**实验五 光线追踪综合实验**

姓名 徐昊博 学号21013134 专业班级 计213 成绩\_\_\_\_\_\_\_\_\_

实验日期2023年12月4日 实验地点 信息楼215 指导教师(签名)

一．实验目的

1、 学习光线追踪原理。

2、 利用GLSL编程实现对模型的渲染。

二．实验工具与设备

**推荐Visual Studio 2019.**

三、实验内容

在本科学习平台（s.ecust.edu.cn）资料栏下，下载以下文件：“实验五资料.zip”

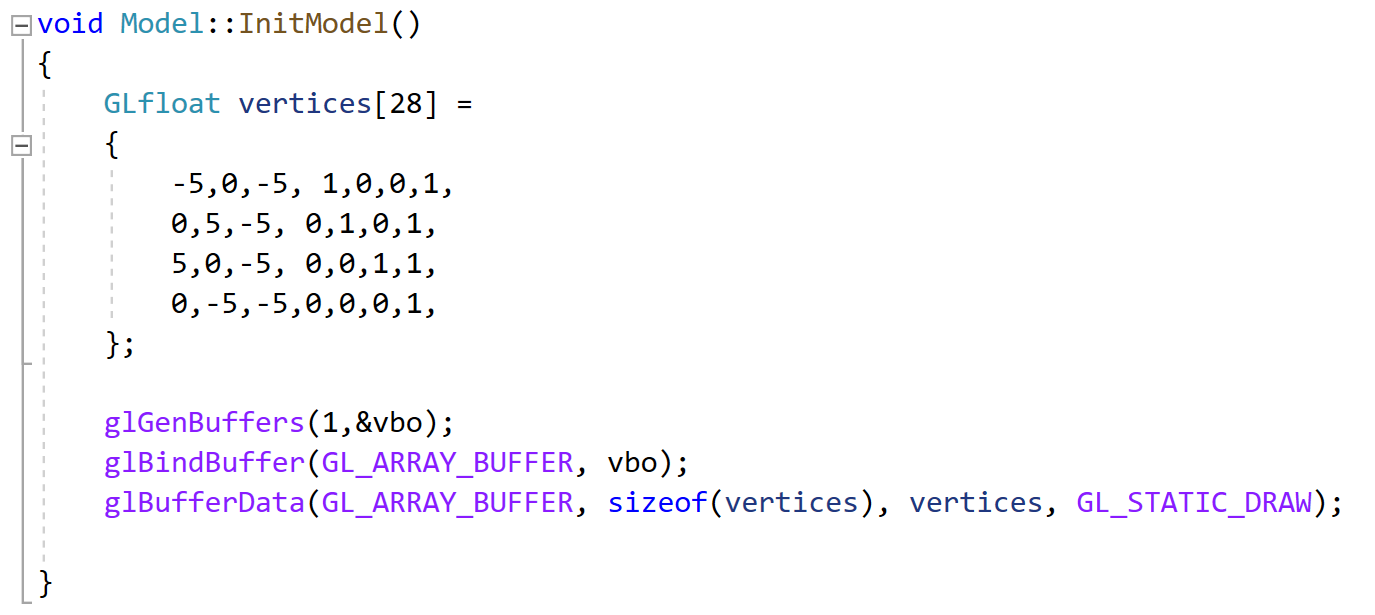
1. 学习着色器编程

利用“实验五资料.zip”中的“实验五-环境安装指南.docx”配置glew，再配置好FreeGLUT，结合“ShaderProgram.ppt”，运行“ShaderTest.rar”实例，总结出着色器编程的实现要点；编写绘制一个四边形的着色器编程实例，给出修改要点，并截图。

**一、程序实现要点：**

在这个实验项目中着色器主要通过Model的类函数来进行实现，主要有以下两个函数构成：

（一）初始化模型



其中vertices数组确定了需要绘制的顶点的信息，需要注意的是一个顶点一共有7个参数，在上述代码中每一行的7个数字都代表了一个顶点信息，其中前三个为坐标的xyz坐标值，4-6个数字代表了RGB颜色的设置，而最后一个数字代表不透明度。

后三行代码是用于创建和配置 OpenGL 的顶点缓冲对象（VBO），它是在 GPU 上存储顶点数据的一种方式。

1. glGenBuffers：

glGenBuffers()用于生成一个缓冲对象的标识符（ID），这里是为一个缓冲对象生成一个唯一的标识符。第一个参数是要生成缓冲对象的数量，这里是生成一个缓冲对象。第二个参数 `&vbo` 是一个指向vbo变量地址的指针，vbo用来存储生成的缓冲对象的标识符。

1. glBindBuffer：

glBindBuffer()用于绑定缓冲对象第一个参数 GL\_ARRAY\_BUFFER表示将要绑定的缓冲对象类型，这里表示顶点数组缓冲对象。第二个参数vbo是之前生成的缓冲对象的标识符，将这个缓冲对象绑定到 GL\_ARRAY\_BUFFER类型上。

1. glBufferData：

glBufferData()用于将数据复制到当前绑定的缓冲对象。第一个参数GL\_ARRAY\_BUFFER表示当前绑定的是哪个类型的缓冲对象。第二个参数sizeof(vertices)表示要传输的数据大小，即顶点数据数组vertices的字节数。第三个参数vertices是包含顶点数据的数组。最后一个参数GL\_STATIC\_DRAW是关于数据使用方式的提示，它表示这些数据将被修改一次，但会被频繁使用。（二）初始化模型



这段代码中的几行涉及使用着色器程序、创建模型视图矩阵和投影矩阵，通常在图形渲染过程中用于设置渲染的环境和变换。

1. glUseProgram(shader->GetShaderProgram())：

- glUseProgram() 函数用于激活特定的着色器程序。

- shader->GetShaderProgram() 返回着色器程序的 ID，这里使用该 ID 来激活对应的着色器程序，使得后续的渲染调用会使用这个着色器程序。

2. `Matrix4f modelView = Matrix4f::createLookAt(Vector3f(0,0,0), Vector3f(0,0,-1), Vector3f(0,1,0));：

- 这行代码创建了一个模型视图矩阵 modelView。

- Matrix4f::createLookAt()` 通常是一个辅助函数，用来创建模型视图矩阵。在这里，它使用了摄像机位置 `(0, 0, 0)`、观察目标 `(0, 0, -1)` 和上方向 (0, 1, 0) 来构建视图矩阵。

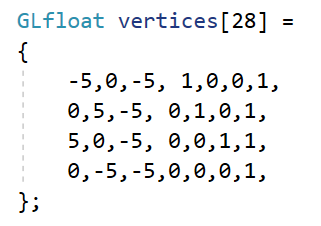
3. Matrix4f proj = Matrix4f::createFrustum(-1, 1, -1, 1, 1, 50);：

- 这行代码创建了一个投影矩阵 proj。

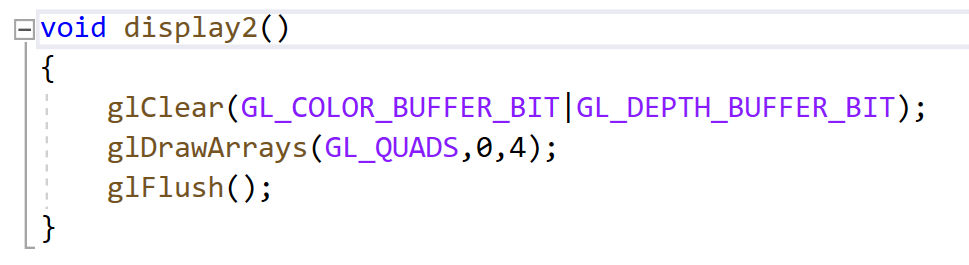
- Matrix4f::createFrustum() 通常是一个用来创建透视投影矩阵的辅助函数。它接收了左、右、底部、顶部、近裁剪平面和远裁剪平面的参数来构建透视投影矩阵，定义了视景体的形状和大小。

**二、绘制一个四边形的着色器编程实例：**

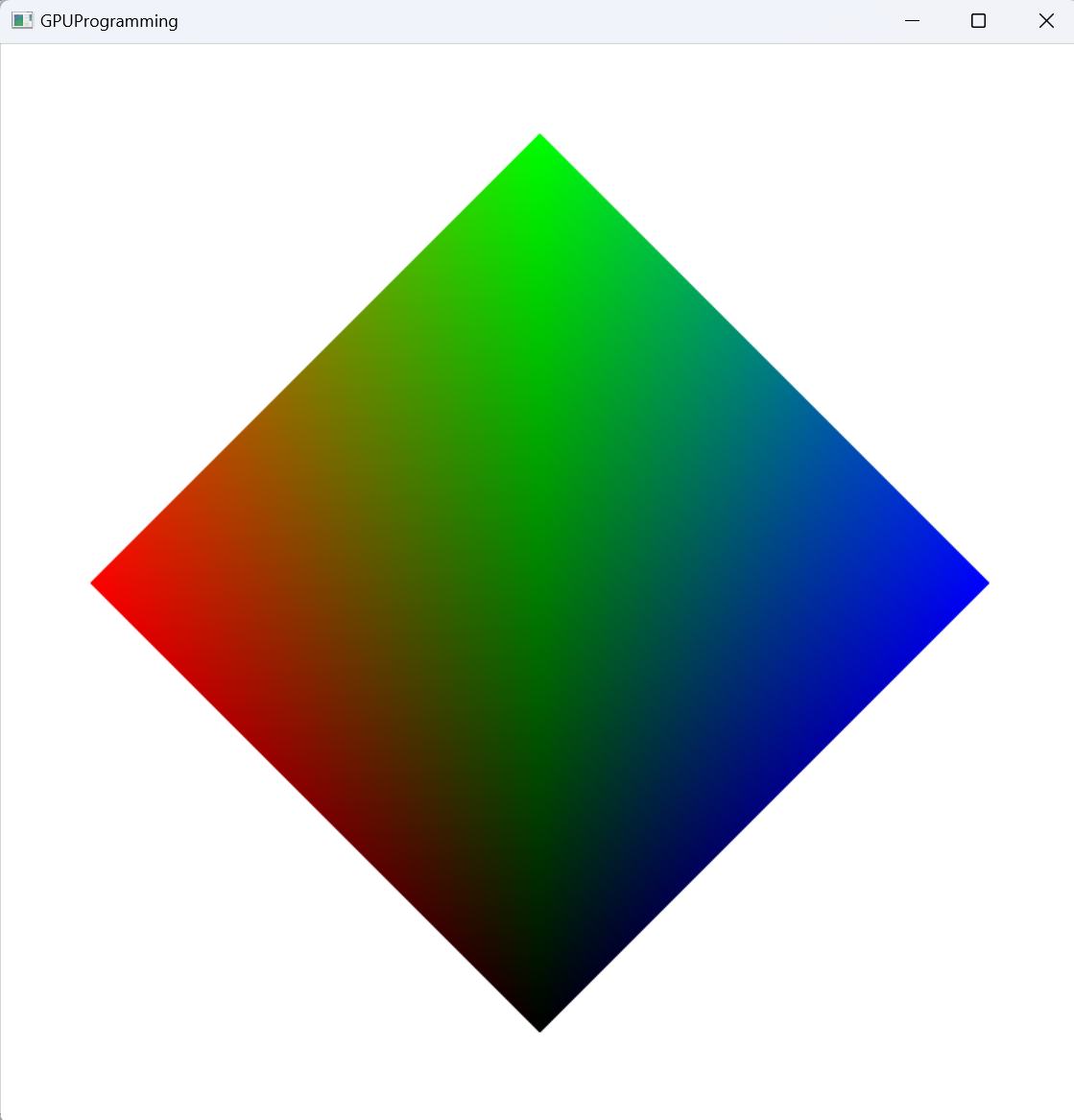
修改下列代码即可：



首先将顶点修改为4个，所以数组大小从21上升到28，在增加的数组中确定新的点的相关参数，这里确定了坐标为（0，-5，-5），颜色为黑色，即RGB全都为0，根据几何知识可以知道绘制的图形应当是正方形。



除此之外还需要修改Display2函数，将DrawArrays中参数改为GL\_GUADS，即修改为四边形模式，同时后两个参数表示使用代表了前四个点的参数来进行四边形绘制。绘制结果如下所示：



可以清晰看到最下面添加的点颜色为黑色，符合预期。

2、RayCasting编程

利用“实验五资料.zip”中的“实验五-环境安装指南.docx”和“RayCasting”代码，分析代码过程，简单写出程序的实现要点；在原程序界面球体右方再添加一个物体（长方体、正方体、球体都可），并给出修改要点，并截图。

**一、程序实现要点：**

该实验主要通过着色器代码进行实现，主要内容在于rayTraceComputerShader.glsl当中是一个使用计算着色器的简单光线追踪器。它使用了OpenGL的Compute Shader来实现。主要过程如下：

1. 定义结构体和数据：Ray结构体定义了射线的起点和方向。Collision结构体包含了碰撞信息，包括碰撞时间t、碰撞位置p、法线n是否从内部开始inside以及对象索引object\_index。

2. 定义物体参数：定义了一个球体和一个盒子，每个物体都有位置、大小和颜色。

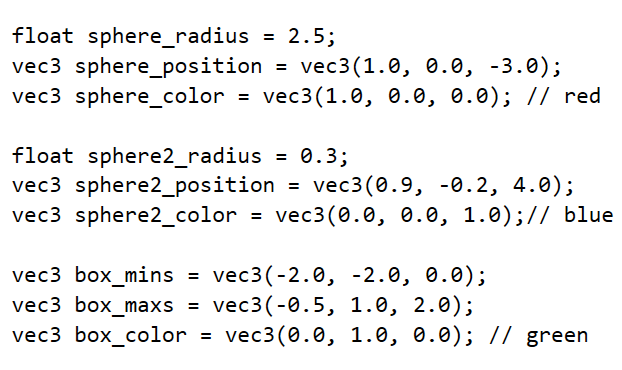
3. 光线和物体的碰撞检测：intersect\_box\_object和intersect\_sphere\_object函数用于检测光线与盒子、球体之间的碰撞。这些函数基于射线与物体的相交算法进行计算，并返回 Collision 结构体。get\_closest\_collision` 函数返回光线与场景中最近物体的碰撞信息。

4. 光线追踪：raytrace函数使用get\_closest\_collision来获取光线与物体的最近碰撞，然后根据碰撞对象的索引返回对应的颜色。

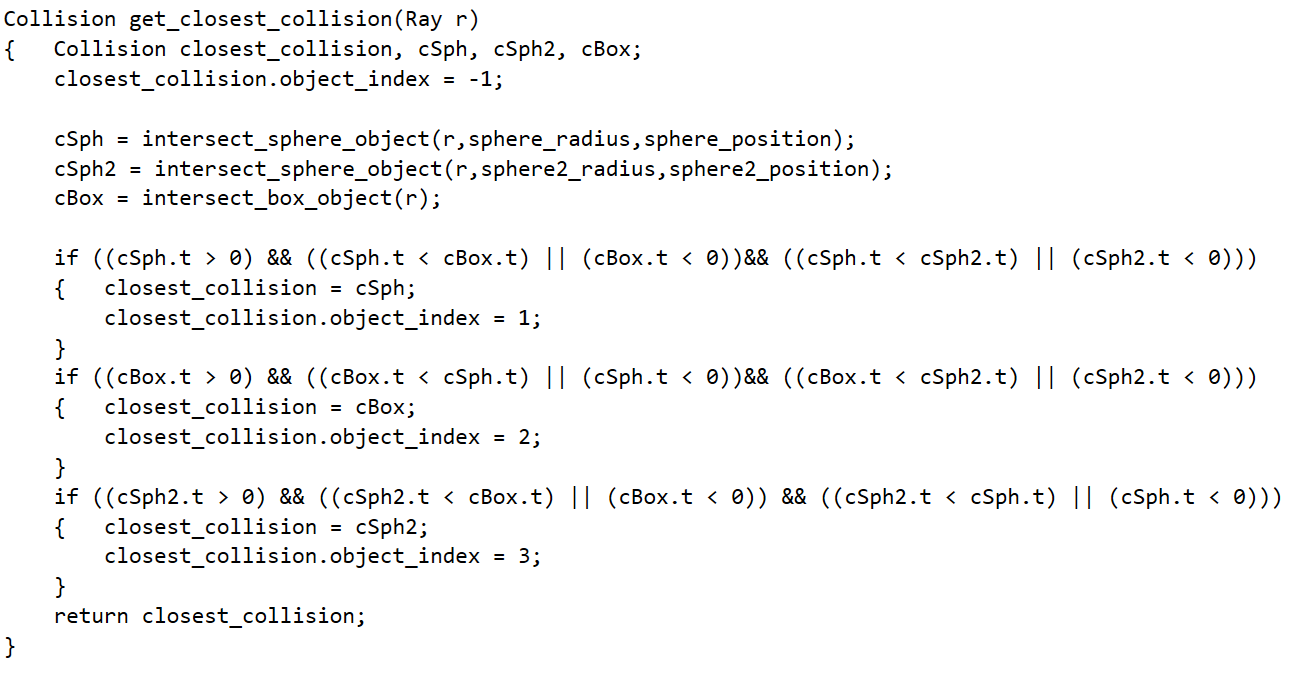
5. 主函数：main函数用于计算每个像素的颜色。它首先将屏幕空间坐标转换为世界空间中的射线。然后，使用raytrace函数获得该射线与场景中物体的碰撞信息，并返回相应的颜色值。最后，将计算得到的颜色存储到输出纹理中。

**二、添加一个物体修改过程：**

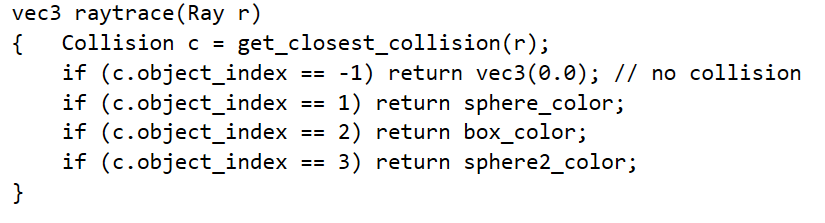
目标：在原有球体旁边新建一个球体，修改步骤如下：

1. 首先仿照前一个球体的参数进行设置新的球体模型参数，球体名称设为sphere2，相关参数为：半径0.3，球体坐标为（0.9，-0.2，4.0），颜色为（0，0，1）代码如下：

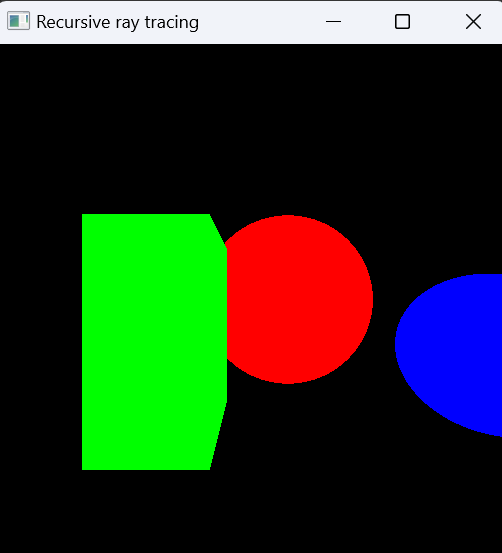
2、获取光线与场景中物体的最近相交点函数修改，将新的球体初始化为一开始的碰撞结构体，然后与整个代码中其他物体进行碰撞检测，并返回每个物体的object\_index值，注意调用球体的相交函数时需要带入修改后的参数列表，代码如下：



同样，颜色值返回也需要增加，代码修改如下：



按照上述代码修改完成之后运行就可以看到新的球体位于原来球体右侧。



3、Recursive ray tracing编程

利用“实验五资料.zip”中的“实验五-环境安装指南.docx”和“Recursive ray tracing”代码，分析代码过程，简单写出程序的实现要点；在原程序界面球体右方再添加一个物体（长方体、正方体、球体都可），并给出修改要点，并截图。

**一、程序实现要点：**

该实验主要通过着色器代码进行实现，主要内容在于rayTraceComputerShader.glsl当中这段代码是一个使用着色器进行光线追踪的实现。它使用OpenGL着色器语言(OpenGL Shading Language, GLSL)编写，实现了一个简单的光线追踪渲染器。

这段代码主要包括以下几个部分：

1. 射线(Ray)结构体定义了光线的起点和方向。

2. 碰撞(Collision)结构体用于存储光线与场景中物体相交时的信息，例如碰撞点、法线、是否在物体内部等。

3. Stack\_Element结构体定义了堆栈中的元素，用于实现递归光线追踪。

4. buildTranslate、buildRotateX、buildRotateY、buildRotateZ函数用于构建平移和旋转矩阵。

5. intersect\_sphere\_object、intersect\_box\_object、intersect\_room\_box\_object、intersect\_plane\_object\*\* 函数分别用于计算射线与球体、长方体、房间长方体和平面的相交。

6. get\_closest\_collision函数用于获取光线与场景中物体的最近相交点。

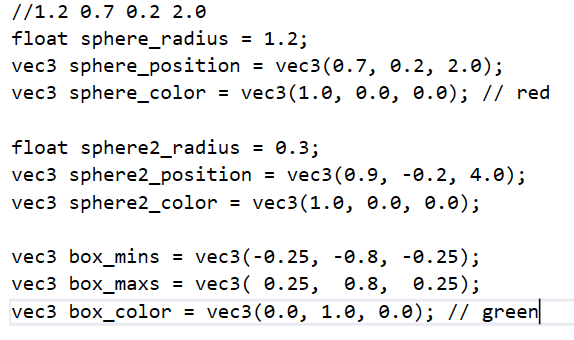
7. ads\_phong\_lighting函数实现了Phong光照模型，计算了环境光、漫反射和镜面反射的光照颜色。

8. push、pop、process\_stack\_element函数用于管理递归堆栈，实现递归光线追踪。

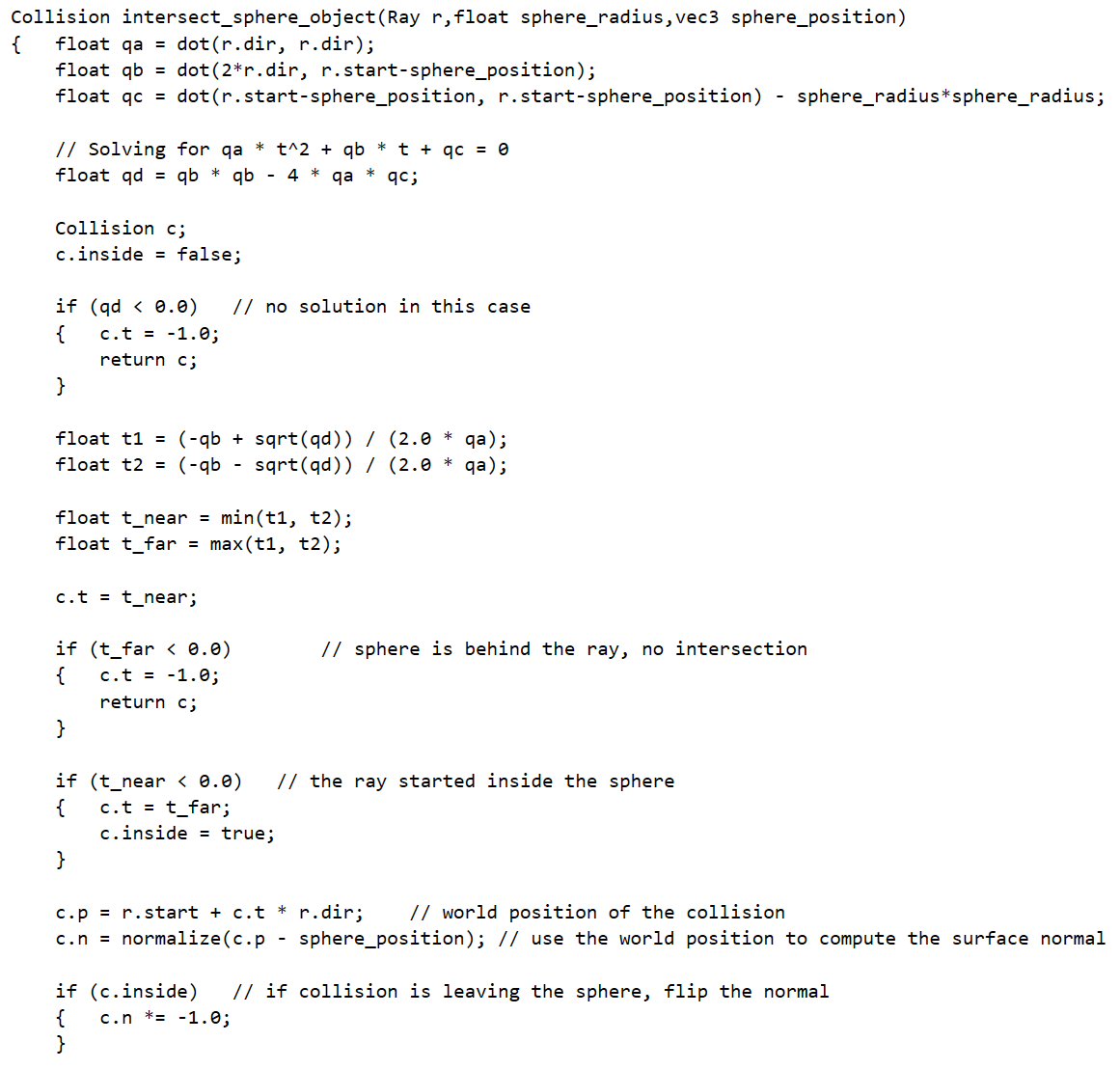
**二、添加一个物体修改过程：**

目标：在原有球体旁边新建一个球体，修改步骤如下：

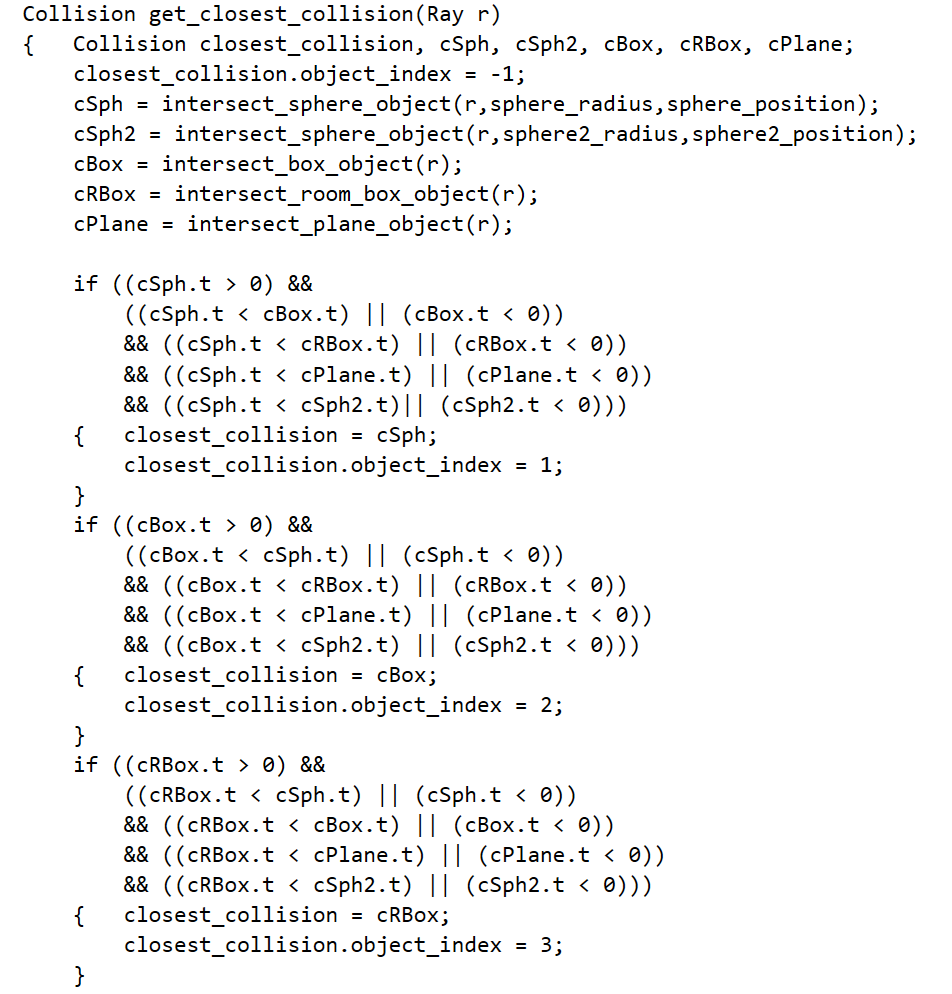
1、首先仿照前一个球体的参数进行设置新的球体模型参数，球体名称设为sphere2，相关参数为：半径0.3，球体坐标为（0.9，-0.2，4.0）代码如下：

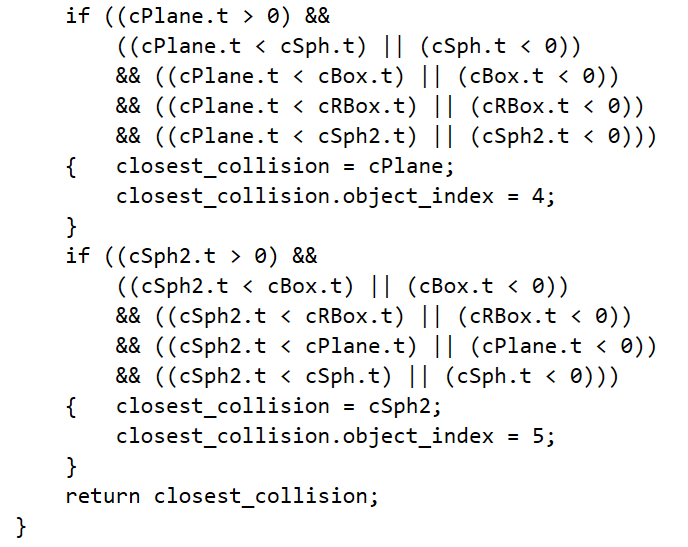


2、修改加入球体函数的参数，由于新建的球体参数需要加入，所以修改原来的球体相交函数内的形式参数，将半径和位置加入函数中，这样在后面函数需要调用时把sphere2的相关参数带入函数就能实现第二个球的光照效果与绘制效果等等，调用的半径位置等形参设置为函数内原先有的名称即可，代码如下：

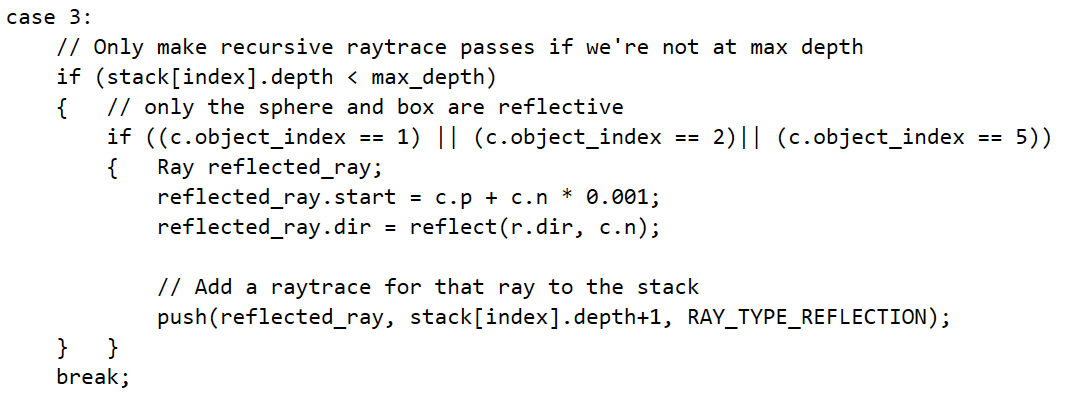


3、获取光线与场景中物体的最近相交点函数修改，将新的球体初始化为一开始的碰撞结构体，然后与整个代码中其他物体进行碰撞检测，并返回每个物体的object\_index值，注意调用球体的相交函数时需要带入修改后的参数列表，代码如下：

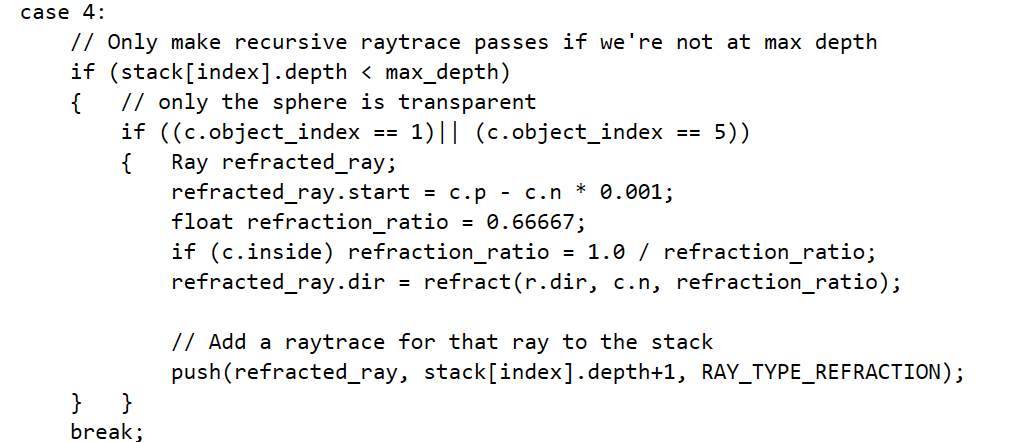




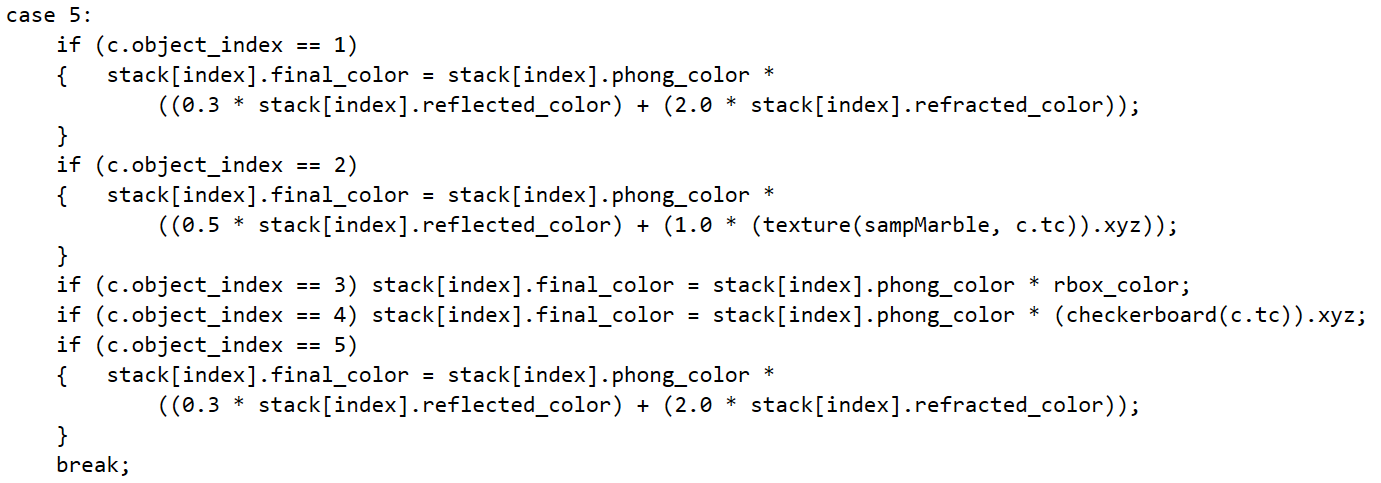
4、实现递归光线追踪修改，由于需要对球体和长方体进行反射光线处理，新添加的球体object\_index值为5所以增加到if语句中即可，代码如下：



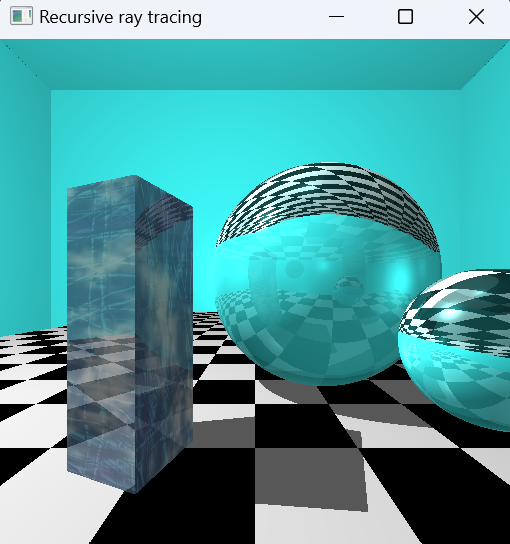
同样的由于折射仅对球体有效，所以仅对object\_index值为1和5的两个球体进行判定即可：



最终的颜色设置与object\_index值为1的球体做相同的分支处理即可，代码如下：



按照上述代码修改完成之后运行就可以看到新的球体位于原来球体右侧，并且在原先球体中可以看到反射光线。



四、思考题

**1．程序中使用freeglut、glew、glfw、glad库的各自作用是什么？**

这些库都是用于 OpenGL 开发的辅助工具库，它们在不同方面提供支持和功能，让 OpenGL 开发更加便捷和灵活。

1. FreeGLUT：

FreeGLUT（Free OpenGL Utility Toolkit）是一个用于创建和管理 OpenGL 窗口的工具库。它提供了一个跨平台的开源替代品，可以创建窗口、处理输入事件（如键盘和鼠标输入）、管理多窗口、以及处理窗口大小和位置等。简化了创建和管理 OpenGL 窗口的过程，使得开发者能够更专注于图形渲染而不必过多关注窗口系统的底层细节。

2. GLEW（OpenGL Extension Wrangler）：

GLEW 是一个用于管理 OpenGL 扩展的库，可以方便地让开发者在不同的硬件和驱动上使用 OpenGL 的各种扩展功能。OpenGL 的功能可以根据硬件和驱动的支持情况有所不同，有时需要加载特定的扩展才能使用新功能。GLEW 提供了简单的接口来查询和加载这些扩展，使得开发者能够方便地使用 OpenGL 提供的全部功能。

3. GLFW：

GLFW（Graphics Library Framework）是一个专注于创建窗口、处理用户输入和创建 OpenGL 上下文的库。它和 FreeGLUT 类似，但更轻量级且提供更多现代化的功能，包括创建 OpenGL 上下文、多窗口管理、处理输入（键盘、鼠标等）、监测窗口变化以及处理事件等。同时GLFW 是一个跨平台的工具库，适用于 Windows、macOS 和 Linux 等多种操作系统。

4. GLAD（OpenGL Loader Generator）：

GLAD 是一个用于加载 OpenGL 函数指针的库，它帮助开发者获取并管理 OpenGL 的函数指针。在使用 OpenGL 时，需要动态加载 OpenGL 的函数指针，而不同的操作系统或驱动提供的 OpenGL 版本可能不同。GLAD 通过生成加载器代码来管理这些指针，使得开发者可以方便地使用特定版本的 OpenGL 函数。

**2.利用着色器实现光线追踪的优缺点**？

优点：

1. 并行性和GPU加速：

GPU 是针对并行处理优化的硬件。利用着色器实现光线追踪可以利用 GPU 的并行性，加速光线追踪过程，特别是对于大规模场景和复杂光线追踪场景效果显著。

2. 灵活性和可编程性：

着色器语言（如OpenGL Shading Language - GLSL 或者 Vulkan的SPIR-V）提供了灵活的编程接口，允许开发者实现各种光线追踪算法和效果，例如阴影、折射、反射、全局光照等。

3. 现代图形API支持：

现代图形API（如OpenGL或Vulkan）提供了强大的图形渲染功能和着色器支持，使得利用GPU实现光线追踪成为可能。

缺点：

1. 难以实现复杂算法：

虽然着色器提供了一定的编程能力，但实现复杂的光线追踪算法可能会面临限制。相比于专门用于计算的CPU，GPU在编程模型、内存限制和指令集方面存在限制。

2. 难以处理动态场景：

光线追踪通常需要追踪光线与场景中物体的相交情况。对于动态场景，特别是需要动态更新的物体，光线追踪的实时性和效率可能会受到影响，因为每次场景发生变化都需要重新计算。

3. 硬件要求和性能成本：

光线追踪通常需要大量的计算资源，包括内存、处理器性能和图形处理能力。在较老或低端的硬件上，实现实时光线追踪可能会面临性能挑战。

综上所述，利用着色器实现光线追踪在GPU加速和灵活性方面具有优势，但在处理复杂算法和动态场景时存在挑战。这种方法适合于静态或半静态的场景，对实时性要求不高的应用，例如高质量渲染、可视化、影视特效等领域。