练 习 题 报 告

课程名称 计算机图形学

项目名称 Phong 光照模型（1）

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 吴嘉楷 学号 2022150168

一、 练习目的

1. 了解 OpenGL 中基本的光照模型
2. 掌握 OpenGL 中实现基于顶点的光照计算
3. 掌握法向量的计算

二．练习完成过程及主要代码说明

1. 在 Trimesh.cpp 中完善 computeTriangleNormals()

函数源码：

```
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
        // 获取面片的三个顶点
        glm::vec3 v0 = vertex_positions[face.x];
        glm::vec3 v1 = vertex_positions[face.y];
        glm::vec3 v2 = vertex_positions[face.z];
        // 计算面片的法向量
        glm::vec3 n = glm::cross(v1 - v0, v2 - v0);
        // 归一化
        glm::vec3 norm = glm::normalize(n);
        // 存储面片的法向量
        face_normals[i] = norm;
    }
}
```

函数说明：

computeTriangleNormals，用于计算每个三角形面片的法向量并将其归一化。

首先，通过调用 resize 函数，代码为 face_normals 分配了和 faces 数组相同大小的空间，用于存储每个面片的法向量。随后，使用 for 循环逐个访问 faces 数组中的每个面片，其中 faces[i] 表示第 i 个三角形面片，包含三个顶点的索引信息。

在循环体中，程序依次从 vertex_positions 数组中取出面片的三个顶点坐标，分别赋值给 v0、v1 和 v2。然后，通过计算两个顶点向量的叉积 glm::cross(v1 - v0, v2 - v0) 来得到当前面片的法向量 n。叉积结果代表了这两个向量垂直的方向，是该三角形面片的法向量。

接着，使用 glm::normalize 函数对计算出的法向量 n 进行归一化处理，得到单位长度的法向量 norm。最后，将归一化后的法向量存储在 face_normals 数组中对应的位置 face_normals[i]。

2. 在 Trimesh.cpp 中完善 computeVertexNormals()

函数源码：

```
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: 先累加面的法向量到顶点
        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]);
    }
}
```

代码说明：

computeVertexNormals 的目的是根据每个三角形面片的法向量来计算其相邻顶点的法向量，这样可以在渲染时获得更平滑的光照效果。

首先，函数检查 face_normals 数组是否为空，如果为空且 faces 数组有元素，就调用 computeTriangleNormals 函数来计算每个三角形面片的法向量，确保 face_normals 已经包含了所有面片的法向量。

接着，代码通过调用 resize 函数对 vertex_normals 数组分配空间，使其大小与 vertex_positions 相同，并将所有法向量初始化为 (0,0,0)(0, 0, 0)(0,0,0)。这一步是为了准备好存储每个顶点的法向量，之后会对其进行累加和归一化。

在主循环中，代码遍历 faces 数组中的每个三角形面片，通过 faces[i] 取得当前面片的三个顶点索引。然后将面片的法向量 face_normals[i] 分别累加到这三个顶点的法向量上。这种累加操作的原理是将面片法向量分配给它的相邻顶点，使得顶点法向量代表了与该顶点相邻面片的平均法向量方向。

在完成累加后，代码使用另一个循环遍历 vertex_normals 数组，将每个顶点的法向量进行归一化处理。归一化的目的是确保法向量的长度为 1，以便在光照计算中使用时不失真。这一过程完成后，vertex_normals 中的每个法向量都代表了顶点的法向量方向，是相邻面片法向量的平均结果，使得顶点法向量在渲染中能够更平滑地过渡。

3. 在 main.cpp 中完善 bindObjectAndData()

（1）打开 glBufferSubData 的注释

代码截图：

```
// @TODO: Task1 修改完TriMesh.cpp的代码成后再打开下面注释，否则程序会报错
glBufferSubData(GL_ARRAY_BUFFER, (mesh->getPoints().size() + mesh->getColors().size()) * sizeof(glm::vec3), mesh->getNormals().size() * sizeof(glm::vec3), &mesh->getNormals()[0]);
```

图 1 解开代码注释

（2）从顶点着色器中初始化顶点的法向量

代码截图：

```
// @TODO: Task1 从顶点着色器中初始化顶点的法向量
object.nLocation = glGetAttribLocation(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 2 设置法向量的顶点属性

代码说明：

首先，通过 glGetAttribLocation 函数获取着色器程序 object.program 中属性变量 "vNormal" 的位置。这个位置保存在 object.nLocation 中，用来指定顶点法向量的存储位置。

接着，调用 glEnableVertexAttribArray(object.nLocation) 启用这个属性数组，使其在渲染过程中可用。这一步表示在渲染时可以通过 vNormal 访问每个顶点的法向量数据。

最后，glVertexAttribPointer 函数指定了 object.nLocation 顶点属性的格式。它指出每个法向量包含三个分量（对应 x, y, z），每个分量的数据类型是浮点数，不需要进行归一化，并且各个法向量在缓冲区中的排列没有间隔。最后一个参数 BUFFER_OFFSET((mesh->getPoints().size() + mesh->getColors().size()) * sizeof(glm::vec3)) 用来确定法向量数据在缓冲区中的位置偏移量。它计算了顶点和颜色数据的总大小，假设法向量数据在这些数据之后存储，因此通过总大小来确定偏移量。

4. 在 vshader.glsl 中完善 main()

代码截图：

```
// @TODO: Task2 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 - pos);
vec3 L = normalize(l_pos - pos);
vec3 R = reflect(-L, N);

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

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

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

// 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);
}
```

图 3 完善 vshader 着色器的 main()函数

代码解释：

这段代码实现了基于光照模型的颜色计算，包括环境光、漫反射和镜面反射三个主要光

照分量。

首先，代码计算了四个归一化的向量 N、V、L 和 R。其中，N 表示法向量，它通过对 norm 向量归一化得到；V 表示从当前表面点 pos 到观察者位置 eye_position 的向量；L 表示从表面点到光源位置 l_pos 的方向向量；R 是反射向量，它通过反射计算函数 reflect 得到，代表光源方向 L 关于法向量 N 的镜面反射方向。

接着，计算环境光分量 I_a。环境光反映了物体表面在阴影区域的亮度，通常是光源和物体材质的环境光颜色的乘积。

然后计算漫反射系数和漫反射光分量 I_d。首先，通过 dot(N, L) 计算法向量 N 和光源方向 L 的点积，代表光源对表面的照射强度，并取其与 0 的最大值，以防止负值影响结果。I_d 是点积结果与光源的漫反射光颜色和材质的漫反射光颜色的乘积，这个分量会模拟光线垂直照射到物体表面时的亮度。

接下来，代码计算镜面反射系数和镜面反射光分量 I_s。通过 dot(R, V) 计算反射向量 R 和观察方向 V 的点积，代表视线对反射方向的对准程度，并使用 pow 函数对其进行幂运算以实现聚光效果，幂次为材质的光泽度 material.shininess。I_s 是该幂结果与光源的镜面反射光颜色和材质的镜面反射光颜色的乘积，这一分量会在视线方向接近反射方向时产生强烈的高光。

最后，代码检查 diffuse_dot 是否小于 0.0，用于判断光源是否在物体背面，如果是，则设置镜面反射分量 I_s 为零，这样就不会产生不合理的高光效果。

5. 在 main.cpp 中 init()函数里改变材质参数

代码截图：

```
// @TODO: Task3 请自己调整光源参数和物体材质参数来达到不同视觉效果
// 设置光源位置
light->setTranslation(glm::vec3(0.0, 0.0, 2.0));
light->setAmbient(glm::vec4(1.0, 1.0, 1.0, 1.0)); // 环境光
light->setDiffuse(glm::vec4(1.0, 1.0, 1.0, 1.0)); // 漫反射
light->setSpecular(glm::vec4(1.0, 1.0, 1.0, 1.0)); // 镜面反射

// 设置物体的旋转位移
mesh->setTranslation(glm::vec3(0.0, 0.0, 0.0));
mesh->setRotation(glm::vec3(0, 0.0, 0.0));
mesh->setScale(glm::vec3(1.0, 1.0, 1.0));

// 设置材质(自定义)
mesh->setAmbient(glm::vec4(0.1, 0.1, 0.1, 1.0)); // 环境光
mesh->setDiffuse(glm::vec4(0.8, 0.8, 0.8, 1.0)); // 漫反射
mesh->setSpecular(glm::vec4(0.2, 0.2, 0.2, 1.0)); // 镜面反射
mesh->setShininess(10.0); //高光系数
```

图 4 改变物体材质参数

代码说明：

这段代码通过设置环境光、漫反射、镜面反射和高光系数来定义物体的材质，使得光照效果更自然。

- **环境光**：setAmbient 函数用来定义物体在环境光下的颜色，这里为暗灰色 vec4(0.1, 0.1, 0.1, 1.0)，模拟了弱光中的柔和亮度。
- **漫反射光**：setDiffuse 定义了漫反射的颜色，即光线在物体表面散射的效果，设置为 vec4(0.8, 0.8, 0.8, 1.0)，让物体在光照下表现出较亮的颜色，使其更加生动。
- **镜面反射光**：setSpecular 定义了镜面反射，即高光区域的颜色和光泽度，这里为较低的 vec4(0.2, 0.2, 0.2, 1.0)，因此物体表面有柔和的光泽。
- **高光系数**：setShininess(10.0) 设置了高光系数，数值决定了高光的集中程度，较低值让

光晕较为分散，产生柔和的反射效果。

6. 同步修改“复原”的键盘交互逻辑

代码截图：

```
else if(key == GLFW_KEY_MINUS && action == GLFW_PRESS && mode == 0x0000)
{
    mesh->setAmbient(glm::vec4(0.1, 0.1, 0.1, 1.0));
    mesh->setDiffuse(glm::vec4(0.8, 0.8, 0.8, 1.0));
    mesh->setSpecular(glm::vec4(0.2, 0.2, 0.2, 1.0));
    mesh->setShininess(10.0);
}
```

图 5 修改复原的参数信息

7. 在 main.cpp 中 mainWindow_key_callback ()函数完善键盘交互

（1）调整漫反射系数： 如果用户按下数字键'4'至'6'，分别增减漫反射的红、绿、蓝分量

代码截图：

```
// @TODO: Task4 添加更多交互 1~9增减反射系数（1~3已写好），0增减高光指数
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);
}
else if (key == GLFW_KEY_6 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT) {
    glm::vec4 diffuse = mesh->getDiffuse();
    tmp = diffuse.z;
    diffuse.z = std::max(tmp - 0.1, 0.0);
    mesh->setDiffuse(diffuse);
}
```

图 6 调整漫反射系数的逻辑

代码说明：

按键 4、5 和 6 分别控制漫反射颜色的红、绿、蓝三个通道的强度值。代码通过检查键盘输入来判断是否增加或减少每个颜色通道的值。

当用户按下 4 键（无修饰键）时，获取当前漫反射颜色并增加红色通道的值 0.1，且确保不超过最大值 1.0。如果用户同时按下 Shift 键和 4，则减少红色通道的值 0.1，且保证不低于 0.0。键 5 和 6 的逻辑相同，分别对应绿色和蓝色通道。当按下 5 键时，它会增加或减少绿色通道的值；按下 6 键时，则调整蓝色通道的值。这样，用户可以通过按键实时控制物体的漫反射颜色，使得颜色在渲染过程中发生动态变化。

（2）调整镜面反射系数：若用户按下数字键'7'至'9'，分别增减镜面反射的红、绿、蓝分量
代码截图：

```
else if (key == GLFW_KEY_7 && action == GLFW_PRESS && mode == 0x0000) {
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.x;
    specular.x = std::min(tmp + 0.1, 1.0);
    mesh->setSpecular(specular);
}
else if (key == GLFW_KEY_7 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT) {
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.x;
    specular.x = std::max(tmp - 0.1, 0.0);
    mesh->setSpecular(specular);
}
else if (key == GLFW_KEY_8 && action == GLFW_PRESS && mode == 0x0000) {
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.y;
    specular.y = std::min(tmp + 0.1, 1.0);
    mesh->setSpecular(specular);
}
else if (key == GLFW_KEY_8 && action == GLFW_PRESS && mode == GLFW_MOD_SHIFT) {
    glm::vec4 specular = mesh->getSpecular();
    tmp = specular.y;
    specular.y = std::max(tmp - 0.1, 0.0);
    mesh->setSpecular(specular);
}
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);
    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);
    mesh->setSpecular(specular);
}
```

图 7 调整镜面反射系数的逻辑

代码说明：

按键 7、8 和 9 分别调整镜面反射颜色的红、绿、蓝通道。代码根据用户的输入来增加或减少每个颜色通道的强度。

当用户按下键 7（无修饰键）时，获取当前镜面反射颜色，并增加红色通道的值 0.1，且保证不超过 1.0。如果用户按下 Shift 和 7，则减少红色通道的值 0.1，并确保值不低于 0.0。按键 8 和 9 的逻辑与 7 类似，分别对应绿色和蓝色通道的调整。按 8 键时会增加或减少绿色通道的强度，按 9 键时则调整蓝色通道的强度。

（3）调整高光指数：若用户按下数字键‘0’，就会增减高光指数的值
代码截图：

```
else if (key == GLFW_KEY_0 && action == GLFW_PRESS && mode == 0x0000) {
    float shininess = mesh->getShininess();
    float delta = 0.1; // 根据需要调整增减的步长
    // 根据用户的操作增减高光指数
    if (mode == GLFW_MOD_SHIFT) {
        shininess -= delta;
    }
    else {
        shininess += delta;
    }
    // 更新高光指数
    shininess = glm::clamp(shininess, 0.1f, 100.0f);
    // 将新的高光指数设置到物体上
    mesh->setShininess(shininess);
}
```

图 8 调整高光指数

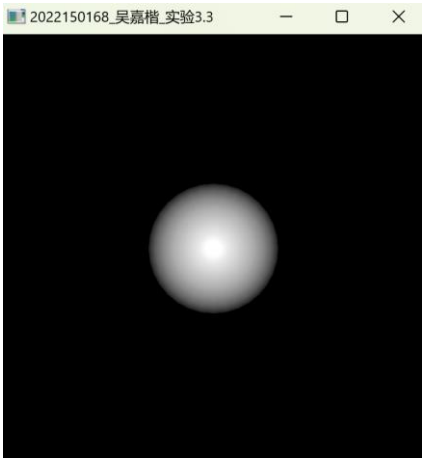
8. 修改运行窗口的标题、尺寸等属性

```
GLFWwindow* mainwindow = glfwCreateWindow(700, 700, "2022150168_吴嘉楷_实验3.3", NULL, NULL);
```

图 9 修改窗口配置

9. 运行结果

（1）初始效果　VS　增加旋转、up 角度，减少相机距离



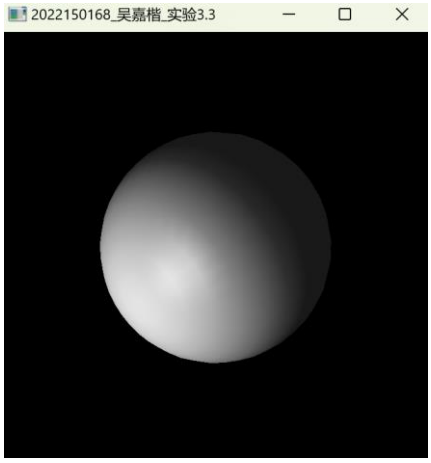
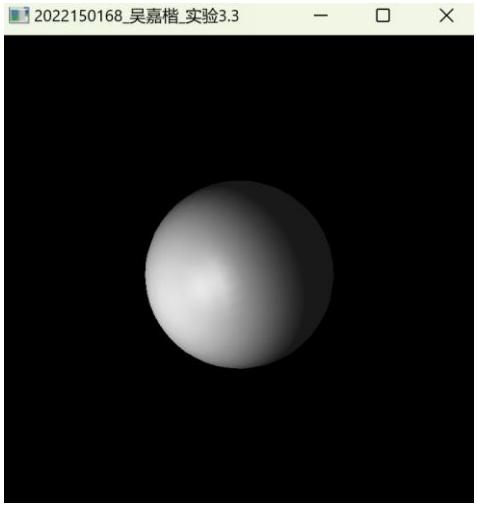


图 10.1.1 初始 sphere 模型的效果　　图 10.1.2 调整旋转、up 角度和相机距离后的效果

（2）默认材质　VS　键盘交互调整漫反射系数



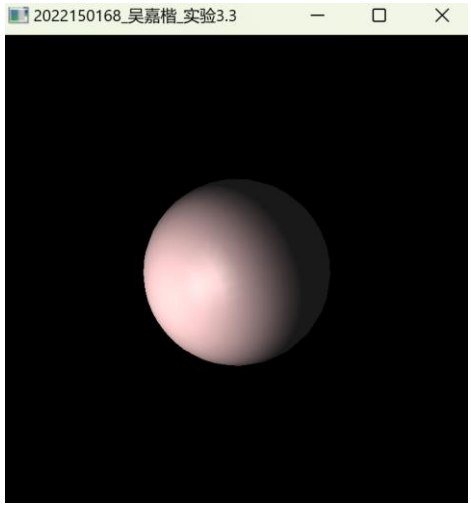
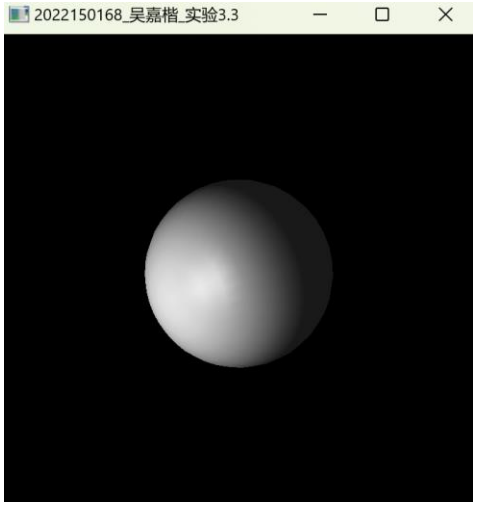


图 10.2.1 默认材质的显示效果　　图 10.2.2 按下数字键 4 调整漫反射系数后的效果

（3）默认材质　VS　键盘交互调整镜面反射系数



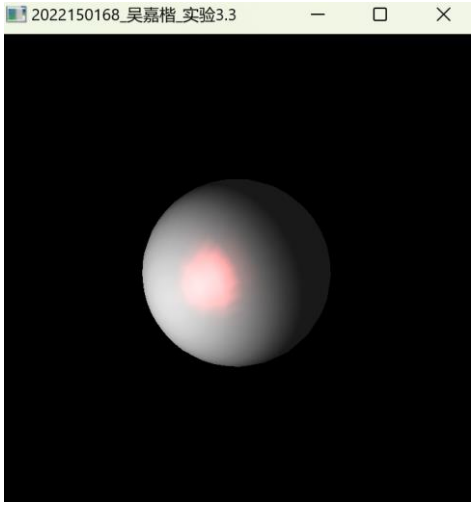


图 10.3.1 默认材质的显示效果　　图 10.3.2 按下数字键 7 调整镜面反射系数后的效果

（4）默认高光指数　　VS　　增加高光指数

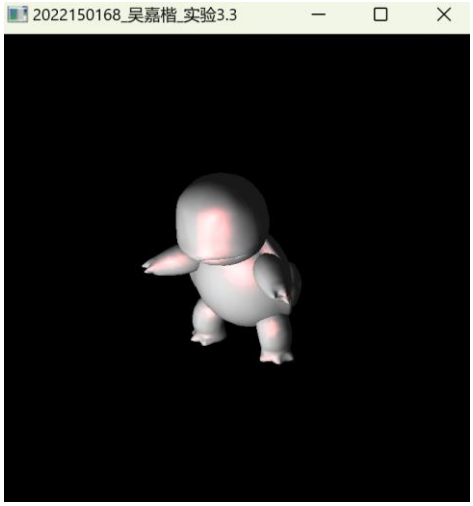


图 10.4.1 默认高光指数的效果

图 10.4.2 增加高光指数后的效果

（5）默认光源位置　　VS　　改变光源位置

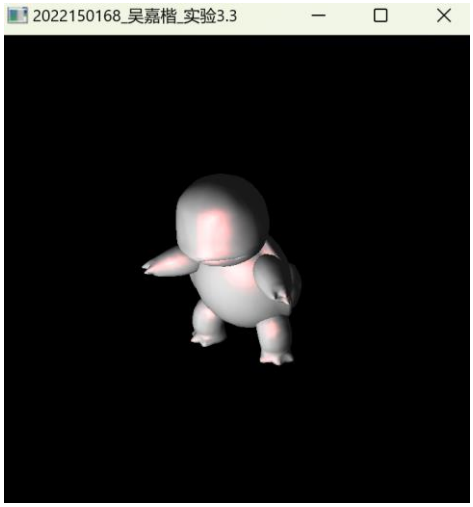


图 10.5.1 默认光源位置的效果

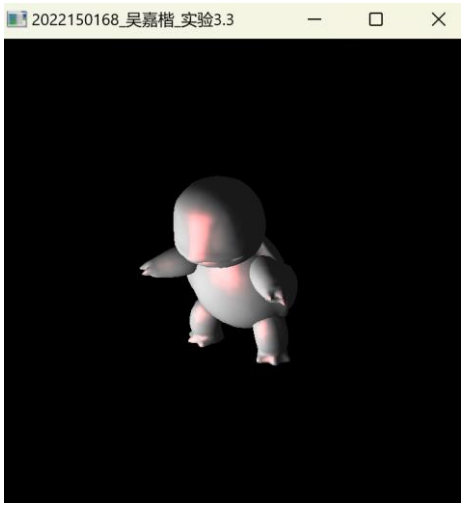


图 10.5.2 改变光源位置后的效果

（6）默认材质　　VS　　调整材质

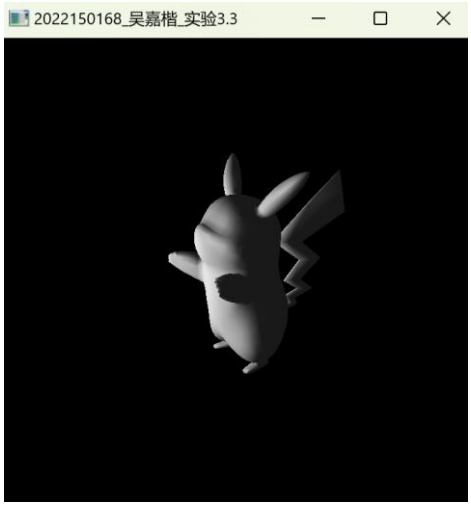


图 10.6.1 默认材质的效果

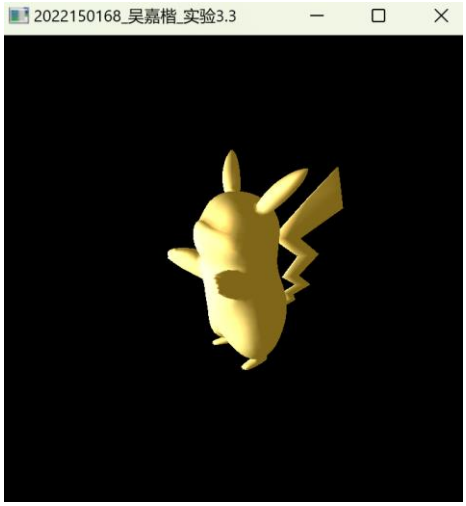


图 10.6.2 调整材质参数后的效果