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）
21301005	段初凡	9	27	16	33	85

论文规范性较好，但实验结果分析有些简单

计算机图形学课程论文

一个基于OpenGL的简单光线追踪demo的设计与实现

Design and Implementation of a Simple Ray Tracing Demo Based on OpenGL

学　　院：软件学院

专　　业：软件工程

学生姓名：段初凡

学　　号：21301005

指导教师：吴雨婷

北京交通大学

2024 年 9 月

中文摘要

光线追踪（Ray Tracing）技术是计算机图形学中一种模拟光线传播的特殊渲染技术，通过追踪光线在场景中的路径，计算光与物体的相互作用，从而生成高度逼真的图像。该技术在近几年内发展迅猛，受到了越来越多的研究与关注。本文旨在通过对光线追踪技术的研究，以及对国内外该领域现有研究成果的学习，完成一个使用光线追踪技术的简单 OpenGL 项目 demo，其可以展示一些光线追踪技术的基本应用，如局部光照明、物体之间的镜面反射和阴影等，目前以上功能均可成功实现。

关键词：光线追踪；计算机图形学；OpenGL；局部光照；反射；阴影

Abstract

Ray Tracing technology is a specialized rendering technique in computer graphics that simulates the propagation of light. By tracing the paths of rays within a scene and calculating their interactions with objects, it produces highly realistic images. This technology has experienced rapid development in recent years and has garnered increasing research and attention. This article aims to study Ray Tracing technology and learn from existing research results in this field both domestically and internationally. As part of this study, a simple OpenGL project demo utilizing Ray Tracing technology will be completed. This demo can showcase some basic applications of Ray Tracing, such as local illumination, specular reflection between objects, and shadows. All these functionalities have been successfully implemented.

Keywords: Ray Tracing; Computer Graphics; OpenGL; Local Illumination; Reflection; Shadow

目　录

1. 引言

光线追踪技术在近些年取得了显著的发展，成为计算机图形学领域的一个重要里程碑，受到了业界广泛的关注与研究，其也被认为是几个最有潜力的计算机图形学技术之一。这是一种模拟光线传播的特殊渲染技术，通过追踪光线在场景中的路径，计算光与物体的相互作用，从而生成高度逼真的图像。与传统的光栅化渲染方法相比，光线追踪可以更真实地表现光影效果、反射、折射和阴影，使得生成的图像更接近现实世界。

近年来，光线追踪技术的发展主要体现在硬件和软件的进步上。硬件方面，英伟达（NVIDIA）近年革命性地更新了其旗下的当家产品 GeForce GTX 系列显卡，变为能够大幅提升光线追踪计算性能的 GeForce RTX 系列显卡，其集成了 RT Core 和 Tensor Core，大幅提升了光线追踪的计算效率。软件方面，图形 API 如 DirectX 12 和 Vulkan 都增加了对光线追踪的支持，使得开发者能够更方便地在游戏和其他图形应用中实现光线追踪效果。游戏行业也是光线追踪发展中不可不谈的一大受益者，飞速发展的光线追踪技术频频被利用在近年来的游戏大作中，如《赛博朋克 2077》、《生化危机 4RE》以及《原子之心》等等。这项革命性技术的应用使游戏画面的真实感与沉浸感大幅提升，对于近年来隐隐触碰到瓶颈的游戏技术发展来说无疑是一剂强心针。

总而言之，光线追踪技术在计算机图形学领域是不可多得的新鲜血液，本文旨在通过研究光线追踪技术以及利用其设计并完成一项简单的 demo，来加深对光线追踪的理解和学习。

2. 相关工作介绍

近年来，在光线追踪领域，国内外都出现了大量研究工作及成果，本章将分国内研究现状及成果与国外研究现状及成果进行介绍。

2.1. 国外研究现状及成果

以最受瞩目的 NVIDIA 公司为代表，近年来全球光线追踪领域的研究呈现出一种爆发式的增长。近日，该公司市值一跃超过微软与苹果，位列全球第一，这与其在在光线追踪技术的研究和应用方面取得的显著成果也有一定联系。硬件上，NVIDIA 推出了专门用于光线追踪计算的 RTX 系列显卡，这些显卡集成了 RT Core 和 Tensor Core，显著提高了光线追踪的计算效率。RTX 显卡的推出使得实时光线追踪成为可能，大大提升了图形渲染的逼真度和细节表现。

在软件方面，NVIDIA 推出了 OptiX 光线追踪引擎，并在其图形 API 中增加了对光线追踪的支持，如 DirectX 12 和 Vulkan。OptiX 利用 NVIDIA 的 GPU 加速光线追踪计算，广泛应用于游戏、影视、虚拟现实等领域，推动了这些行业的视觉效果和用户体验的提升。

此外，NVIDIA 还积极参与光线追踪标准的制定，与行业内的其他公司和机构合作，共同推动光线追踪技术的发展和普及。NVIDIA 的研究成果不仅体现在硬件和软件的改进上，还通过发布各种技术文档和开发工具，帮助开发者更好地应用光线追踪技术。

同时，国外也有一大批学者投身光线追踪领域的研究，为该领域持续注入新鲜活力，整个行业呈现出良好的发展势头，极具资本市场青睐。

2.2. 国内研究现状及成果

虽然国内光线追踪领域的研究起步较晚，相较国外目前较为成熟的研究现状也存在一定差距，但各大研究机构、公司以及众多学者仍然产出了很多具有研究价值的成果。

国防科技大学的闫润,黄立波,郭辉等学者完成了实时光线追踪相关研究综述，一定程度上巩固了该领域研究的基础[1]；在此基础上，大量学者进行了进一步的

研究，例如李家振,纪庆革完成了一种动态低采样环境光遮蔽的实时光线追踪分子渲染方法,其中提出了光线追踪中简易的重投影方法,用于实现动态低采样环境光遮蔽的时间性降噪;以及提出了阴影光线包装策略,改进了光线遍历场景时的计算并行度[2]；王平,李益文,乔磊等人则对基于加速结构和 CUDA 的光线追踪算法进行了深入研究[3]；此外，也有一些学者试图将光线追踪应用到其他领域，如张初旭针对有源系统光线在传输过程中因辐射产生的聚焦问题,引入光线追踪的方法,并结合辐射换热计算的理论,研究了聚焦点产生的温度及其分布区域,并通过实例及模拟结果的对比分析,验证了该方法的可行性[4]。

总体来看，国内光线追踪领域的研究仍处于起步阶段，但发展前景与潜力都不可忽视。

3. 实验设置

3.1. 实验环境

本实验 Demo 基于 C++实现完整的含递归调用的光线追踪算法。

在 Clion 平台下结合 OpenGL 开发

操作系统：Windows10

处理器：AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz

内存：16.00GB

系统类型：64 位操作系统

3.2. 实验原理

3.2.1. 光线追踪的基本原理

1. 渲染图象的组成

假设在相机前面放置一个网格平面，称为视平面(View plane)，网格平面中的每一个小格，就是渲染图像中的一个像素，小网格的多少有渲染输出图像的分辨率决定，如渲染输出图像的分辨率为 800*600，则此网格平面就由 800*600 的小网格组成，如果从相机的位置去看小网格，每一个小网格都覆盖了场景中的一小块区域。可见，如果能计算出每个小网格所覆盖区域的平均颜色，并将此颜色做为小网格的颜色，对小网格进行填充，将网格平面中的所有小格都填充完，也就得到了所需要的渲染结果（如图 3.2.1.1 所示）。

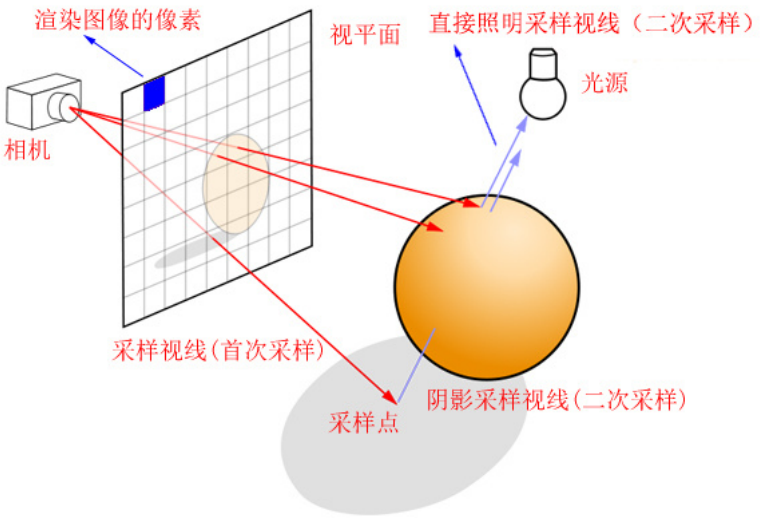


图 3.2.1.1　　示意图 1

2.　计算这些小网格的平均颜色

以相机的中点为起点，向小网格的中点发出一条辅助射线(Ray)，此射线与场景中的物体相交(如没有相交，则视为与背景相交)，如果计算出此交点的颜色，也就得到了小网格的颜色。从相机发出的辅助射线与视线方向相同，与场景中物体反射到眼睛中的光线的方向相反，故应称为视线，为了方便说明，将此辅助射线，称为采样视线，辅助射线与场景的交点，称为采样点。采样点的颜色由采样点所在物体的材质、场景中的光源，场景中的其它物体及背景等多方面因素相互作用决定的。

除了需要计算采样点在光源的直接照射下，所产生的颜色外：

如果采样点的材质具有反射属性，则需计算出采样点的反射颜色。

如果采样点的材质具有折射属性，则需计算出采样点的折射颜色。

如果采样点与光源之间有其它物体，则需要计算出采样点的阴影颜色。

如果采样点的周边有其它物体，还需要计算其它物体对此采样点所产生的间接照明效果。

如果开启了焦散效果，还需要计算出采样点的焦散颜色。

如果开启了相机的景深及运动模糊效果，还需要计算出采样点的相关模糊颜色。

将上述采样点的所有颜色综合在一起，就会得到采样点的最终颜色，

可见采样点的的最终颜色包含了许多种不同属性的颜色成分。

3. 如何计算采样点不同属性的颜色成分？

(1)采样点直接照明颜色的求法

从采样点向光线发出采样视线，求出光源与采样点的位置关系，根据光源的亮度、颜色等参数再结果采样的材质属性，就可以求出采样点在光源直接照明下所产生的颜色。

(2)采样点反射颜色的求法

如果采样点的材质具有反射属性，根据光线的反射原理，此采样点继续发出采样视线，去与场景中的物体相交，将新的交点称为二次采样点，求出二次采样点的颜色，就是此采样点反射的颜色。如果二次采样点还具有反射属性，则此采样点继续重复上面的采样计算，直到所规定的反射次数，或反射颜色减弱到一定阀值后终止。

(3)采样点折射颜色的求法

如果采样点的材质具有透明属性，根据光线的折射原理，此采样点继续发出采样视线，去与场景中的物体相交，将新的交点称为二次采样点，求出二次采样点的颜色，就是此采样点反射的颜色。如果二次采样点还具有透明属性，则此采样点继续重复上面的采样计算，直到所规定的折射次数，或折射颜色减弱到一定阀值后终止。

(4)采样点阴影颜色的求法

从采样点向光线求出阴影采样视线，如果光源与采样点间有物体遮挡，则根据光源的阴影参数及遮挡物体物属性，就可以计算出采样点的阴影颜色。光线追踪，简单地说，就是从摄影机的位置，通过影像平面上的像素位置(比较正确的说法是取样(sampling)位置，发射一束光线到场景，求光线和几何图形间最近的交点，再求该交点的颜色。如果该交点的材质是反射性的，可以在该交点向反射方向继续追踪。光线追踪除了容易支持一些全局光照效果外，亦不局限于三角形作为几何图形的单位。任何几何图形，能与一束光线计算交点(intersection point)，就能支持。

从视点出发向屏幕上每一个像素发出一条光线，追踪此光路并计算其逆向光线的方向，映射到对应的像素上。如图 3.2.1.2，通过计算光路上颜色衰

减和叠加，即可基本确定每一个像素的颜色。

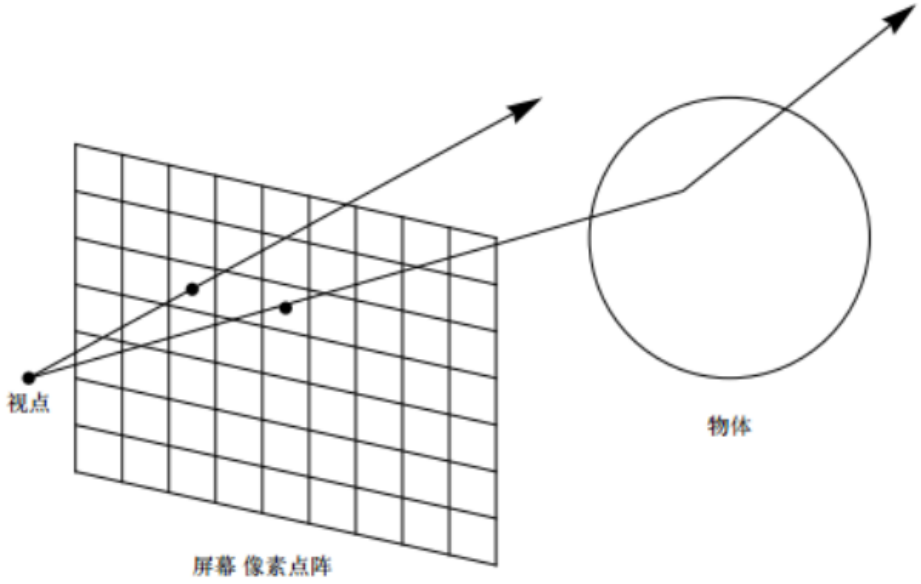


图 3.2.1.2　　示意图 2

3.2.2.　光线与物体相交具体原理

1. 光线的数学表达式

一条光线实际上只是一个起点和一个传播方向，因此光线表达式为：

$$\boldsymbol{p}(\mathrm{t}) = \boldsymbol{e} + t\,(\boldsymbol{s}\text{-}\boldsymbol{e})$$

具体的参数意义如图 3.2.2.1 所示。所以给定 t，可以确定点 p。

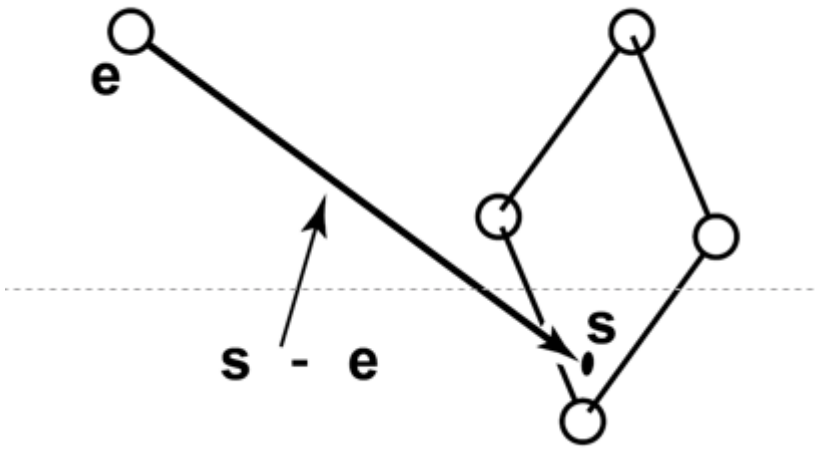


图 3.2.2.1　　光线表达式示意图

2. 光线与球相交

已知球体的隐式方程为：

$$(x-x_c)^2+(y-y_c)^2+(z-z_c)^2-R^2=0$$

$$(p-c)\cdot(p-c)-R^2=0$$

把光线表达式代入上述方程，得到：

$$(e+td-c)\cdot(e+td-c)-R^2=0$$

求解得到：

$$t=\frac{-d\cdot(e-c)\pm\sqrt{(d\cdot(e-c))^2-(d\cdot d)((e-c)\cdot(e-c)-R^2}}{(d\cdot d)}$$

故已知 t，即可确定交点 p。

3. 光线与平面相交

平面在空间几何中可以用一个向量(法向量)和平面中的一点 P0 来表示。

$$n\cdot(p-p0)=0$$

将光线

$$p(t)=p0+tu$$

代入平面方程

$$n\cdot p+d=0$$

最后求得 t：

$$t=\frac{-d-(n\cdot p0)}{n\cdot u}$$

故已知 t，可以确定交点 p。

3.2.3. 反射与折射具体原理

1. 反射光线的计算

镜面反射光线的表达式为（法向量朝外）：

$$r = d - 2(d \cdot n) \cdot n$$

反射光线的原理如图 3.2.3.1：

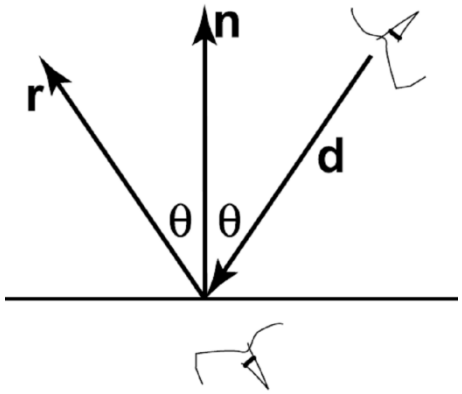


图 3.2.3.1　　反射光线

2. 折射光线的计算

折射光线的原理如图 3.2.3.2：

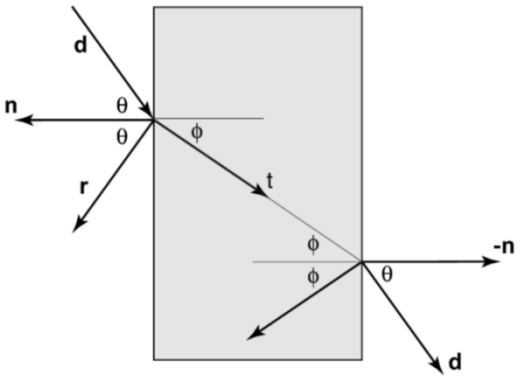


图 3.2.3.2　　折射光线

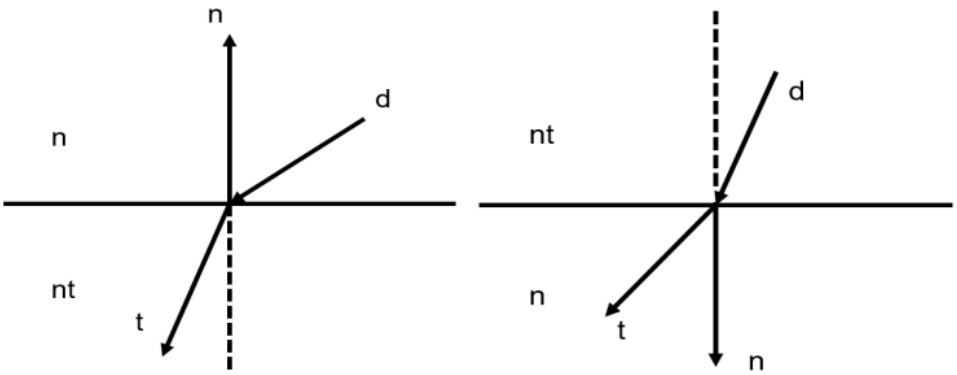


图 3.2.3.3　　折射的两种情况

如图 3.2.3.3 中左侧这种情况时，即 d 和 n 成钝角，用下列的折射公式

$$t = \frac{n(\vec{d} - \vec{n}(\vec{d} \cdot \vec{n}))}{nt} - \vec{n}\sqrt{1 - \frac{n^2(1 - (\vec{d} \cdot \vec{n})^2)}{nt^2}}$$

如图 3.2.3.3 中右侧这种情况时，即 d 和 n 成锐角，用下列的折射公式

$$t = \frac{nt(\vec{d} - \vec{n}(\vec{d} \cdot \vec{n}))}{n} + \vec{n}\sqrt{1 - \frac{nt^2(1 - (\vec{d} \cdot \vec{n})^2)}{n^2}}$$

3.3. 实验方法

3. 3. 1.　　基本数据结构

三维向量 Vec3 类：

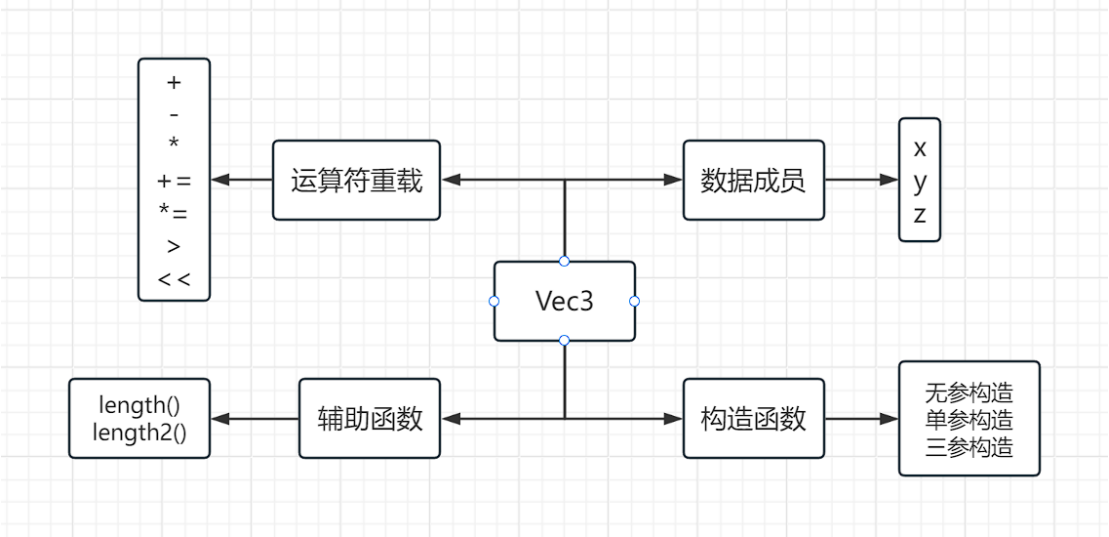


图 3.3.1.1　　Vec3 类数据结构

以下为 Vec3 类具体代码实现：

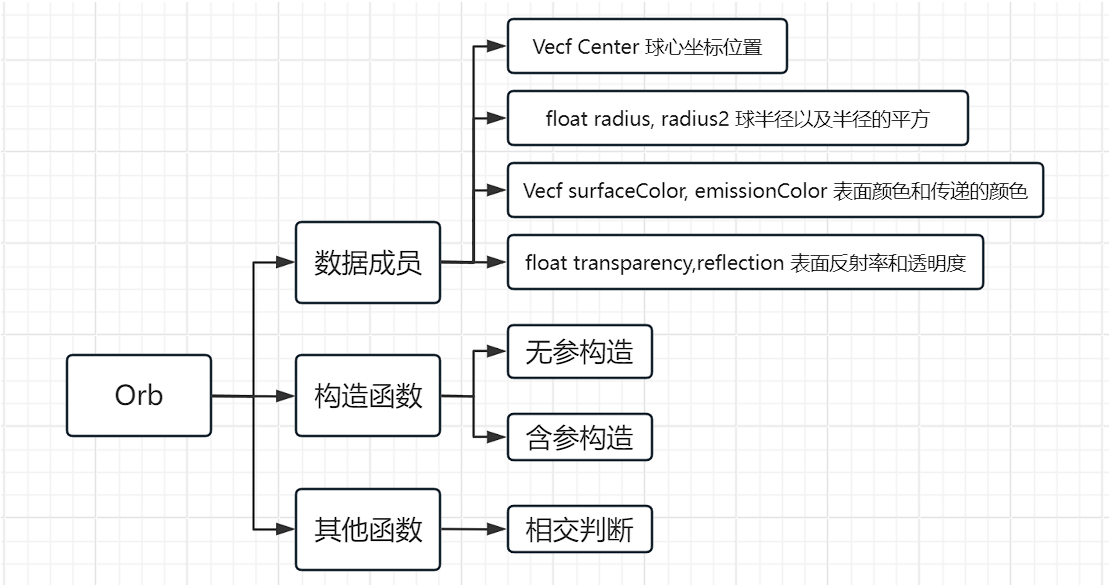
```
template<typename T>
class Vec
{
public:
    T x, y, z;
    Vec() : x(T(0)), y(T(0)), z(T(0)) {}
    Vec(T xx) : x(xx), y(xx), z(xx) {}
    Vec(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}
    Vec& normalize()// 向量单维化
    {
        T nor2 = length2();
        if (nor2 > 0)
        {
            T invNor = 1 / sqrt(nor2);
            x *= invNor, y *= invNor, z *= invNor;
        }
        return *this;
    }

    // Vec 中部分运算符的重载

    Vec<T> operator * (const T &f) const { return Vec<T>(x * f, y * f,
z * f); }
    Vec<T> operator * (const Vec<T> &v) const { return Vec<T>(x * v.x,
y * v.y, z * v.z); }
    T dot(const Vec<T> &v) const { return x * v.x + y * v.y + z * v.z; }
    Vec<T> operator - (const Vec<T> &v) const { return Vec<T>(x - v.x,
y - v.y, z - v.z); }
```

```
        Vec<T> operator + (const Vec<T> &v) const { return Vec<T>(x + v.x, y + v.y, z + v.z); }
        Vec<T>& operator += (const Vec<T> &v) { x += v.x, y += v.y, z += v.z; return *this; }
        Vec<T>& operator *= (const Vec<T> &v) { x *= v.x, y *= v.y, z *= v.z; return *this; }
        Vec<T> operator - () const { return Vec<T>(-x, -y, -z); }
        T length2() const { return x * x + y * y + z * z; }// 获取到原点位置平方数据
        T length() const { return sqrt(length2()); }
        friend std::ostream & operator << (std::ostream &os, const Vec<T> &v)
        {
            os << "[" << v.x << " " << v.y << " " << v.z << "]";
            return os;
        }
    };
    typedef Vec<float> Vecf;
```

球体 Orb 类，用来组织简单的球体：



以下为 Orb 类具体代码实现：

```
    class orb
    {
    public:
        Vecf center;                           // 球心坐标位置
        float radius, radius2;                 // 球半径以及半径的平方
        Vecf surfaceColor, emissionColor;      // 表面颜色和传递的颜色
        float transparency, reflection;        // 表面反射率和透明度
```

```
        /*orb* left = NULL;
        orb* right = NULL;*/

        orb() { }
        orb(
                const Vecf &c, //中心点向量
                const float &r, //半径
                const Vecf &sc, //表面颜色
                const float &refl = 0, //折射率
                const float &transp = 0, //反射率
                const Vecf &ec = 0):
                center(c), radius(r), radius2(r * r), surfaceColor(sc),
emissionColor(ec),
                transparency(transp), reflection(refl)
        { }

        bool intersect(const Vecf &rayorig, //光线原点
                       const Vecf &raydir,  //光线方向
                       float *t0, //第一个交点
                       float *t1 //第二个交点
        ) const
        {
            Vecf l = center - rayorig;         // 光线原点到球心向量 l
            float cos = l.dot(raydir);         // 入射方向和光线到球心向量的
夹角余弦,cos = l*cos 夹角
            if (cos < 0) return false;          // 如果夹角大于 90 度，光线不
可能射中球体
            float d2 = l.dot(l) - (cos * cos);  // d2 = l^2-l^2*cos^2 =
l^2*sin^2
            if (d2 > radius2) return false;     // 光线和球无交点
            float thc = sqrt(radius2 - d2); // radius2 = r^2
            if (t0 != nullptr && t1 != nullptr)
            {
                *t0 = cos - thc;               // 到前一个交点的距离
                *t1 = cos + thc;               // 到后一个交点的距离
            }
            return true;
        }
    };
```

3.3.2. 主要函数分析

该函数为主跟踪函数。它以光线作为参数（由原点和方向表示光线）。将测试此光线是否与场景中的球相交。

如果光线与物体相交，在交点处计算交点坐标，法向，并使用此信息对该点进行着色。着色取决于表面属性（透明、反射、漫反射），函数返回光线的颜色。光线颜色为交叉点处对象的颜色，或者背景色。

本次实验考虑了菲涅尔效应，不同光波分量被折射和反射 视线垂直于表面时，反射较弱，而当视线非垂直表面时，夹角越小，反射越明显，如果目标是圆球，那圆球中心的反射较弱，靠近边缘较强。不过这种过度关系被折射率影响。

具体实现：

```
Vecf trace(
        const Vecf &rayorig, // 光线原点
        const Vecf &raydir, // 光线的单位方向向量
        const std::vector<orb *> &orbs, // 球体集合
        const int &depth) // 递归深度
{
    float tnear = INFINITY; // 一开始定义初始距离为无穷
    const orb* orb = nullptr; // 此处相当于 temp

    // 在场景中找到此光线与球体最前面的交点
    for (auto i : orbs) // 对每一个球体进行相交判断
    {
        float t0 = INFINITY, t1 = INFINITY; // 直线与球面要么两个交点，要么没有交点
        if (i->intersect(rayorig, raydir, &t0, &t1)) // 进行光线与球体的相交判断
        {
            if (t0 < 0) t0 = t1; // 如果光线在球的里面，就采用前面的交点

            if (t0 < tnear)
            {
                // 判断 tnear 是否是最近的交点
                tnear = t0; // 将最近的交点设置为 t0
                orb = i; // 设置球体
            }
        }
    }
```

```
        if(orb != NULL)
        {
            Vecf surfaceColor = 1.0;              // 球体表面的颜色
            Vecf phit = rayorig + raydir * tnear; // 通过光线原点+t*单位方向向量获得交点
            Vecf nhit = phit - orb->center;    // 计算交点法向量
            nhit.normalize();                     // 交点法向量规范化
            float bias = 1e-4;                    // 在要跟踪的点上添加一些偏差
            if (raydir.dot(nhit) > 0)             //如果法线和视图方向不相反，反转法线方向
            {
                nhit = -nhit;
            }
            if ((orb->transparency > 0 || orb->reflection > 0) && depth < MAX_RAY_DEPTH)
            {
                // 进行反射计算
                float IdotN = raydir.dot(nhit); // 光线方向规范化
                float facingratio = std::max(float(0), -IdotN); // 如果-IdotN为负，说明在视点背面，不用显示，取0
                // 不同光波分量被折射和反射，当视线垂直于表面时，反射较弱，而当视线非垂直表面时，夹角越小，反射越明显(菲涅尔效应)
                float fresneleffect = mix(pow(1 - facingratio, 3), 1, 0.1); //菲涅尔效应
                Vecf refldir; // 反射光线
                refldir = raydir - nhit * 2 * raydir.dot(nhit); // r = d + 2 ( d·n ) n 反射光线计算公式（PPT里出现的
                refldir.normalize(); //反射光线向量规范化
                // 递归调用
                Vecf reflection = trace(phit + nhit * bias, refldir, orbs, depth + 1); // 交点作为原点，进行光线追踪，递归深度++ 返回颜色

                Vecf refraction = 0; //初始化折射率
                // 如果透明度不为零，进行折射计算
                if (orb->transparency > 0)
                {
                    // 折射
                    float ior = 1.2; //折射系数
                    float eta = 1 / ior; //折射率
                    float k = 1 - eta * eta * (1 - IdotN * IdotN); // 菲涅尔折射系数
                    Vecf refrdir = raydir * eta - nhit * (eta *  IdotN + sqrt(k)); // 方向向量乘上折射率，然后加上菲涅尔效应的影响
```

```
                refrdir.normalize(); //折射光线规范化
                refraction = trace(phit - nhit * bias, refrdir, orbs,
depth + 1); // 交点作为原点，进行光线追踪，递归深度++ 返回颜色
            }
            // 结果是反射和折射的混合（如果球体是透明的）
            surfaceColor =
                   (reflection * fresneleffect + //反射部分的颜色
                    refraction    *    (1    -    fresneleffect)    *
orb->transparency)//折射部分的颜色
                   * orb->surfaceColor; //两者的和乘上物体表面颜色，得到
最后颜色
            //R(color(p+tr)+(1-R)color(p+t*t))
        }
        else
        {
            // 这是一个折射率和反射率为 0 的物体，不需要再进行光线追踪
            double shadow = 1.0;
            for (unsigned i = 0; i < orbs.size(); ++i) //遍历每个物体，
依次计算是否相交，相交的话更新阴影
            {
                if (orbs[i]->emissionColor.x > 0)
                {
                    Vecf transmission = 1.0; //初始化
                    Vecf lightDirection = orbs[i]->center - phit; //球
体法向量
                    lightDirection.normalize(); //法向量规范化

                    for (unsigned j = 0; j < orbs.size(); ++j)
                    {
                        if (i != j)
                        {
                            float t0, t1;
                            // 判断该点的光是否和源光线相交，如果相交，计算
阴影
                            if (orbs[j]->intersect(phit + (nhit * bias),
lightDirection, &t0, &t1))
                            {
                                shadow = std::max(0.0, shadow - (1.0 -
orbs[j]->transparency)); //相交的话更新折射率
                                transmission = transmission * shadow; //
计算转化率
                            }
                        }
                    }
```

```
                    // 用 phong 模型计算每一条对这点像素造成影响的光线
                    surfaceColor += orb->surfaceColor * transmission *
orbs[i]->emissionColor; //加上最后传递的颜色
                }
            }
        }
        return surfaceColor; //返回最终的颜色
    }
    else
    {
        return Vecf(1.0, 1.0, 1.0); //背景为白色
    }
}
```

3.3.3. 项目结构

本项目项目组织结构如图 3.3.3.1 所示

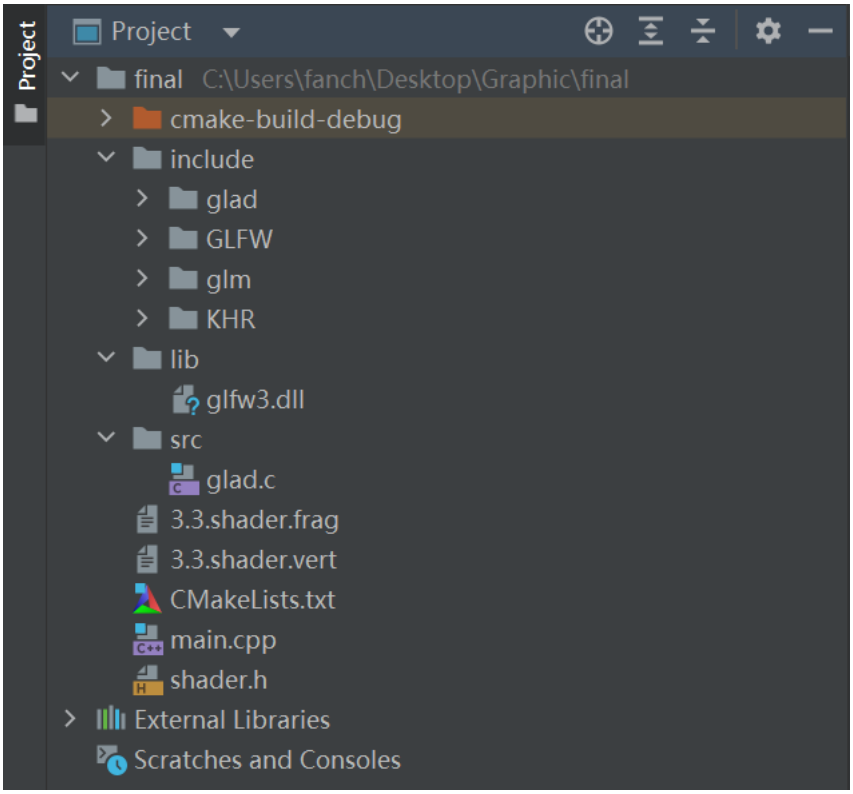


图 3.3.3.1 项目结构

本项目基于 GLFW,Glad 和 Glm 库，使用顶点着色器以及片段着色器，之后通过一个主程序以及一个着色器配置文件完成功能。

4. 实验结果

运行效果如图 3.4.1 所示

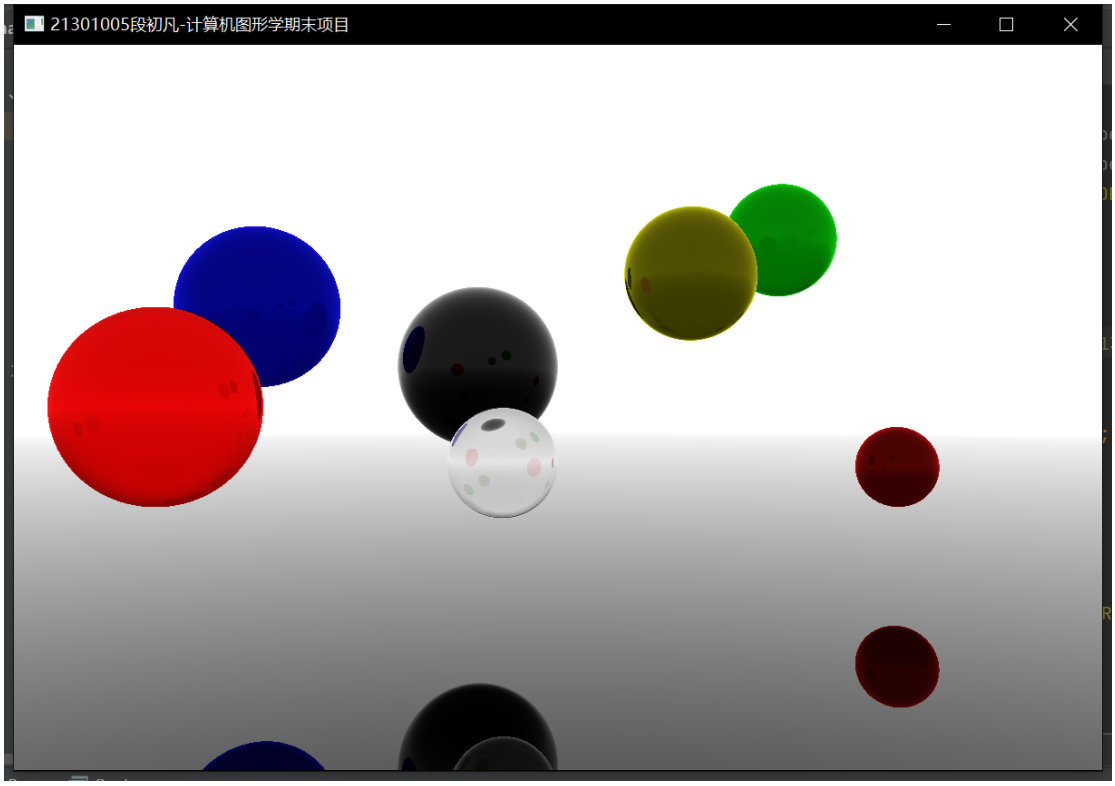


图 3.4.1 运行效果

在场景中包括底部的巨大白色球体和六个彩色的小球体(半径为 1~2 个单位)，经过复杂的三维位置放置以及相互遮盖，可以看到光线追踪中反射和折射的明显效果。

底部的白色球体设置成了全反射，所以可以看到 4 个反射出来的球体。同时中间那个透明球体也可以看出反射出了周围各个球体的模型，体现出了光线追踪的效果。

5. 总结与展望

通过对光线追踪研究现状及文献的查阅，并且设计并实现了一个基于 C++和 OpenGL 的简单光线追踪 Demo，笔者对于光线追踪的研究有了一个较为全面和深入的了解。经过本次的学习，在之后笔者可以继续深入了解光线追踪的应用，改良本次项目场景过于简单以及无更多变化的缺点，实现对该领域的更深入全面的研究。

参考文献

[1] 闫润,黄立波,郭辉,等. 实时光线追踪相关研究综述[J]. 计算机科学与探索,2023,17(2):263-278. DOI:10.3778/j.issn.1673-9418.2207067.

[2] 李家振,纪庆革. 动态低采样环境光遮蔽的实时光线追踪分子渲染[J]. 计算机科学,2022,49(1):175-180. DOI:10.11896/jsjkx.210200042.

[3] 王平,李益文,乔磊,等. 基于加速结构和 CUDA 的光线追踪算法和技术研究[J]. 承德石油高等专科学校学报,2022,24(5):42-46. DOI:10.3969/j.issn.1008-9446.2022.05.009.

[4] 张初旭. 光线追踪在系统热辐射排故分析中的应用[J]. 内燃机与配件,2023(17):33-36. DOI:10.3969/j.issn.1674-957X.2023.17.011.