**实验五 光线追踪综合实验**

姓名 陈正江 学号 19001531 专业班级 计211 成绩\_\_\_\_\_\_\_\_\_

实验日期\_2023.12.4 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

**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。

**着色器编程实现要点：**

从基本意义上来讲，着色器只是一种把输入转化为输出的程序。着色器是使用一种叫GLSL的类C语言写成的，GLSL是为图形计算量身定制的，它包含一些针对向量和矩阵操作的有用特性。大致可分为配置数据阶段、顶点着色与片元着色阶段、数据传输阶段和绘制阶段：

1）配置数据阶段需要把三维模型的数据与vbo**顶点缓冲对象（Vertex Buffer Object）**关联起来，以后我们如果需要使用这块内存中的数据，只要调用函数glBindBuffer(GL\_ARRAY\_BUFFER, vbo)即可。

2）顶点着色与片元着色阶段是可编程渲染管线的核心，通过顶点着色器、片段着色器实现输入与输出，顶点着色器中定义一个输出，在片段着色器中定义输入来接收这个输出。

3）数据传输阶段通过使用glEnableVertexAttribArray()与glVertexAttribPointer()函数，定义了内存块中数据的读取方式与含义，并将之上传到着色器中。通过in、Uniform 等数据发送方式，从CPU中的应用向GPU中的着色器发送数据，但uniform和顶点属性有些不同： uniform是全局的(Global)，无论把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。

4）绘制阶段输入顶点属性传递颜色GLfloat vertices[]并调用OpenGL绘制函数进行渲染绘制。

**绘制四边形着色器编程实例代码：**

void display2(){

glClear(GL\_COLOR\_BUFFER\_BIT|GL\_DEPTH\_BUFFER\_BIT);

glDrawArrays(GL\_TRIANGLES,0,6);//使用glDrawArrays函数构造四边形

glFlush();

}

void Model::InitModel(){

GLfloat vertices[42] = { //含有42个元素的数组，表示各个顶点的位置坐标和颜色值

0,0,-5, 1,0,0,1,

0,5,-5, 0,1,0,1,

5,0,-5, 0,0,1,1,

5,5,-5, 1,0,0,1,

0,5,-5, 0,1,0,1,

5,0,-5, 0,0,1,1,

};

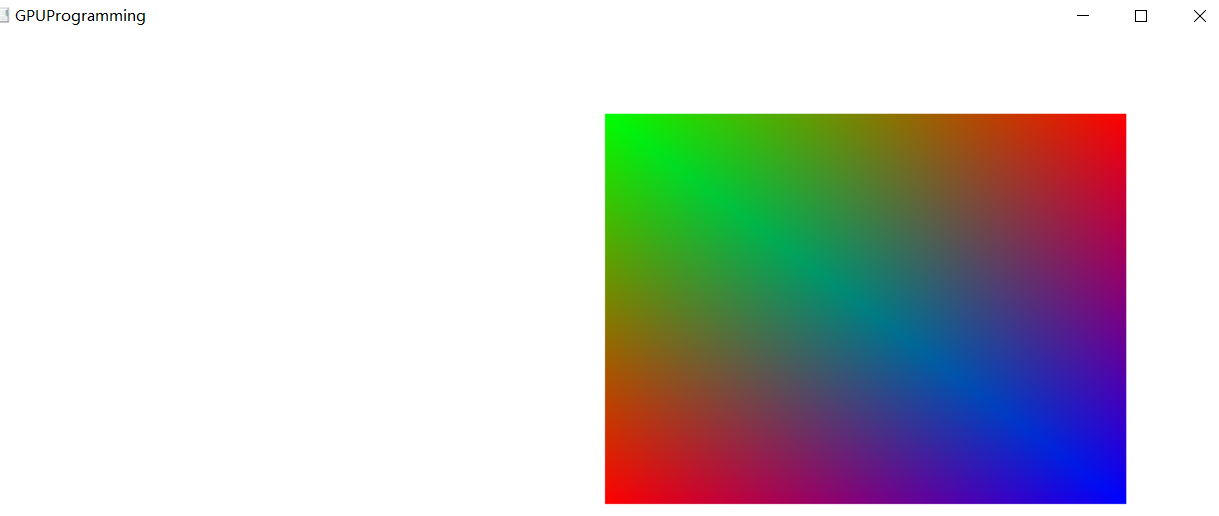
glGenBuffers(1,&vbo);

glBindBuffer(GL\_ARRAY\_BUFFER, vbo);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

}

**运行结果：**



**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。

**RayCasting程序实现要点：**

从人眼（视作点光源1）发出eye ray，即发出一条光线（经过2维像素点，也视作这个像素发出的光线），并且这条光线只记录在路径上遇见的第一个物体点，然后将交点与光源连线，若此连线路径无遮挡则这个点被照亮，若有遮挡则此点在阴影中，自此就可以对交点着色并写入像素点，通过intersect\_box\_object(Ray r)函数判断光线是否进入离开所定义的box。

基于光线的起点和终点。事先算好了光线的起点和终点，根据这两个点可以确定光线的方向。同时根据这两个点之间的距离与光线前进的步长可以计算出光线上有多少个采样点。在起点与终点之间的每一个采样点处，不需要判断这个点是否在长方体内。通过intersect\_sphere\_object(Ray r)函数检查光线是否与定义的球体相交，get\_closest\_collision(Ray r)来得到光线最近的一个碰撞物体。以及通过raytrace(Ray r , sphere s1 , sphere s2)函数将光线投射到场景中，并返回像素的最终颜色。

**程序修改要点：**

struct sphere { //定义球类结构体

float sphere\_radius;

vec3 sphere\_position;

vec3 sphere\_color;

};

Collision intersect\_sphere\_object(Ray r , sphere s) { //检查射线r是否与Sphere球体相交

float qa = dot(r.dir, r.dir);

float qb = dot(2\*r.dir, r.start-s.sphere\_position);

float qc = dot(r.start-s.sphere\_position, r.start-s.sphere\_position) - s.sphere\_radius\*s.sphere\_radius;

float qd = qb \* qb - 4 \* qa \* qc;

Collision c;

c.inside = false;

if(qd < 0.0) { // 无解

c.t = -1.0;

return c;

}

……

c.p = r.start + c.t \* r.dir; // 发生碰撞所在的世界位置

c.n = normalize(c.p - s.sphere\_position); // 使用世界位置来计算表面法线

if(c.inside) c.n \*= -1.0; // 如果碰撞离开球体，翻转法线

return c;

}

Collision get\_closest\_collision(Ray r , sphere s1, sphere s2 ) //计算哪个像素距离光线起点最近

{ Collision closest\_collision, cSph1, cSph2, cBox;

closest\_collision.object\_index = -1;

cSph1 = intersect\_sphere\_object(r,s1);

cSph2 = intersect\_sphere\_object(r,s2);

cBox = intersect\_box\_object(r);

if ((cSph1.t > 0) && ((cSph2.t < cBox.t)&& (cSph1.t < cSph2.t)|| (cBox.t < 0))){

closest\_collision = cSph1;

closest\_collision.object\_index = 1;

}

if ((cSph2.t > 0) && ((cSph1.t < cBox.t)&& (cSph2.t < cSph1.t)|| (cBox.t < 0))){

closest\_collision = cSph2;

closest\_collision.object\_index = 2;

}

if ((cBox.t > 0) && ((cBox.t < cSph1.t) || (cSph1.t < 0))&&((cBox.t < cSph2.t) || (cSph2.t < 0))){

closest\_collision = cBox;

closest\_collision.object\_index = 3;

}

return closest\_collision;

}

vec3 raytrace(Ray r , sphere s1 , sphere s2) { //将光线投射到场景中，并返回像素的最终颜色

Collision c = get\_closest\_collision(r,s1,s2);

if (c.object\_index == -1) return vec3(0.0); // no collision

if (c.object\_index == 1) return s1.sphere\_color;

}

void main(){

sphere a,b; //定义两个球体，分别设置其坐标位置、半径和颜色

a.sphere\_radius = 2.5;

a.sphere\_position = vec3(1.0, 0.0, -3.0);

a.sphere\_color = vec3(1.0, 0.0, 0.0); // red

b.sphere\_radius = 3.5;

b.sphere\_position = vec3(4.0, 2.0, -1.0);

b.sphere\_color = vec3(0.0, 0.0, 1.0); // blue

int width = int(gl\_NumWorkGroups.x);

int height = int(gl\_NumWorkGroups.y);

ivec2 pixel = ivec2(gl\_GlobalInvocationID.xy);

//将这个像素的屏幕空间位置转换为世界空间

float x\_pixel = 2.0 \* pixel.x/width - 1.0;

float y\_pixel = 2.0 \* pixel.y/height - 1.0;

// 得到这个像素的世界空间射线

Ray world\_ray;

world\_ray.start = vec3(0.0, 0.0, camera\_pos);

vec4 world\_ray\_end = vec4(x\_pixel, y\_pixel, camera\_pos-1.0, 1.0);

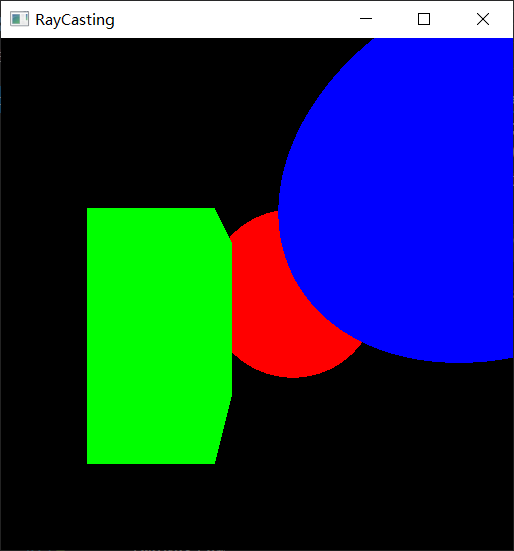
world\_ray.dir = normalize(world\_ray\_end.xyz - world\_ray.start);

// 将光线投射到世界中，并使光线与物体相交

vec3 color1 = raytrace(world\_ray,a,b);

imageStore(output\_texture, pixel, vec4(color1,1.0));

}

**运行结果：**

**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。

**Recursive ray tracing程序实现要点：**

开始步骤和RayCasting程序中相似，从人眼或摄像机向近投影平面上的每一个像素点发射一条光线，判断与场景物体的交点，考虑遮挡关系，只去找最近的交点。通过intersect\_plane\_object(Ray r) 、intersect\_box\_object(Ray r)、intersect\_sphere\_object(Ray r, sphere s )函数，检查光线是否与平面、所定义的box和sphere球体相交，接着通过get\_closest\_collision(Ray r,sphere a,sphere b)函数得到与光线碰撞最近的物体。

当然除了镜面反射之外，自然也存在折射，同时反射与折射出去的光线会可能与场景中的物体再次碰撞，发生第二次折射与反射，在任意一个接收到光线的点可以视作光源继续发射折射光线。这些由同一根光线（同一个像素点发射出来的光线）与物体产生的交点以及折射所产生的交点都与光源做连线并着色返回给像素点。通过ads\_phong\_lighting(Ray r, Collision c,sphere a,sphere b)函数，计算入射光线在物体表面的环境漫反射(ADS) Phong照明并返回颜色。

整体过程是一个递归的过程，需要一定的递归终止条件，比如允许的最大迭代深度4。同时由于GLSL无法实现递归调用，我们定义堆栈并将已实现的递归光线跟踪元素存放至堆栈中，通过process\_stack\_element(int index,sphere a,sphere b)函数实现光线追踪碰撞检测、Phong ADS照明计算、折射透明度照射计算以及最终混合以产生最后的颜色。

**程序修改要点：**

struct sphere{ //定义球类结构体

float sphere\_radius;

vec3 sphere\_position;

vec3 sphere\_color;

};

Collision intersect\_sphere\_object(Ray r, sphere s ){ // 检查射线r是否与定义的球体相交

float qa = dot(r.dir, r.dir);

float qb = dot(2\*r.dir, r.start-s.sphere\_position);

float qc = dot(r.start-s.sphere\_position, r.start-s.sphere\_position) - s.sphere\_radius\*s.sphere\_radius;

float qd = qb \* qb - 4 \* qa \* qc;

Collision c;

c.inside = false;

……

}

Collision get\_closest\_collision(Ray r,sphere a,sphere b) { //返回射线最近的碰撞

Collision closest\_collision, cSph\_a, cSph\_b, cBox, cRBox, cPlane;

closest\_collision.object\_index = -1;

cSph\_a = intersect\_sphere\_object(r,a);

……

cPlane = intersect\_plane\_object(r);

if ((cSph\_a.t > 0) &&

((cSph\_a.t < cBox.t) || (cBox.t < 0))

&& ((cSph\_a.t < cRBox.t) || (cRBox.t < 0))

&& ((cSph\_a.t < cSph\_b.t) || (cSph\_b.t < 0))

&& ((cSph\_a.t < cPlane.t) || (cPlane.t < 0)))

{ closest\_collision = cSph\_a;

closest\_collision.object\_index = 1;

}

……

if ((cSph\_b.t > 0) && //添加判断射线最近的碰撞物是否为新添加球体

((cSph\_b.t < cBox.t) || (cBox.t < 0))

&& ((cSph\_b.t < cSph\_a.t) || (cSph\_a.t < 0))

&& ((cSph\_b.t < cRBox.t) || (cRBox.t < 0))

&& ((cSph\_b.t < cPlane.t) || (cPlane.t < 0))){

closest\_collision = cSph\_b;

closest\_collision.object\_index = 5;

}

return closest\_collision;

}

vec3 ads\_phong\_lighting(Ray r, Collision c,sphere a,sphere b)

{ //计算入射光线r在物体表面的环境漫反射(ADS) Phong照明。返回颜色。

vec4 ambient = worldAmb\_ambient + pointLight\_ambient \* objMat\_ambient;

vec4 diffuse = vec4(0.0);

vec4 specular = vec4(0.0);

Ray light\_ray;

light\_ray.start = c.p + c.n \* 0.01;

light\_ray.dir = normalize(pointLight\_position - c.p);

bool in\_shadow = false;

Collision c\_shadow = get\_closest\_collision(light\_ray,a,b);

……

}

void process\_stack\_element(int index,sphere a,sphere b){

if (popped\_stack\_element != null\_stack\_element) {

if (popped\_stack\_element.type == RAY\_TYPE\_REFLECTION)

stack[index].reflected\_color = popped\_stack\_element.final\_color;

else if (popped\_stack\_element.type == RAY\_TYPE\_REFRACTION)

stack[index].refracted\_color = popped\_stack\_element.final\_color;

popped\_stack\_element = null\_stack\_element;

}

Ray r = stack[index].ray;

Collision c = stack[index].collision;

switch (stack[index].phase)

{ // PHASE 1 -光线追踪碰撞检测

case 1:

c = get\_closest\_collision(r,a,b); // 对场景投射光线，存储碰撞结果

if (c.object\_index != -1)

stack[index].collision = c;

break;

// PHASE 2 - Phong ADS照明计算

case 2:

stack[index].phong\_color = ads\_phong\_lighting(r, c,a,b);

break;

// PHASE 3 -反射反弹通过计算

……

case 6: { popped\_stack\_element = pop(); return; } // 当所有五个阶段都完成时，结束递归

}

stack[index].phase++;

return; //每个process\_stack\_element()调用只处理一个阶段

}

vec3 raytrace(Ray r,sphere a,sphere b){ //模拟对任何所需深度的光线追踪的递归调用

push(r, 0, RAY\_TYPE\_REFLECTION);

while (stack\_pointer >= 0) {

int element\_index = stack\_pointer; // Peek at the topmost stack element

process\_stack\_element(element\_index,a,b); // Process this stack element

}

return popped\_stack\_element.final\_color;

}

void main()

{

sphere a,b;

a.sphere\_radius = 1.2;

a.sphere\_position = vec3(0.7, 0.2, 2.0);

a.sphere\_color = vec3(1.0, 0.0, 0.0); // red

b.sphere\_radius = 0.5;

b.sphere\_position = vec3(0.9, 0.5, 3.0);

b.sphere\_color = vec3(0.0, 0.0, 1.0); // blue

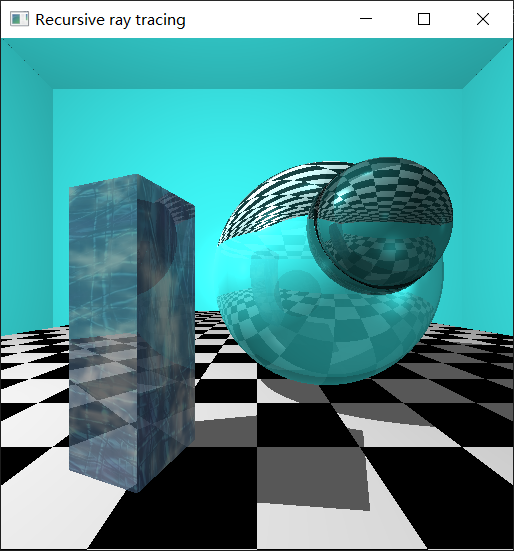
……

vec3 color = raytrace(world\_ray,a,b);

imageStore(output\_texture, pixel, vec4(color,1.0));

}

**运行结果：**



四、思考题

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

（1）freeglut

通过使用库中的glutInitWindowSize()、glutInitWindowPosition()、glutDisplayFunc()等函数设置初始窗口位置和大小，以及使用glutDisplayFunc()自动调用display函数重绘窗口。

（2）glew

通过在创建GL上下文后调用glewInit()函数使得不是直接与 GL 函数链接，而是在初始化阶段后获取函数指针，允许在运行时检查存在哪些扩展以及可以使用哪些功能。

（3）glfw

通过使用glfwWindowHint()、glfwSetWindowSizeCallback()等函数，单独设置整数值提示,实现显示窗口和捕捉窗口事件，提供了一些渲染物体所需的最低限度的接口。

（4）glad

glad是目前最新，也是最流行的库，用来管理OpenGL的函数指针，使OpenGL基础渲染变得简单。

**2.利用着色器实现光线追踪的优缺点**？

（1）光线追踪的优点

光线追踪基于几何光学原理，通过模拟光的传播路径，来确定反射、折射和阴影等，由于每个像素都单独计算，因此它能够很好地表现曲面细节。光线跟踪技术为整体光照模型提供了一种简单有效的绘制手段，能够生成高度真实的图形。

（2）光线追踪的缺点

光线追踪算法的计算量非常大，需要运用其他的技术加以改进。

（3）关于光线追踪算法的计算量及改进

光线跟踪算法，本质上是一个递归算法，每个像素的颜色和光强必须综合各级递归计算的结果才能获得。

光线跟踪算法中最核心的运算是求交。早期算法中约有95%的时间用于光线与物体表面的求交计算。所以想要优化该算法的时间，就需要设计高效率的求交算法：包围盒技术。

考虑到有时候射线与物体相距甚远，不必具体计算它们的交点，只要判断出它们不可能相交，即可以不必求交点，这就是包围盒技术或称空间细分技术，即将相邻物体用一个包围盒包起来，然后用光线与包围盒求交，若无交点，则无须对被包围的物体进行求交测试。