _material;
        p1 = _p1; // 如果为(0,0,0)则说明用户定义无限长圆柱面
        p2 = _p2;
    }

    Hit intersect(const Ray &ray)
    {
        Hit hit;
        vec3 A_operand = ray.dir - dot(ray.dir, va) * va;
        float A = dot(A_operand, A_operand);
        vec3 delta_p = ray.start - pa;
        vec3 B_operand = delta_p - dot(delta_p, va) * va;
        float B = 2 * dot(A_operand, B_operand);
        float C = dot(B_operand, B_operand) - radius * radius;
        float discr = B * B - 4 * A * C;
        if (discr < 0)
            return hit;

        float sqrt_discr = sqrtf(discr);
        float t1 = (-B + sqrt_discr) / 2 * A; // t1>=t2
        float t2 = (-B - sqrt_discr) / 2 * A;

        if (p1 == vec3(0, 0, 0) || p2 == vec3(0, 0, 0))
        {
            if (t1 < 0)
                return hit;

            hit.t = (t2 > 0) ? t2 : t1; // 取近的那个交点
            hit.position = ray.start + ray.dir * hit.t;
            vec3 N = hit.position - pa - dot(va, hit.position - pa) * va;
            hit.normal = normalize(N);
            hit.material = material;
            return hit;
        }
```

图 10 定义圆柱并计算光线与圆柱的交点

3.5 相机

光线追踪不是从光源发出光线，而是从眼睛发出光线。因此，定义了一个相机类 Camera，用于表示用户视线。定义包含相机位置、视线方向、右方向和上方向，以及视场角。

运用 set 方法根据传入的参数初始化相机的位置和视角，并计算视窗大小；

运用 getRay 方法根据屏幕像素坐标 X 和 Y 生成从相机位置出发的光线；运用 Animate 方法通过旋转相机位置来模拟动画效果，并更新相机的视角参数。实现代码如图 11 所示。

```
class Camera
{                                   // 用相机表示用户视线
    vec3 eye, lookat, right, up; // eye用来定义用户位置；lookat(视线中心)，right和up共同定义了视窗大小
    float fov;

public:
    void set(vec3 _eye, vec3 _lookat, vec3 _up, float _fov) // fov视场角
    {
        eye = _eye;
        lookat = _lookat;
        fov = _fov;
        vec3 w = eye - lookat;
        float windowSize = length(w) * tanf(fov / 2);
        right = normalize(cross(_up, w)) * windowSize; // 要确保up、right与eye到lookat的向量垂直(所以叉乘)
        up = normalize(cross(w, right)) * windowSize;
    }

    Ray getRay(int X, int Y)
    {
        vec3 dir = lookat + right * (2 * (X + 0.5f) / windowWidth - 1) + up * (2 * (Y + 0.5f) / windowHeight - 1) - eye;
        return Ray(eye, dir);
    }

    void Animate(float dt) // 修改eye的位置（旋转）
    {
        vec3 d = eye - lookat;
        eye = vec3(d.x * cos(dt) + d.z * sin(dt), d.y, -d.x * sin(dt) + d.z * cos(dt)) + lookat;
        set(eye, lookat, up, fov);
    }
};
```

图 11　定义相机及参数

3.6 场景

定义了一个场景类，包含物体、光源、光线和相机，实现代码如图 12 所示。在构建场景的时候，设置相机属性并初始化。然后定义光源，并将光源添加到 lights 中。定义镜面反射系数，创建了一个镜面反射球、一个漫反射球、一个有限长的圆柱体和一个反射平面加入到场景中，指定它们的材质和位置，将它们添加到 objects 列表中。

```
class Scene
{ // 场景，物品和光源集合
    std::vector<Intersectable *> objects;
    std::vector<Light *> lights;
    Camera camera;
    vec3 La; // 环境光
public:
    void build()
    {
        vec3 eye = vec3(0, 0, 4), vup = vec3(0, 1, 0), lookat = vec3(0, 0, 0);
        float fov = 45 * M_PI / 180;
        camera.set(eye, lookat, vup, fov);

        La = vec3(0.4f, 0.4f, 0.4f);
        vec3 lightDirection(1, 1, 1), Le(2, 2, 2);
        lights.push_back(new Light(lightDirection, Le));

        vec3 ks(2, 2, 2);

        objects.push_back(new Sphere(vec3(-0.85, 0, 0), 0.5, new ReflectiveMaterial(vec3(0.14, 0.16, 0.13), vec3(4.1, 2.3, 3.1)))); // 镜面反射球

        // objects.push_back(new Cylinder(vec3(0.55,0,0), vec3(0,1,0), 0.5f, new RoughMaterial(vec3(0.1, 0.2, 0.3), ks, 100)));   // 无限长圆柱

        objects.push_back(new Sphere(vec3(0, 0.5, -0.8), 0.5, new RoughMaterial(vec3(0.3, 0, 0.2), ks, 20))); // 漫反射球

        objects.push_back(new Cylinder(vec3(0.55, 0, 0), vec3(0, 1, 0), 0.4f, new RoughMaterial(vec3(0.1, 0.2, 0.3), ks, 100), vec3(0.55, -0.1, 0), vec3(0.55, 0.4, 0))); // 有限长圆柱

        objects.push_back(new Plane(vec3(0, -0.6, 0), vec3(0, 1, 0), new ReflectiveMaterial(vec3(0.14, 0.16, 0.13), vec3(4.1, 2.3, 3.1))));
    }
```

图 12　定义场景类

3.7 光线追踪

定义 trace 方法用来实现光线追踪算法。

首先进行递归深度检查，若递归深度大于 5，则返回环境光，防止无限递归。然后计算光线与场景中物体最近的交点，若没有交点，也返回环境光 La。

对于粗糙的物体材质，初始化反射光之后，遍历场景中的每个光源，计算阴影光线，计算光线方向与物体表面法线之间的角度余弦值 cosTheta，若大于 0 则表示光线照射到物体的正面，然后检查光线到光源的路径上是否有遮挡物，如果没有遮挡物则按 Phong 模型计算光线的漫反射和镜面反射，实现代码如图 13 所示。

```
vec3 trace(Ray ray, int depth = 0) // 光线追踪
{
    if (depth > 5) // 设置迭代上限5次
        return La;
    Hit hit = firstIntersect(ray);
    if (hit.t < 0) // 不再有交，则返回环境光即可
        return La;

    if (hit.material->type == ROUGH)
    {
        vec3 outRadiance = hit.material->ka * La; // 初始化返回光线（或者说阴影）
        for (Light *light : lights)
        {
            Ray shadowRay(hit.position + hit.normal * epsilon, light->direction);
            float cosTheta = dot(hit.normal, light->direction);
            if (cosTheta > 0) // 如果cos小于0（钝角），说明光找到的是物体背面，用户看不到
            {
                if (!shadowIntersect(shadowRay)) // 如果与其他物体有交，则处于阴影中；反之按Phong模型计算
                {
                    outRadiance = outRadiance + light->Le * hit.material->kd * cosTheta;
                    vec3 halfway = normalize(-ray.dir + light->direction);
                    float cosDelta = dot(hit.normal, halfway);
                    if (cosDelta > 0)
                        outRadiance = outRadiance + light->Le * hit.material->ks * powf(cosDelta, hit.material->shininess);
                }
            }
        }
        return outRadiance;
    }
```

图 13 Phong 模型计算光线的漫反射和镜面反射

对于镜面反射的物体材质，计算反射比 F 和反射光线的方向，递归调用 trace 方法追踪反射光线，将返回的颜色乘以 F 后加入到 outRadiance 中。

对于折射的物体材质，计算折射比 disc 和折射光线的方向，递归调用 trace 方法追踪折射光线，将返回的颜色乘以 (one - F) 后加入 outRadiance。

最后返回 outRadiance，完成光线追踪过程，如图 14 所示。

```
vec3 trace(Ray ray, int depth = 0) // 光线追踪
{
    if (depth > 5) // 设置迭代上限5次
        return La;
    Hit hit = firstIntersect(ray);
    if (hit.t < 0) // 不再有交，则返回环境光即可
        return La;

    if (hit.material->type == ROUGH)
    {
        vec3 outRadiance = hit.material->ka * La; // 初始化返回光线（或者说阴影）
        for (Light *light : lights)
        {
            Ray shadowRay(hit.position + hit.normal * epsilon, light->direction);
            float cosTheta = dot(hit.normal, light->direction);
            if (cosTheta > 0) // 如果cos小于0（钝角），说明光找到的是物体背面，用户看不到
            {
                if (!shadowIntersect(shadowRay)) // 如果与其他物体有交，则处于阴影中；反之按Phong模型计算
                {
                    outRadiance = outRadiance + light->Le * hit.material->kd * cosTheta;
                    vec3 halfway = normalize(-ray.dir + light->direction);
                    float cosDelta = dot(hit.normal, halfway);
                    if (cosDelta > 0)
                        outRadiance = outRadiance + light->Le * hit.material->ks * powf(cosDelta, hit.material->shininess);
                }
            }
        }
        return outRadiance;
    }

    float cosa = -dot(ray.dir, hit.normal); // 镜面反射（继续追踪）
    vec3 one(1, 1, 1);
    vec3 F = hit.material->F0 + (one - hit.material->F0) * pow(1 - cosa, 5);
    vec3 reflectedDir = ray.dir - hit.normal * dot(hit.normal, ray.dir) * 2.0f; // 反射光线R = v + 2Ncosa
    vec3 outRadiance = trace(Ray(hit.position + hit.normal * epsilon, reflectedDir), depth + 1) * F;

    if (hit.material->type == REFRACTIVE) // 对于透明物体，计算折射（继续追踪）
    {
        float disc = 1 - (1 - cosa * cosa) / hit.material->ior / hit.material->ior;
        if (disc >= 0)
        {
            vec3 refractedDir = ray.dir / hit.material->ior + hit.normal * (cosa / hit.material->ior - sqrt(disc));
            outRadiance = outRadiance + trace(Ray(hit.position - hit.normal * epsilon, refractedDir), depth + 1) * (one - F);
        }
    }
    return outRadiance;
}
```

图 14 实现光线追踪

4 实验结果与分析

渲染效果如图 14 所示，运行程序后，三个物体绕着中心旋转，显示出动态的光线追踪效果。

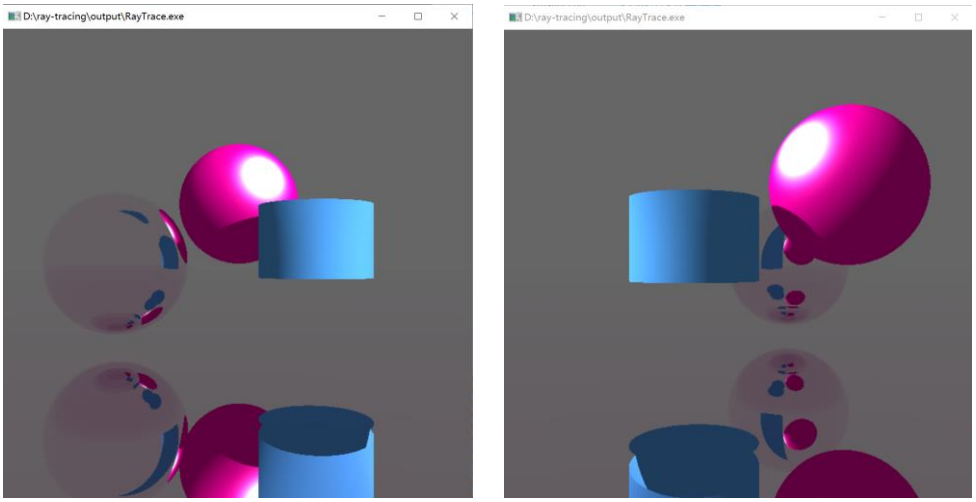

图 14 光线追踪渲染效果图

从运行结果中可以看出，反射的透明球体能够清晰地反映出周围环境和其他物体的颜色和形状，说明程序对光线正确的进行了反射。漫反射球体在光源方向表现出亮点，而其他部分则较为暗淡，光线在漫反射表面上均匀散射。圆柱体垂直于光源，展示了较为均匀的光照分布，其边缘部分稍显暗淡。地面上的反射效果清晰地显示了物体的镜像，也说明反射光线得到了正确的计算。

5 总结与展望

本次实验成功实现了光线追踪算法。实验结果显示，粉色球体和蓝色圆柱体在光线照射下展现出真实的反光与折射效果，透明球体显示了多层次的光线反射，表明实现的光线追踪算法能有效模拟光线在不同表面间的反射和折射，从而产生逼真的阴影和高光效果。

通过本次课程实验，我对计算机图形学中的光线追踪技术有了较为全面的了解。通过将理论知识应用于实际操作，我不仅加深了对该领域的理解，还提高了自己的实践能力，收获颇多。此外，我切身体会到光线追踪在生成高质量渲染图像方面的优势，相信随着硬件和算法的不断进步，光线追踪技术的应用前景将更加广阔。

6 参考文献

[1] 黄杨昱. 基于实时全局光照的 3D 绘制引擎研究和开发 [D]; 北京化工大学.

[2] APPEL A. Some techniques for shading machine renderings of solids [J]. AFIPS, 1968: 1-6.

[3] WHITTED T. An improved illumination model for shaded display [J]. Communications of the ACM, 1980: 1-6.

[4] COOK R L, PORTER T, CARPENTER L. Distributed ray tracing [J]. ACM SIGGRAPH Computer Graphics, 1984: 1-6.

[5] KAJIYA J T. The rendering equation [J]. ACM SIGGRAPH Computer Graphics, 1986: 1-6.

[6] MALOCA P M, DE CARVALHO J E R, HEEREN T, et al. High-Performance Virtual Reality Volume Rendering of Original Optical Coherence Tomography Point-Cloud Data Enhanced With Real-Time Ray Casting [J]. Translational Vision Science & Technology, 2018, 7(4).

[7] 姚晔. 基于光线追踪的阴影和反射渲染算法研究 [D], 2023.

[8] ZHI-WEI Z, ZHI-FANG W U, TING-DUN W. A Simple Model for Measuring Refractive Index of a Liquid Based upon Fresnel Equations [J]. 中国物理快报：英文版, 2007, 24(11): 4.