

深圳大学实验报告

课程名称： 计算机图形学

实验项目名称： 实验三 光照与阴影

学院： 计算机与软件学院

专业： 软件工程（腾班）

指导教师： 熊卫丹

报告人： 黄亮铭 学号： 2022155028 班级： 腾班

实验时间： 2024 年 11 月 21 日 - 2024 年 11 月 27 日

实验报告提交时间： 2024 年 11 月 27 日

教务部制

实验目的与要求:

1. 掌握 OpenGL 三维场景的读取与绘制方法, 理解光照和物体材质对渲染结果的影响, 强化场景坐标系转换过程中常见矩阵的计算方法, 熟悉阴影的绘制方法。
2. 创建 OpenGL 绘制窗口, 读入三维场景文件并绘制。
3. 设置相机并添加交互, 实现从不同位置/角度、以正交或透视投影方式观察场景。
4. 实现 Phong 光照效果和物体材质效果。
5. 自定义投影平面 (为计算方便, 推荐使用 $y=0$ 平面), 计算阴影投影矩阵, 为三维物体生成阴影。
6. 使用鼠标点击 (或其他方式) 控制光源位置并更新光照效果, 并同时更新三维物体的阴影。

实验过程及内容:

1.Camera 类

定义一个相机所需要的参数: 位置参数、观察方向向量、观察正向向量和位于相机胶平面的向量。

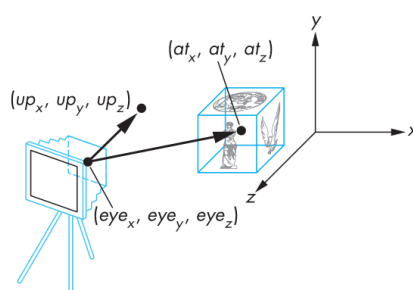


图 1 相机类

1.1 设置相机位置和视图平面

实现思路:

- a) 我们首先要初始化一个四维的单位矩阵。
- b) 然后根据相机的位置 eye 和物体中心 (参考点) at 计算 VPN 并归一化处理。使用数学公式表示为:

$$VPN = at - eye$$
$$n = \frac{VPN}{|VPN|}$$

- c) 再通过 at 和 VPN 生成与两者均垂直的方向向量并对其归一化, 同时计算得到

up（相机胶片平面的上方）在相机胶片平面上的投影。使用数学公式表示为：

$$u = \frac{n \times up}{|n \times up|}$$
$$v = \frac{u \times n}{|u \times n|}$$

- d) 接下来我们将相机从坐标原点移动到视点，这一步骤使用实验 2 中的平移矩阵可以完成。使用数学公式表示为：

$$\begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

代码实现： glm 库中已有相应的函数实现，因此这里我们可以选择直接调用对应的函数。

```
glm::mat4 Camera::lookAt(const glm::vec4& eye, const glm::vec4& at, const glm::vec4& up)
{
    // use glm.
    glm::vec3 eye_3 = eye;
    glm::vec3 at_3 = at;
    glm::vec3 up_3 = up;

    glm::mat4 view = glm::lookAt(eye_3, at_3, up_3);

    return view;
}
```

图 2lookAt 函数

1.2 相机视角的旋转和平移

实现思路：

是对eye每个分量进行正确的赋值，根据图 3，我们可以得出正确的赋值公式，下面将依次给出每个分量的赋值公式。

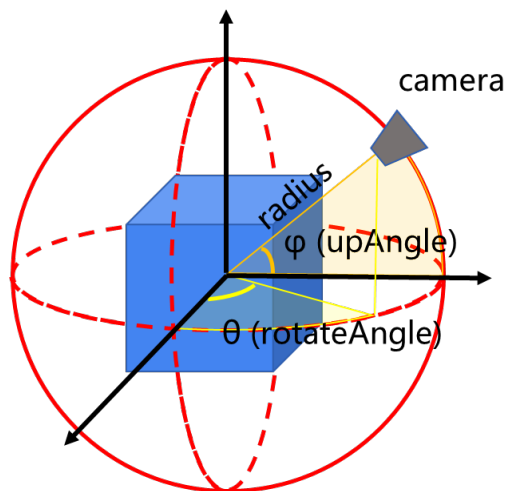


图 3：相机位置

- 对于 eye_x ， $eye_x = radius * \cos \varphi * \sin \theta$ 。
- 对于 eye_y ， $eye_y = radius * \sin \varphi$ 。
- 对于 eye_z ， $eye_z = radius * \cos \varphi * \cos \theta$ 。

如果使用相对于 at 的角度控制相机，在 upAngle 大于 90 时候，相机坐标系的 up 向量就会变成相反的方向。因此，我们在对 up 向量赋值时，需要判断 upAngle 是否大于 90。如果 upAngle 大于 90（小于-90），则将 up 向量反向；如果 upAngle 小于或等于 90，则不进行操作。

实现代码：代码实现如下图所示。

```
// 计算相机的 up 向量
up = glm::vec4(0.0, 1.0, 0.0, 0.0);
if (upAngle > 90) {
    up.y = -1;
}
else if (upAngle < -90) {
    up.y = -1;
}

float eyex = radius * cos(upAngle * M_PI / 180.0) * sin(rotateAngle * M_PI / 180.0);
float eyey = radius * sin(upAngle * M_PI / 180.0);
float eyez = radius * cos(upAngle * M_PI / 180.0) * cos(rotateAngle * M_PI / 180.0);

eye = glm::vec4(eyex, eyey, eyez, 1.0);
at = glm::vec4(0.0, 0.0, 0.0, 1.0);
// up = vec4(0.0, 1.0, 0.0, 0.0);
```

图 4updateCamera 函数

1.3 投影

投影变换的目的则是定义一个视景体，使得视景体外多余的部分被裁减掉，最终进入到投影平面上的只是视景体内的部分。投影包含正交投影和透视投影两种。

两种投影的矩阵如下图所示。

$$N = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 5a 正交投影

$$N = \begin{bmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$top = near * \tan\left(\frac{fov}{2}\right)$$

$$right = top * aspect$$

图 5b 透视投影

根据上述矩阵，我们可以编写出正交投影函数和透视投影函数。需要注意的是，glm 中的矩阵和数组矩阵是对称的，因此在初始化矩阵后还需要对矩阵实施转置的操作。

具体实现代码如下所示。

```
glm::mat4 Camera::ortho(const GLfloat left, const GLfloat right,
    const GLfloat bottom, const GLfloat top,
    const GLfloat zNear, const GLfloat zFar)
{
    glm::mat4 c = glm::mat4(1.0f);
    c[0][0] = 2.0 / (right - left);
    c[1][1] = 2.0 / (top - bottom);
    c[2][2] = -2.0 / (zFar - zNear);
    c[3][3] = 1.0;
    c[0][3] = -(right + left) / (right - left);
    c[1][3] = -(top + bottom) / (top - bottom);
    c[2][3] = -(zFar + zNear) / (zFar - zNear);

    c = glm::transpose(c);
    return c;
}
```

图 6a 正交投影函数

```
glm::mat4 Camera::perspective(const GLfloat fovy, const GLfloat aspect,
    const GLfloat zNear, const GLfloat zFar)
{
    GLfloat top = tan(fovy * M_PI / 180 / 2) * zNear;
    GLfloat right = top * aspect;

    glm::mat4 c = glm::mat4(1.0f);
    c[0][0] = zNear / right;
    c[1][1] = zNear / top;
    c[2][2] = -(zFar + zNear) / (zFar - zNear);
    c[2][3] = -(2.0 * zFar * zNear) / (zFar - zNear);
    c[3][2] = -1.0;
    c[3][3] = 0.0;

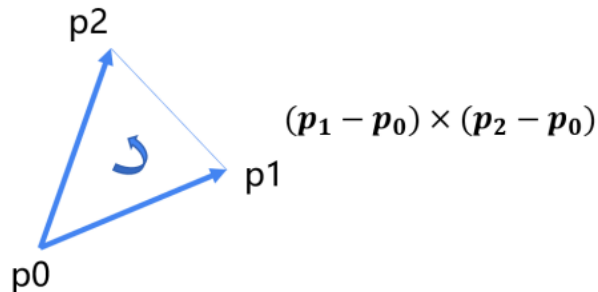
    c = glm::transpose(c);
    return c;
}
```

图 6b 透视投影函数

2Phong 光照模型

实现 Phong 光照模型可以分成如下几个核心部分：计算由三个顶点组成三角面片的法向量、计算顶点法向量、向顶点着色器传送数据、顶点着色器向片元着色器传送数据和在片元着色器中完成计算。

2.1 计算三角面片法向量



计算由3个顶点 p_0, p_1, p_2 组成的三角面片的面法向量

图 7 计算面片法向量方法

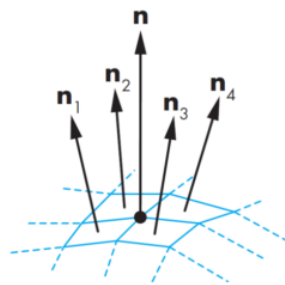
我们需要根据面片的顶点的下表找到相应的顶点 p_0, p_1, p_2 。然后计算面片的法向量，计算公式为 $n = (p_1 - p_0) \times (p_2 - p_0)$ 。最后我们需要对向量 n 进行归一化操作。对于每一个面片，上述操作都需要进行一次。

综上所述，实现的核心代码如下图所示。

```
void TriMesh::computeTriangleNormals()
{
    // 这里的resize函数会给face_normals分配一个和faces一样大的空间
    face_normals.resize(faces.size());
    for (size_t i = 0; i < faces.size(); i++) {
        auto& face = faces[i];
        // @TODO: Task1 计算每个面片的法向量并归一化
        glm::vec3 norm;
        glm::vec3 p[3] = { vertex_positions[face.x],
                          vertex_positions[face.y],
                          vertex_positions[face.z] };
        norm = glm::normalize(glm::cross((p[1] - p[0]), (p[2] - p[0])));
        face_normals[i] = norm;
    }
}
```

图 8 代码实现

2.2 计算顶点法向量



$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}$$

再根据顶点相邻的所有面片的面法向量计算出顶点法向量

图 9：计算顶点法向量方法

一个顶点会被很多面片拥有。因此，计算该顶点处的法向量相当于计算包含该顶点的所有面片的法向量之和，最后对其进行归一化处理。

综上所述，实现的核心代码如下图所示。

```
void TriMesh::computeVertexNormals()
{
    // 计算面片的法向量
    if (face_normals.size() == 0 && faces.size() > 0) {
        computeTriangleNormals();
    }

    // 这里的resize函数会给vertex_normals分配一个和vertex_positions一样大的空间
    // 并初始化法向量为0
    vertex_normals.resize(vertex_positions.size(), glm::vec3(0, 0, 0));
    // @TODO: Task1 求法向量均值
    for (size_t i = 0; i < faces.size(); i++) {
        auto& face = faces[i];
        // @TODO: 先累加面的法向量
        vertex_normals[face.x] += face_normals[i];
        vertex_normals[face.y] += face_normals[i];
        vertex_normals[face.z] += face_normals[i];
    }
    // @TODO 对累加的法向量归一化
    for (size_t i = 0; i < vertex_normals.size(); i++) {
        vertex_normals[i] = glm::normalize(vertex_normals[i]);
    }
}
```

图 10 代码实现

2.3 向顶点着色器传送数据

该部分在`bindObjectAndData`函数实现。在该函数中，我需要完成的任务有两个：1）将向量数据传送到 glfw 的缓存中；2）仿照初始化顶点坐标，从顶点着色器中初始化顶点的法向量。

对于第一个任务，我们需要将向量数据的起始地址的指针以及向量数据的大小作为参数输入到`glBufferSubData`函数中。

对于第二个任务，我首先使用`glGetAttribLocation`函数获取顶点着色器中法向量的存储地址。然后允许传送数据，最后将缓存中的数据传送到相应的地址中。

具体代码如下图所示。

```

// @TODO: Task1 修改完TriMesh.cpp的代码后再打开下面注释, 否则程序会报错
glBufferSubData(GL_ARRAY_BUFFER, (mesh->getPoints().size() + mesh->getColors().size()) * 3, 0, mesh->getColors());

object.vshader = vshader;
object.fshader = fshader;
object.program = InitShader(object.vshader.c_str(), object.fshader.c_str());

// 从顶点着色器中初始化顶点的坐标
object.pLocation = glGetUniformLocation(object.program, "vPosition");
glEnableVertexAttribArray(object.pLocation);
glVertexAttribPointer(object.pLocation, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

// 从顶点着色器中初始化顶点的颜色
object.cLocation = glGetUniformLocation(object.program, "vColor");
glEnableVertexAttribArray(object.cLocation);
glVertexAttribPointer(object.cLocation, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(mesh->getPoints().size() * 3));

// @TODO: Task1 从顶点着色器中初始化顶点的法向量
object.nLocation = glGetUniformLocation(object.program, "vNormal");
glEnableVertexAttribArray(object.nLocation);
glVertexAttribPointer(object.nLocation, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET((mesh->getPoints().size() + mesh->getColors().size()) * sizeof(glm::vec3)));

```

图 11 代码实现

2.4 顶点着色器向片元着色器传送数据

该部分的实现非常简单。在顶点着色器中，我们已经定义了具有 in 属性的变量接收来自缓存的数据。因此，我们需要定义具有 out 属性的变量，然后将 in 属性的变量赋值给 out 属性的变量即可完成顶点着色器向片元着色器传送数据的过程。

2.5 在片元着色器中完成计算

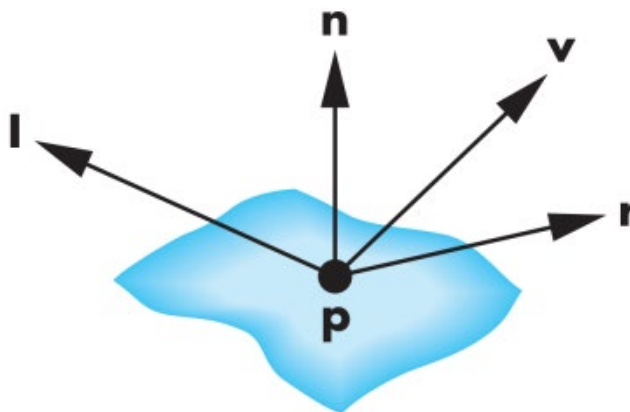


图 12: 反射模型

其中 p 为三维物体表面上的一点，l 是从点 p 指向光源位置的向量，n 表示 p 点的法向量，v 是从 p 点指向相机（观察者）的向量，r 是沿着 l 方向入射光线按照反射定律的出射方向。

为了简单考虑，我们这里假设衰减系数 $\frac{1}{a+bd+cd^2} = 1$ 。

我们的任务为：1) 计算上述四个向量并进行归一化处理；2) 根据下面的公式计算漫反射分量和镜面反射分量。

$$I = \frac{1}{a + bd + cd^2} (k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha + k_a L_a)$$

该等式右边一共有三部分，第一部分为漫反射，第二部分为镜面反射，第三部分为环境光。

在片元着色器完善和修改的代码如下图橙色框所示。

```
// @TODO: 计算四个归一化的向量 N,V,L,R (或半角向量H)
vec3 N = normalize(norm);
vec3 V = normalize(eye_position - position);
vec3 L = normalize(light.position - position);
vec3 R = reflect(-L, N);

// 环境光分量I_a
vec4 I_a = light.ambient * material.ambient;

// @TODO: Task2 计算系数和漫反射分量I_d
float diffuse_dot = 0.0;
diffuse_dot = max(dot(L, N), 0);
vec4 I_d = diffuse_dot * light.diffuse * material.diffuse;

// @TODO: Task2 计算系数和镜面反射分量I_s
float specular_dot_pow = 0.0;
specular_dot_pow = pow(max(dot(R, V), 0), material.shininess);
vec4 I_s = specular_dot_pow * light.specular * material.specular;

// @TODO: Task2 计算高光系数beta和镜面反射分量I_s
// 注意如果光源在背面则去除高光
if( dot(L, N) < 0.0 ) {
    I_s = vec4(0.0, 0.0, 0.0, 1.0);
}

// 合并三个分量的颜色, 修正透明度
fColor = I_a + I_d + I_s;
fColor.a = 1.0;
```

图 13: 完善片元着色器

3 生成阴影

理论知识: 以 $z=0$ 为投影平面例。

下面我们来推导阴影投影矩阵如下。假设光源位置在 (x_l, y_l, z_l) ，三角形任意一个顶点坐标为 (x, y, z) ，投影到投影平面之后的坐标为 (x_k, y_k, z_k) ，因为该点在 $z = 0$ 平面上，所以 $z_k = 0$ 。根据比例关系可得如下公式：

$$\frac{x_l - x}{x - x_k} = \frac{z_l - z}{z - z_k}$$

求解可得：

$$x_k = \frac{x_l z - x z_l}{z - z_l}$$

同理可得：

$$y_k = \frac{y_l z - y z_l}{z - z_l}$$

为了能够方便的通过矩阵表示出投影关系，我们将所有坐标设置在齐次坐标系下，那么投影关系就能表示成如下公式：

$$\begin{pmatrix} x'_k \\ y'_k \\ z'_k \\ w'_k \end{pmatrix} = \begin{pmatrix} x_l z - x z_l \\ y_l z - y z_l \\ 0 \\ z - z_l \end{pmatrix} = \begin{pmatrix} -z_l & 0 & x_l & 0 \\ 0 & -z_l & y_l & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -z_l \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

综上所述，我们得到了阴影的投影矩阵： $\begin{pmatrix} -z_l & 0 & x_l & 0 \\ 0 & -z_l & y_l & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -z_l \end{pmatrix}$ 。需要注意的是 glm

库中的矩阵是数组矩阵的转置。

在完成投影矩阵的求解之后，我们将投影矩阵右乘物体的模型矩阵即可得到物体阴影的模型矩阵。然后仿照物体的绘制方法绘制阴影即可。

代码实现：

```

glm::mat4 Light::getShadowProjectionMatrix() {
    // 这里只实现了Y=0平面上的阴影投影矩阵，其他情况自己补充
    float lx, ly, lz;

    glm::mat4 modelMatrix = this->getModelMatrix();
    glm::vec4 light_position = modelMatrix * glm::vec4(this->translation, 1.0);

    lx = light_position[0];
    ly = light_position[1];
    lz = light_position[2];

    // ...
    return glm::mat4(
        -lz, 0.0, 0.0, 0.0,
        0, -lz, 0.0, 0.0,
        lx, ly, 0.0, 1.0,
        0.0, 0.0, 0.0, -lz
    );
}

```

图 14 阴影投影矩阵

```

// @MYDO
modelMatrix = light->getShadowProjectionMatrix() * modelMatrix;
// 传递矩阵
glUniformMatrix4fv(mesh_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
// 将着色器 isShadow 设置为0，表示正常绘制的颜色，如果是1着表示阴影
glUniform1i(mesh_object.shadowLocation, 1);
// 绘制
glDrawArrays(GL_TRIANGLE_FAN, 0, mesh->getPoints().size());
// @END

```

图 15 绘制物体阴影

4 鼠标、键盘交互

4.1 鼠标交互

我们可以使用鼠标控制光源的位置（x 方向和 y 方向），z 方向是固定的（光源的高度固定）。具体代码实现如下图所示。

```

void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        double x, y;
        glfwGetCursorPos(window, &x, &y);

        float half_winx = WIDTH / 2.0;
        float half_winy = HEIGHT / 2.0;
        float lx = float(x - half_winx) / half_winx;
        float ly = float(HEIGHT - y - half_winy) / half_winy;

        glm::vec3 pos = light->getTranslation();

        pos.x = lx;
        pos.y = ly;

        light->setTranslation(pos);
    }
}

```

图 16 键盘回调函数

4.2 键盘交互

- 1) 我们可以通过键盘的 U、I 和 O 键控制相机的旋转角度、俯仰角度和相机与原点的距离。同时，我们也可以通过键盘的空格键重新初始化。

```

if (key == GLFW_KEY_U && action == GLFW_PRESS && mode == 0x0000)
{
    rotateAngle += 5.0;
}
else if (key == GLFW_KEY_U && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    rotateAngle -= 5.0;
}
else if (key == GLFW_KEY_I && action == GLFW_PRESS && mode == 0x0000)
{
    upAngle += 5.0;
    if (u (字段) float Camera::upAngle
        联机搜索
    )
    {}
}
else if (key == GLFW_KEY_I && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    upAngle -= 5.0;
    if (upAngle < -180)
        upAngle = -180;
}
else if (key == GLFW_KEY_O && action == GLFW_PRESS && mode == 0x0000)
{
    radius += 0.1;
}
else if (key == GLFW_KEY_O && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    radius -= 0.1;
}
else if (key == GLFW_KEY_SPACE && action == GLFW_PRESS && mode == 0x0000)
{
    radius = 4.0;
    rotateAngle = 0.0;
    upAngle = 0.0;
    fov = 45.0;
    aspect = 1.0;
    scale = 1.5;
}
}

```

图 17 控制相机

- 2) 我们可以通过键盘的 Q、W、A 和 S 控制输入模型。

```
else if (key == GLFW_KEY_Q && action == GLFW_PRESS)
{
    std::cout << "read sphere.off" << std::endl;
    mesh = new TriMesh();
    mesh->readOff("./assets/sphere.off");
    init();
}
else if (key == GLFW_KEY_A && action == GLFW_PRESS)
{
    std::cout << "read Pikachu.off" << std::endl;
    mesh = new TriMesh();
    mesh->readOff("./assets/Pikachu.off");
    init();
}
else if (key == GLFW_KEY_W && action == GLFW_PRESS)
{
    std::cout << "read Squirtle.off" << std::endl;
    mesh = new TriMesh();
    mesh->readOff("./assets/Squirtle.off");
    init();
}
else if (key == GLFW_KEY_S && action == GLFW_PRESS)
{
    std::cout << "read sphere_coarse.off" << std::endl;
    mesh = new TriMesh();
    mesh->readOff("./assets/sphere_coarse.off");
    init();
}
```

图 18 控制模型输入

- 3) 我们可以通过键盘的数字键 1-3 控制环境光的数值，数字键 4-6 控制漫反射的数值，数字键 7-9 控制镜面反射的数值。这里以控制环境光的代码为示例进行展示，其余部分的代码均类似。

```
else if (key == GLFW_KEY_1 && action == GLFW_PRESS && mode == 0x0000)
{
    ambient = mesh->getAmbient();
    tmp = ambient.x;
    ambient.x = std::min(tmp + 0.1, 1.0);
    mesh->setAmbient(ambient);
}
else if (key == GLFW_KEY_1 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    ambient = mesh->getAmbient();
    tmp = ambient.x;
    ambient.x = std::max(tmp - 0.1, 0.0);
    mesh->setAmbient(ambient);
}
else if (key == GLFW_KEY_2 && action == GLFW_PRESS && mode == 0x0000)
{
    ambient = mesh->getAmbient();
    tmp = ambient.y;
    ambient.y = std::min(tmp + 0.1, 1.0);
    mesh->setAmbient(ambient);
}
else if (key == GLFW_KEY_2 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    ambient = mesh->getAmbient();
    tmp = ambient.y;
    ambient.y = std::max(tmp - 0.1, 0.0);
    mesh->setAmbient(ambient);
}
else if (key == GLFW_KEY_3 && action == GLFW_PRESS && mode == 0x0000)
{
    ambient = mesh->getAmbient();
    tmp = ambient.z;
    ambient.z = std::min(tmp + 0.1, 1.0);
    mesh->setAmbient(ambient);
}
else if (key == GLFW_KEY_3 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    ambient = mesh->getAmbient();
    tmp = ambient.z;
    ambient.z = std::max(tmp - 0.1, 0.0);
    mesh->setAmbient(ambient);
}
```

图 19 控制环境光

- 4) 我们可以通过键盘的数字键 0 控制高光系数的数值。

```
// 高光指数
else if (key == GLFW_KEY_0 && action == GLFW_PRESS && mode == 0x0000)
{
    shininess = mesh->getShininess();
    shininess = shininess + 1;
    shininess = glm::clamp(shininess, 0.1f, 100.0f);
    mesh->setShininess(shininess);
}
else if (key == GLFW_KEY_0 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    shininess = mesh->getShininess();
    shininess = shininess - 1;
    shininess = glm::clamp(shininess, 0.1f, 100.0f);
    mesh->setShininess(shininess);
}
```

图 20 控制高光系数

- 5) 我们还可以按下键盘的 ESC 键退出程序，按下键盘的 H 键获取帮助信息。



```
if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
{
    glfwSetWindowShouldClose(window, GL_TRUE);
}
else if (key == GLFW_KEY_H && action == GLFW_PRESS)
{
    printHelp();
}
```

图 21 其他键盘反馈

5 其他

5.1 材质应用

我在该网站 <http://www.it.hiof.no/~borres/j3d/explain/light/p-materials.html> 找到了如下材质对象。我们可以根据网站给出的材质的环境光反射、漫反射、镜面反射和高光系数等，为自己的模型赋值。

	<pre>//Perl float[] mat_ambient = { 0.25f, 0.20725f, 0.20725f, 0.922f }; float[] mat_diffuse = { 1.0f, 0.829f, 0.829f, 0.922f }; float[] mat_specular = { 0.296648f, 0.296648f, 0.296648f, 0.922f }; float shine = 11.264f;</pre>
	<pre>//Turquoise float[] mat_ambient = { 0.1f, 0.18725f, 0.1745f, 0.8f }; float[] mat_diffuse = { 0.396f, 0.74151f, 0.69102f, 0.8f }; float[] mat_specular = { 0.297254f, 0.30829f, 0.306678f, 0.8f }; float shine = 12.8f;</pre>

6 结果展示

6.1 控制光源位置

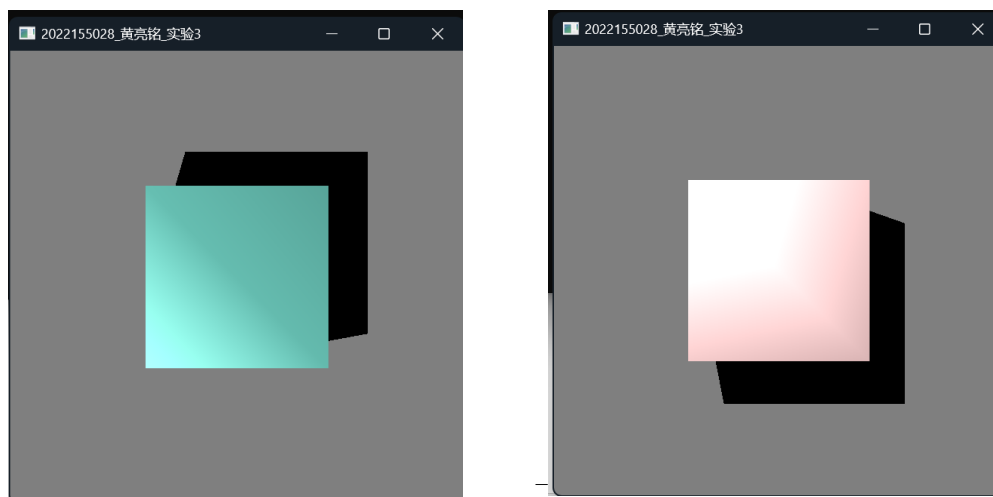


图 22 光源位置（左：左下 右：左上）

6.2 控制相机位置

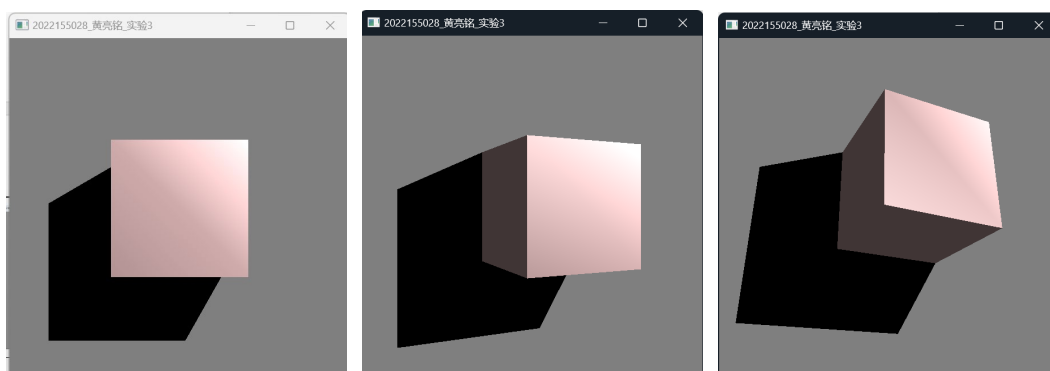


图 23a 相机位置（左：初始 中：改变 rotate 右改变 rotate 和 up）

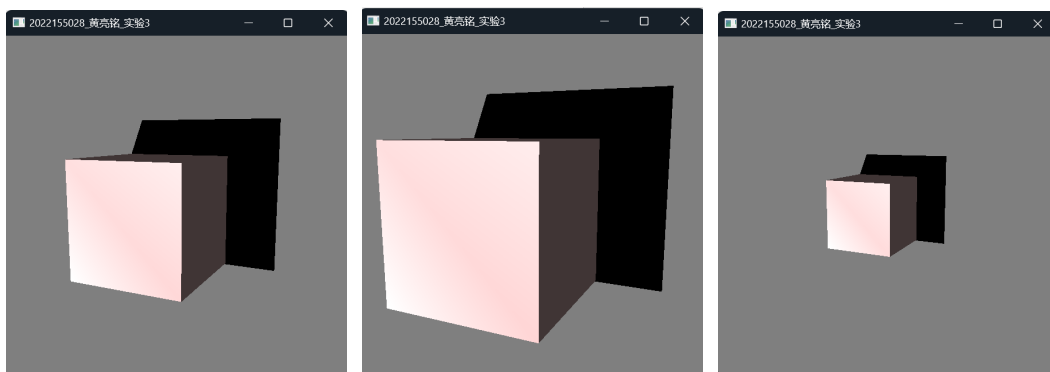


图 23b 相机位置（左：初始 中：减小 radius 右增大 radius）

6.3 改变形状、环境光反射

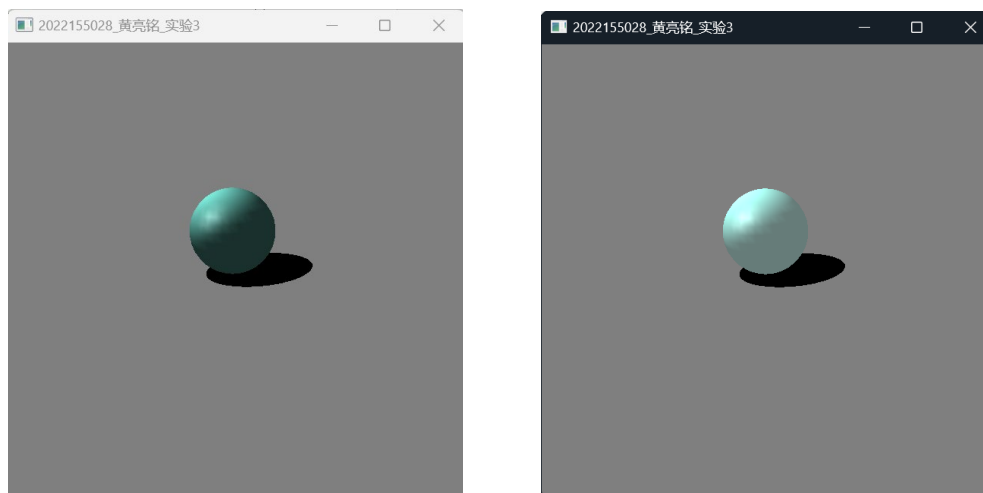


图 24（左：初始 右：增加环境光反射）

6.4 改变形状、漫反射

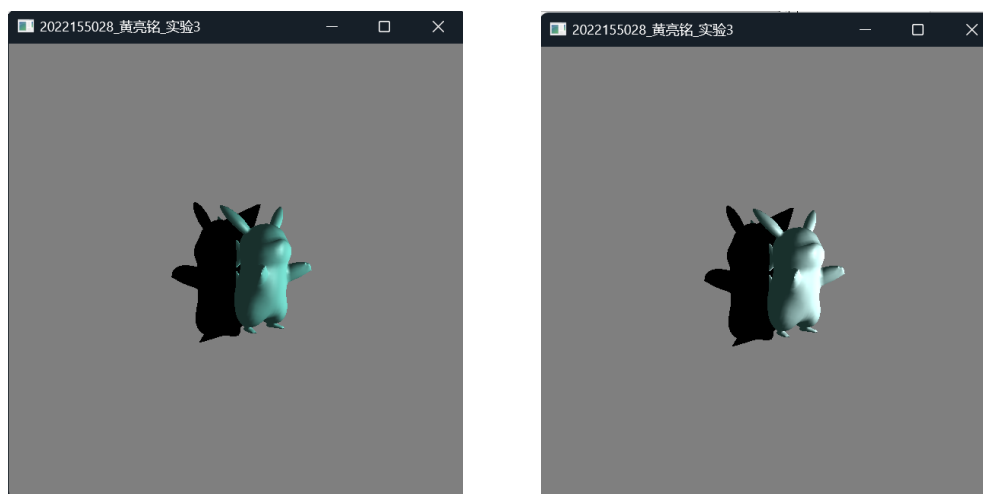


图 25（左：初始 右：增加漫反射）

6.5 改变形状、镜面反射、高光系数



图 26（左：初始 中：增加镜面反射 右：增加镜面反射、高光系数）

实验结论:

- 1) 实验结果表明,我成功地设置相机并添加交互,实现从不同位置/角度、以正交或透视投影方式观察场景。此外,我们通过鼠标实现光照位置的改变。
- 2) 通过本次实验,我成功地掌握了 OpenGL 三维场景的读取与绘制方法,深入理解了光照和物体材质对渲染结果的影响,并强化了场景坐标系转换过程中常见矩阵的计算方法。
- 3) 在实现 Phong 光照效果和物体材质效果的过程中,我通过计算三角面片的法向量、顶点法向量,并在顶点着色器和片元着色器中完成光照计算,成功地模拟了光照效果。
- 4) 通过本次实验,我了解了如何计算阴影投影矩阵,为三维物体成功生成阴影。

- ### 实验结论:
- 1) 实验结果表明,我成功地设置相机并添加交互,实现从不同位置/角度、以正交或透视投影方式观察场景。此外,我们通过鼠标实现光照位置的改变。
 - 2) 通过本次实验,我成功地掌握了 OpenGL 三维场景的读取与绘制方法,深入理解了光照和物体材质对渲染结果的影响,并强化了场景坐标系转换过程中常见矩阵的计算方法。
 - 3) 在实现 Phong 光照效果和物体材质效果的过程中,我通过计算三角面片的法向量、顶点法向量,并在顶点着色器和片元着色器中完成光照计算,成功地模拟了光照效果。
 - 4) 通过本次实验,我了解了如何计算阴影投影矩阵,为三维物体成功生成阴影。

指导教师批阅意见：

成绩评定：

指导教师签字：
年 月 日

指导教师批阅意见：

成绩评定：

指导教师签字：
年 月 日

指导教师批阅意见：

成绩评定：

指导教师签字：
年 月 日

指导教师批阅意见：

成绩评定：

指导教师签字：
年 月 日

备注:

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。