

# 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： 计算机科学与技术

指导教师： 周虹

报告人： 林宪亮 学号： 2022150130 班级： 国际班

实验时间： 2024 年 10 月 22 日 -- 2024 年 11 月 25 日

实验报告提交时间： 2024 年 11 月 18 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. 绘制场景、模型

### 1.1 读入三维场景文件

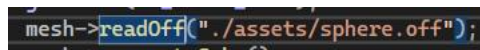


图 1 文件读取

如图 1，使用 readoff 函数读取物体文件，readoff 具体实现如下：

```
void TriMesh::readOff(const std::string& filename)
{
    // fin 打开文件读取文件信息
    if (filename.empty())
    {
        return;
    }
    std::ifstream fin;
    fin.open(filename);
    // 此函数读取 OFF 文件中三维模型的信息
    if (!fin)
    {
        printf("File on error\n");
        return;
    }
    else
    {
        printf("File open success\n");
        cleanData();

        int nVertices, nFaces, nEdges;

        // 读取 OFF 字符串
        std::string str;
        fin >> str;
        // 读取字符串中的顶点数、面片数、边数
        fin >> nVertices >> nFaces >> nEdges;
        // 根据顶点数，循环读取每个顶点坐标
        for (int i = 0; i < nVertices; i++)
        {
            glm::vec3 tmp_node;
            fin >> tmp_node.x >> tmp_node.y >> tmp_node.z;
            vertex_positions.push_back(tmp_node);
            vertex_colors.push_back(tmp_node);
        }
        // 根据面片数，循环读取每个面片信息，并用构建的 vec3 结构体保存
        for (int i = 0; i < nFaces; i++)
        {
            int num, a, b, c;
            // num 记录该面片由几个顶点构成，a、b、c 为构成该面片顶点序号
            fin >> num >> a >> b >> c;
            faces.push_back(vec3i(a, b, c));
        }
    }
    fin.close();
    storeFacePoints();
};
```

图 2 readoff 实现

这个函数 readOff 用于读取指定 OFF 文件中的三维模型数据。首先，我打开文件

并验证其是否有效，如果文件无法打开，我会输出错误信息并返回。接着，我清理已有数据以确保不受旧数据影响。然后，从文件中依次读取顶点数、面片数和边数，并根据顶点数读取每个顶点的坐标，将其存储到 `vertex_positions` 和 `vertex_colors` 中。对于面片数据，我读取每个面片的顶点索引，将其存储到 `faces` 中以表示面片的几何结构。最后，我调用 `storeFacesPoints` 来处理 and 存储读取的面片信息，完成三维模型的初始化。

## 1.2 背景色设置为灰色。

```
// 设置灰色背景
glClearColor(0.5, 0.5, 0.5, 1.0);
```

图 3 背景色设置

如图 3，我将背景色设置为灰色。

## 2. 设置相机

### 2.1 设置相机并添加交互，实现从不同位置/角度。

```
void Camera::keyboard(int key, int action, int mode)
{
    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 (upAngle > 180)
            upAngle = 180;
    }
    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 4 相机交互

当按下 U 键时，我可以调整旋转角度，普通模式下增加 5 度，按下 Shift 键则减少 5 度；按下 I 键可以调整垂直角度，同样通过普通模式和 Shift 模式分别增加或减少 5 度，并限制角度范围在 -180 到 180 度之间；按下 O 键可以调整摄像机与原点的半径，普通模式增加 0.1，Shift 模式减少 0.1；按下空格键时，我可以将摄像机参数重置为默认值，包括半径、角度、视场角、宽高比和缩放比例。通过这些操作，我可以方便地控制摄像机视角，灵活调整场景显示效果。

## 2.2 正交或透视投影方式观察场景

### 2.2.1 正交投影

```
glm::mat4 Camera::ortho(const GLfloat left, const GLfloat right,
    const GLfloat bottom, const GLfloat top,
    const GLfloat zNear, const GLfloat zFar)
{
    // 创建一个单位矩阵作为正交投影矩阵的初始值
    glm::mat4 orthoMatrix(1.0f);

    // 计算投影矩阵的元素值
    orthoMatrix[0][0] = 2.0f / (right - left);    // X 缩放因子
    orthoMatrix[1][1] = 2.0f / (top - bottom);    // Y 缩放因子
    orthoMatrix[2][2] = -2.0f / (zFar - zNear);    // Z 缩放因子 (注意为负值)
    orthoMatrix[3][0] = -(right + left) / (right - left);    // X 平移因子
    orthoMatrix[3][1] = -(top + bottom) / (top - bottom);    // Y 平移因子
    orthoMatrix[3][2] = -(zFar + zNear) / (zFar - zNear);    // Z 平移因子

    // 返回计算好的正交投影矩阵
    return orthoMatrix;
}
```

图 5 正交投影函数

这个函数主要为了实现下面的正交投影矩阵：

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

图 6 正交投影矩阵

首先，我创建了一个单位矩阵作为初始值。接下来，我计算了 x、y、z 方向的缩放因子，分别将它们填入矩阵的对角线上。x 和 y 方向的缩放通过视锥体的左右、上下边界计算，而 z 方向则通过近平面和远平面计算，注意这里 z 轴的缩放因子为负值。然后，我计算了平移因子，以确保整个视锥体被正确地投影到坐标系的中心。最后，我返回计算好的正交投影矩阵。

### 2.2.2 透视投影

```
glm::mat4 Camera::perspective(const GLfloat fov, const GLfloat aspect,
    const GLfloat zNear, const GLfloat zFar)
{
    // 创建一个单位矩阵作为透视投影矩阵的初始值
    glm::mat4 perspectiveMatrix(1.0f);

    // 计算透视投影矩阵的元素值
    float f = 1.0f / tan(glm::radians(fov / 2.0f)); // 计算焦距
    perspectiveMatrix[0][0] = f / aspect;          // X 缩放因子
    perspectiveMatrix[1][1] = f;                  // Y 缩放因子
    perspectiveMatrix[2][2] = (zFar + zNear) / (zNear - zFar); // Z 缩放因子
    perspectiveMatrix[2][3] = -1.0f;              // Z 平移因子 (注意为负值)
    perspectiveMatrix[3][2] = (2.0f * zFar * zNear) / (zNear - zFar); // Z 平移因子
    perspectiveMatrix[3][3] = 0.0f;                // 齐次坐标

    // 返回计算好的透视投影矩阵
    return perspectiveMatrix;
}
```

图 7 透视投影函数

这个函数主要是实现下面的矩阵：

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

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

$$right = top * aspect$$

图 8 透视投影矩阵

首先，我创建了一个单位矩阵 `perspectiveMatrix` 作为初始值。然后，我通过焦距的计算来获取视场的缩放因子，焦距使用 `fov`（视场角）通过正切函数得到，并将其填入矩阵的相应位置。接着，我设置了 `x` 轴和 `y` 轴的缩放因子，其中 `x` 轴缩放因子还考虑了宽高比 `aspect`。对于 `z` 轴，我计算了缩放因子并设置了适当的平移因子，确保透视效果正确。最后，我返回计算好的透视投影矩阵，使得 3D 场景中的物体在投影到视口时具备深度感。

通过改变 `display` 函数中：

`camera->projMatrix = camera->getProjectionMatrix(false)` 传入的参数即可选择使用正交投影还是透视投影。

### 3. 添加光照和材质效果

#### 3.1 实现 Phong 光照效果

##### 3.1.1 计算法向量

```
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

        //获取顶点
        glm::vec3 v0 = vertex_positions[face.x];
        glm::vec3 v1 = vertex_positions[face.y];
        glm::vec3 v2 = vertex_positions[face.z];

        //计算平面法向量
        glm::vec3 edge1 = v1 - v0;
        glm::vec3 edge2 = v2 - v0;
        glm::vec3 norm = glm::cross(edge1, edge2);

        //glm::vec3 norm;
        // face_normals[i] = norm;

        //归一化
        norm = glm::normalize(norm);

        //存储
        face_normals[i] = norm;
    }
}
```

图 9 法向量计算

在这个函数中，我实现了对三角形网格中每个面的法向量的计算和归一化。具体步骤如下：

首先，我使用 `resize` 函数给 `face_normals` 分配了一个和 `faces` 数组大小相同的空间，以存储每个面的法向量。

然后，我遍历了 `faces` 数组中的每个面片 `face`，从 `vertex_positions` 数组中提取该面的三个顶点 `v0`、`v1` 和 `v2` 的位置。

使用顶点位置，计算出了两个边向量 `edge1` 和 `edge2`，分别表示从 `v0` 到 `v1` 和从 `v0` 到 `v2` 的向量。

通过 `glm::cross` 函数，我计算出 `edge1` 和 `edge2` 的叉积，这个结果向量 `norm` 就是该面片的法向量。

为了确保法向量的单位长度，我使用 `glm::normalize` 对 `norm` 进行了归一化处理。

最后，将归一化后的法向量 `norm` 存储到 `face_normals[i]` 中，以便后续使用。

这样，所有三角形面的法向量都被计算并存储在 `face_normals` 数组中。

### 3.1.2 计算顶点法向量

```
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.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 计算顶点法向量

在函数中，我计算了三角形网格中每个顶点的法向量。实现过程如下：

首先，我检查 `face_normals` 是否为空。如果为空且 `faces` 数组不为空，就调用 `computeTriangleNormals()` 来计算每个面的法向量。

接着，使用 `resize` 函数给 `vertex_normals` 分配了和 `vertex_positions` 相同大小的空间，并将每个顶点的法向量初始化为 `(0, 0, 0)`。

然后，我遍历 `faces` 数组。对于每个面片 `face`，我累加与该面相关的法向量 `face_normals[i]` 到对应的三个顶点 `vertex_normals[face.x]`、`vertex_normals[face.y]` 和 `vertex_normals[face.z]` 上。这一步相当于将每个面片的法向量平均分配给其三个顶点。

最后，为了确保每个顶点的法向量的单位长度，我遍历 `vertex_normals`，并用 `glm::normalize` 对每个顶点的法向量进行了归一化处理。

通过这种方式，我得到了每个顶点的平均法向量，适合用于光照计算等需要平滑过渡的场景。



### 3.1.3 初始化顶点法向量

```
// @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 初始化顶点法向量

在这段代码中，我在顶点着色器中初始化了顶点的法向量，具体步骤如下：

首先，我使用 `glGetAttribLocation` 函数获取了 `vNormal` 属性的位置 `nLocation`。  
`vNormal` 是顶点着色器中用于接收顶点法向量的属性名称。

然后，我调用 `glEnableVertexAttribArray` 函数，启用 `nLocation` 位置上的顶点属性数组，以便在绘制时将数据传递给着色器。

接下来，我使用 `glVertexAttribPointer` 来定义顶点法向量的数据格式和内存位置。  
这里的关键参数包括：

`nLocation`：指定了顶点属性位置。

3：表示每个法向量包含三个浮点数（x, y, z 分量）。

`GL_FLOAT`：数据类型为浮点数。

`GL_FALSE`：数据不需要标准化。

0：表示每个属性之间没有间隔。

`BUFFER_OFFSET(...)`：设置了法向量数据在缓冲区中的偏移量，以确保正确读取。

在 `BUFFER_OFFSET` 中，我计算了法向量的偏移量，将顶点数据和颜色数据的大小相加，再乘以 `sizeof(glm::vec3)` 来得到法向量数据的起始位置。这种布局将顶点位置、颜色和法向量数据连续存储在一个缓冲区中，使得 `OpenGL` 能够顺利地将这些属性传递给着色器。

### 3.2 片元着色器

```
// 将顶点坐标、光源坐标和法向量转换到相机坐标系
vec3 norm = (vec4(normal, 0.0)).xyz;

// @TODO: Task1 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 = max(dot(L, N), 0.0);
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(max(dot(R, V), 0.0), material.shininess);
vec4 I s = specular_dot * 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 (diffuse_dot <= 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 12 fshader.glsl

这段代码实现了一个简单的光照着色模型，用于计算片段的最终颜色 `fColor`。在实现过程中，我首先判断当前片段是否在阴影中：如果 `isShadow` 等于 1，我直接将片段

颜色设置为纯黑色 (0.0, 0.0, 0.0, 1.0)。

如果片段不在阴影中，我将顶点坐标、光源坐标和法向量转换到相机坐标系，并计算出四个归一化的向量：法向量  $N$ 、视线向量  $V$ （从片段位置到观察者的位置）、光源方向向量  $L$ （从片段位置指向光源的位置）以及反射向量  $R$ （光源方向向量  $L$  关于法向量  $N$  的反射）。

然后，我依次计算光照的三个分量：

环境光分量 ( $I_a$ )：直接将光源的环境光强度和材质的环境反射系数相乘。

漫反射分量 ( $I_d$ )：先计算光源方向向量  $L$  和法向量  $N$  的点积（取最大值为 0），再乘以光源的漫反射强度和材质的漫反射系数。

镜面反射分量 ( $I_s$ )：先计算反射向量  $R$  和视线向量  $V$  的点积，取非负值并根据材质的高光系数  $shininess$  计算幂次。最终，将结果与光源的镜面反射强度和材质的镜面反射系数相乘。

为了保证高光效果的正确性，我检查了光源是否在法向量背面。如果是，我将镜面反射分量直接置为零。

最后，我将环境光、漫反射和镜面反射分量相加，得到片段的最终颜色  $fColor$ ，并将其透明度设置为 1.0，确保颜色完全不透明。

### 3.3 物体材质效果。

```
// 设置材质(自定义)
mesh->setAmbient(glm::vec4(0.2, 0.2, 0.2, 1.0)); // 环境光
mesh->setDiffuse(glm::vec4(0.7, 0.7, 0.7, 1.0)); // 漫反射
mesh->setSpecular(glm::vec4(0.2, 0.2, 0.2, 1.0)); // 镜面反射
mesh->setShininess(50.0); //高光系数
```

图 13 物体材质效果

我定义了自己的物体的材质效果。

## 4. 添加阴影效果

### 4.1 投影平面

```
// 生成平面
plane->generateSquare(glm::vec3(0.7, 0.7, 0.7));
// 设置平面的旋转位移
plane->setTranslation(glm::vec3(0.0, -0.001, 0.0));
plane->setRotation(glm::vec3(0, 90, 90));
plane->setScale(glm::vec3(3.0, 3.0, 3.0));
```

图 14 投影平面

首先，我调用 `generateSquare` 方法，以灰色（RGB 值为 0.7, 0.7, 0.7）的颜色生成一个平面。接下来，我通过设置平面的平移、旋转和缩放属性来调整它在场景中的位置和外观：将平面沿  $y$  轴方向微微下移到 -0.001 的位置，随后将其绕  $x$  和  $y$  轴分别旋转 90 度，使其适应场景的需求。最后，我将平面的尺寸按比例放大到 3 倍，以便更好地填充视图或场景。这一系列操作使平面在三维空间中具备准确的位置、方向和比例，满足场景布局的要求。



## 4.2 阴影效果

```
// 绘制
glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());

glm::vec3 light_pos = light->getTranslation();
float lx = light_pos[0];
float ly = light_pos[1];
float lz = light_pos[2];
glm::mat4 shadowProjMatrix(-ly, 0.0, 0.0, 0.0,
    lx, 0.0, lz, 1.0,
    0.0, 0.0, -ly, 0.0,
    0.0, 0.0, 0.0, -ly);

modelMatrix = shadowProjMatrix * modelMatrix;
glUniform1i(mesh_object.shadowLocation, 1);
glUniformMatrix4fv(mesh_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
glUniformMatrix4fv(mesh_object.viewLocation, 1, GL_FALSE, &camera->viewMatrix[0][0]);
glUniformMatrix4fv(mesh_object.projectionLocation, 1, GL_FALSE, &camera->projMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
```

图 15 绘制阴影

首先，我获取光源的位置 `lightPosition`，用于计算阴影的投影矩阵。通过光源的 `x`、`y`、`z` 坐标，我定义了一个 `shadowMatrix`，这是一个简化的投影矩阵，用来模拟光源下物体在某平面上的阴影投影效果。

接着，我将原始的 `modelMatrix`（模型矩阵）与 `shadowMatrix` 相乘，从而生成变换后的模型矩阵，使得物体被投影到光源方向上的平面。

然后，我通过 OpenGL 的统一变量传递计算好的矩阵和其他参数：将着色器中的 `isShadow` 变量设置为 1，告知渲染器这是一个阴影片段。分别传递模型矩阵、视图矩阵和投影矩阵，确保阴影正确映射到屏幕空间。

最后，我调用 `glDrawArrays` 使用顶点数据绘制阴影，绘制模式为三角形，并根据模型中的顶点数量进行渲染。通过这一流程，我完成了基于光源位置生成的阴影效果绘制，让场景中的光照更具真实感。

## 5. 交互控制光源位置并更新阴影

5.1 参考实验 2.1，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。

```
else if (key == GLFW_KEY_4 && action == GLFW_PRESS && mode == 0x0000)
{
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.x;
    diffuse.x = std::min(tmp + 0.1, 1.0);
    mesh->setDiffuse(diffuse);
}
else if (key == GLFW_KEY_4 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.x;
    diffuse.x = std::max(tmp - 0.1, 0.0);
    mesh->setDiffuse(diffuse);
}
else if (key == GLFW_KEY_5 && action == GLFW_PRESS && mode == 0x0000)
{
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.y;
    diffuse.y = std::min(tmp + 0.1, 1.0);
    mesh->setDiffuse(diffuse);
}
else if (key == GLFW_KEY_5 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.y;
    diffuse.y = std::max(tmp - 0.1, 0.0);
    mesh->setDiffuse(diffuse);
}
else if (key == GLFW_KEY_6 && action == GLFW_PRESS && mode == 0x0000)
{
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.z;
    diffuse.z = std::min(tmp + 0.1, 1.0);
    mesh->setDiffuse(diffuse);
}
```

图 16 key\_callback (1)

在这段代码中，我实现了对物体材质的反射系数和高光指数的交互控制，使用户可

以动态调整环境光、漫反射、镜面反射以及高光效果。具体的交互方式如下：

#### 按键 4-9 控制反射系数：

键位 4 到 6 分别调整漫反射颜色的 x, y, z 分量。

键位 7 到 9 分别调整镜面反射颜色的 x, y, z 分量。

每个按键按下时，可以增加（不加 Shift）或减少（按住 Shift）对应分量的值。通过 `glm::min` 和 `glm::max`，确保值在 `[0.0, 1.0]` 范围内。

```
else if (key == GLFW_KEY_9 && action == GLFW_PRESS && mode == 0x0000)
{
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.z;
    specular.z = std::min(tmp + 0.1, 1.0); // 增加 z 分量
    mesh->setSpecular(specular);
}

else if (key == GLFW_KEY_9 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT)
{
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.z;
    specular.z = std::max(tmp - 0.1, 0.0); // 减少 z 分量
    mesh->setSpecular(specular);
}

// 最后，按键 0 用于增减物体的高光指数 (shininess)，影响高光的集中度
else if (key == GLFW_KEY_0 && action == GLFW_PRESS)
{
    float shininess = mesh->getShininess();
    float delta = 1.0; // 设置高光指数的增减步长

    // 根据是否按下了 Shift 键增加或减少高光指数
    if (mode == GLFW_MOD_SHIFT) {
        shininess -= delta; // 减少高光指数
    }
    else {
        shininess += delta; // 增加高光指数
    }

    // 将高光指数限制在合理范围内 (0.1 到 100.0)
    shininess = glm::clamp(shininess, 0.1f, 100.0f);

    // 更新物体的高光指数
    mesh->setShininess(shininess);
}
```

图 17 key\_callback (2)

#### 按键 0 控制高光指数：

键位 0 用于增加或减少物体的高光指数 (shininess)，从而控制高光的集中程度。通常，高光指数越高，高光区域越小、越亮。

如果用户按住 Shift 键，则高光指数减少；否则，高光指数增加。

使用 `glm::clamp` 限制高光指数在 `[0.1, 100.0]` 范围内，避免指数过低或过高。

这些操作可以让在渲染窗口中直观地调节材质的反射特性和高光效果，以获得不同的视觉效果。

## 6. 实验结果

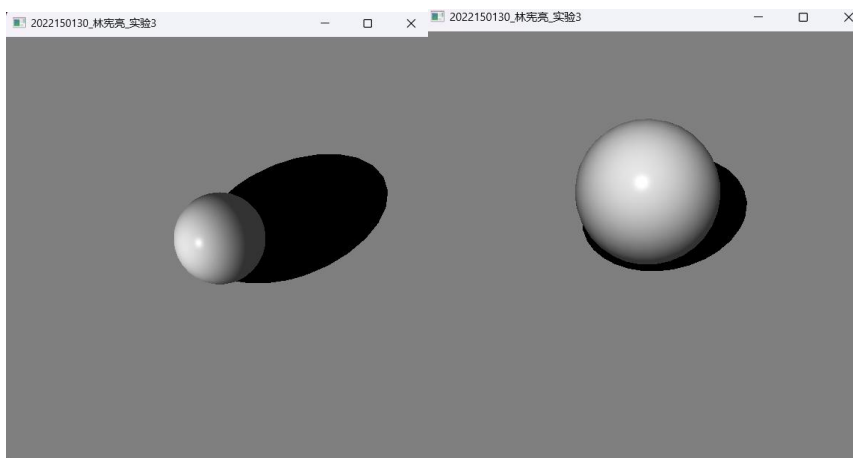


图 18 球形光照投影

如图 18，为球形的光照与投影的结果。

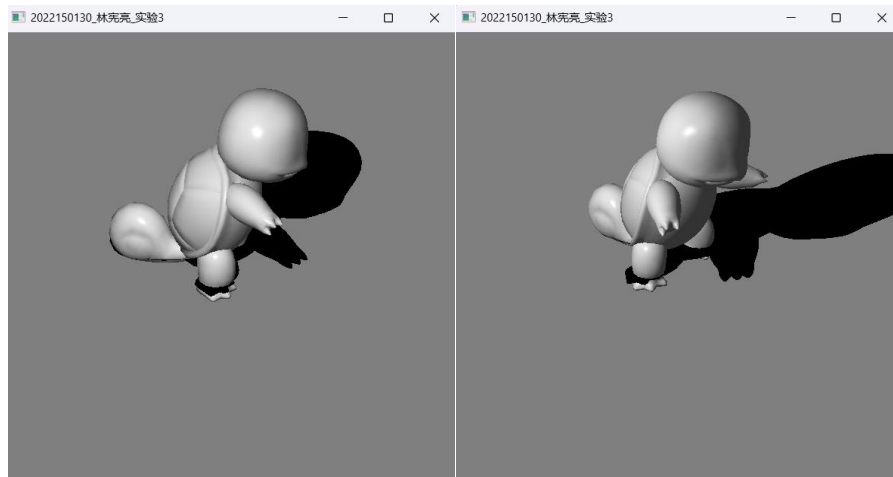


图 19 乌龟光照投影

如图 19，为杰尼龟的光照与投影图。

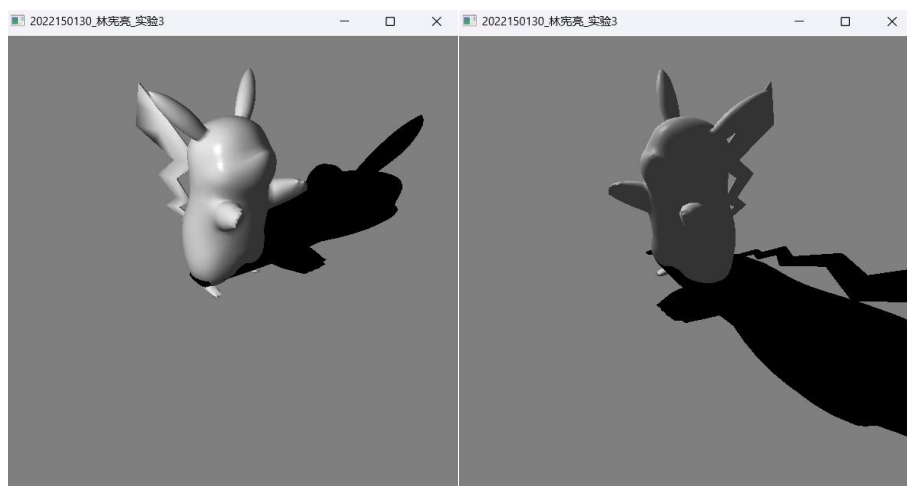


图 20 皮卡丘光照投影

如图 20，为皮卡丘的光照投影图。

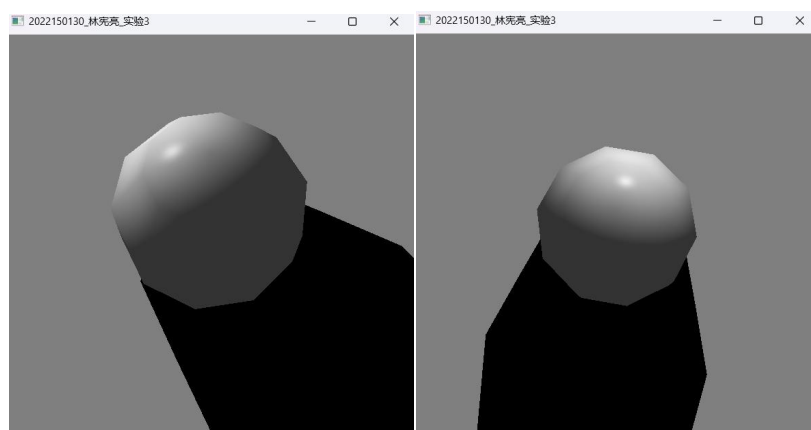


图 21 多面体光照投影

如图 22，为多面体的光照投影图。

### 实验结论:

本次实验围绕三维图形渲染的核心技术，依次完成了场景绘制、相机设置、光照与材质效果、阴影生成以及光源交互控制等功能。首先，我通过解析 .off 文件加载三维几何体数据，利用 OpenGL 的绘图接口绘制模型，并设置窗口背景为灰色，以区分物体和阴影。接着，我使用相机设置实现了从不同角度观察场景，并在正交投影与透视投影间动态切换。为了增强渲染效果，引入 Phong 光照模型，通过配置光源与材质参数，展现了真实的光影变化，通过实时调节材质参数，我观察并分析了环境光、漫反射和镜面反射的权重对物体颜色和亮度的影响。例如，增加漫反射强度能够让物体表现出更加明亮的效果，而提高镜面反射系数则使表面呈现更强的光泽感。这种动态调节让我深刻认识到材质参数如何直接影响最终的视觉表现。在此基础上，我使用光源位置和投影矩阵，在  $y=0$  平面生成物体阴影，呈现出逼真的场景投影效果。最后，我实现了便捷的键盘交互，通过不同按键对反射系数和高光指数进行实时控制。这种交互方式让我能够快速调整渲染参数，并观察到每次调整带来的视觉变化。实验过程加深了我对图形学理论的理解，让我全面掌握了光照模型的实现方法和材质参数的调节技巧，也加深了对光与材质相互作用的理解。通过实时交互调整光照参数，我对物体材质的光照效果有了更加直观和深入的认识，提升了我在 OpenGL 平台上的开发能力，为后续复杂三维场景的设计和实现打下了坚实基础。

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。