深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇 二四 ～二〇 二五 学年度第 一 学期

课程编号 1500610003 课序号 04 课程名称 计算机图形学 主讲教师 周 虹，黄惠 评分

学 号 2022150168 姓 名 吴嘉楷 专业年级 2022 级计算机科学与技术

教师评语：

题 目： 隐士修所虚拟场景

宋体五号，至少八页，可以从下一页开始写。

成绩评分栏：

评分项	俄罗斯方块文档（占 12 分）	俄罗斯方块代码（占 24 分）	俄罗斯方块迟交倒扣分（占 0 分）	虚拟场景建模文档（占 16 分）	虚拟场景建模代码（占 38 分）	演示与答辩（占 10 分）	虚拟场景建模迟交倒扣分（占 0 分）	大作业总分
得分								
评分人								

一、实验内容

在屏幕上显示一个包含多个虚拟物体的虚拟场景，并且响应一定的用户交互操作。如以下例图：

二、具体内容

1. 场景设计和显示

学生可以通过层级建模（ 实验补充 1 和 2）的方式建立多个虚拟物体，由多个虚拟物体组成一个虚拟场景，要求在程序中显示该虚拟场景，场景可以是室内或者室外场景；场景应包含地面。层级建模的最深层次需要达到至少四层。

2. 添加纹理

参考实验 4.1，为场景中至少两个主要物体添加纹理贴图 。

3. 添加光照、材质、阴影效果

参考实验 3.2，实验 3.3 和实验 3.4，实现光照效果、材质、阴影等。

4. 用户交互实现视角切换完成对场景的任意角度浏览

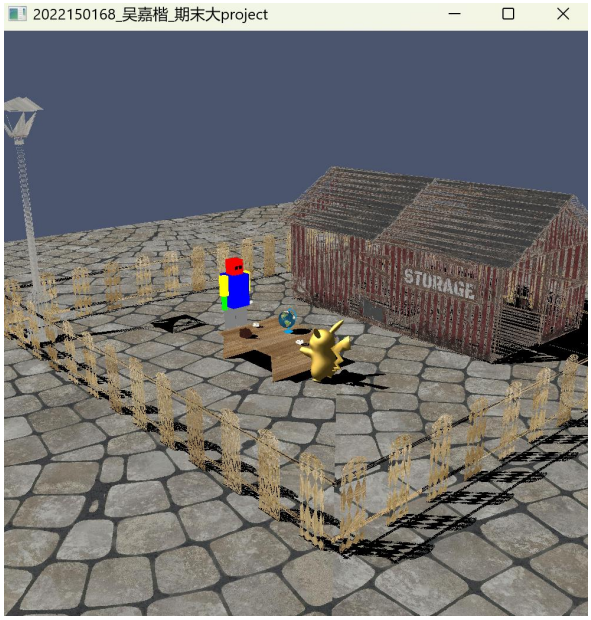
参考实验 3.1，完成相机变换。

5. 通过交互控制物体

参考实验 2.3，实现物体的变换，允许用户通过键盘或者鼠标实现场景中至少两个物体的控制（移动，旋转，缩放等等）。

三、具体内容实现

1、场景设计和显示

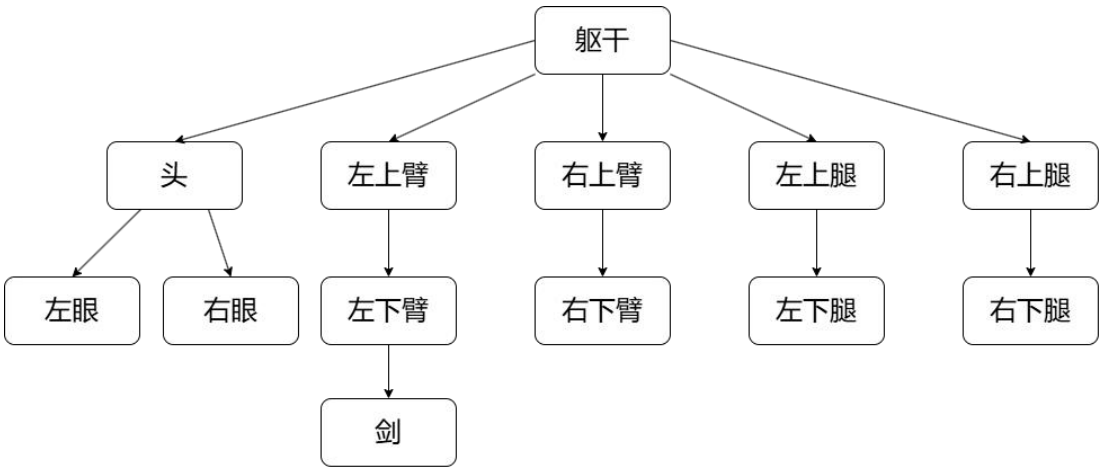


最终场景效果如上图所示。这是一个“隐士修所”的虚拟场景，描述了在一个宁静的夜晚，皮卡丘来到隐士居者的家里做客，二人在小木屋外喝茶、谈笑风声的故事。在夜晚的路灯下，对影成三人，显得隐士修所格外的幽静。

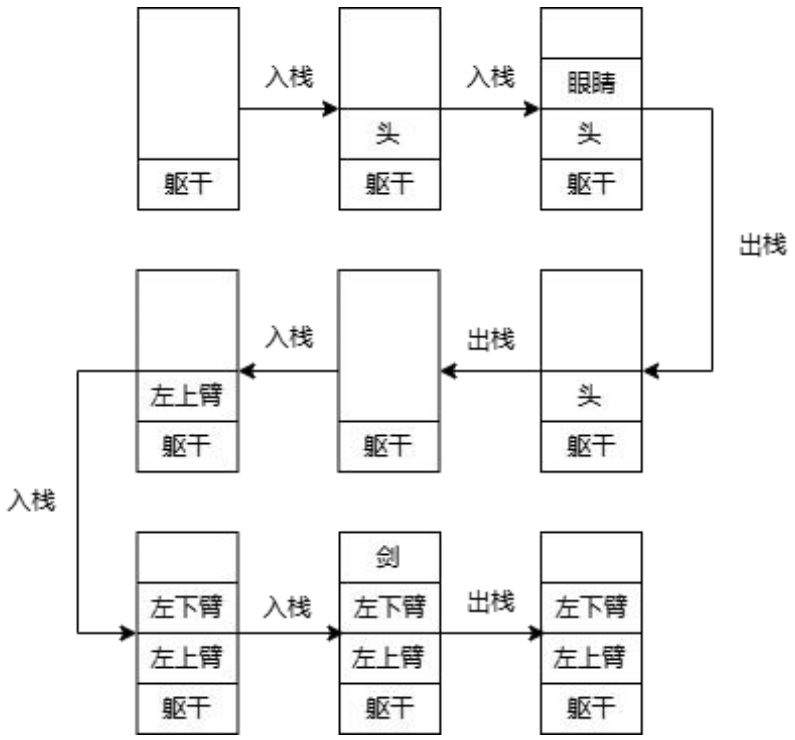
此场景包含的虚拟物体有：地板、围栏、路灯、机器人、皮卡丘、桌子、茶壶、茶杯、地球支架、地球、木屋。其中，运用了物体绘制，层级建模，纹理贴图，光照效果，阴影效果等多种技术。除此之外还有一系列打印出的详细的操作指南，可以通过键盘进行多种视角切换，物体控制和交互等。

具体而言，可通过键盘 T 键对机器人进行第一人称控制，键盘 M 键对皮卡丘进行第三人称控制；鼠标左键可以开启地球的自转，右键可以关闭地球的自传；此外，通过鼠标滚轮的上下滚动，可以调整相机视角的远近（与键盘 o/O 键相似）。其他的交互操作详情见配套的使用说明书文档。

根据实验要求需要进行层级建模，这里主要利用层级树的方法进行遍历绘制多个长方体，最终拼成一个机器人。按深度优先顺序，即 “躯干 -> 头 -> 左上臂 -> 左小臂 -> 剑 -> 右上臂 -> 右下臂 -> 左上腿 -> 左下腿 -> 右上腿 -> 右下腿”的顺序完成层级树的遍历，完成 display（）函数。其中躯干->左上臂->左下臂->剑这一局部达到了四层的最深层次。



在具体实现时，也可以理解为使用堆栈的方式来保存和恢复节点变换矩阵，如下图所示：



具体实现：

（1）实现层级建模机器人（四层）

首先，分别实现机器人各部位的绘制函数，包括躯干、头部、左眼、右眼、左上臂、左下臂、剑、右上臂、右下臂、左上腿、左下腿、右上腿、右下腿等等。

此处以躯干为例进行说明，其他身体组成部分的绘制函数与躯干的相似。我们先看躯干的绘制函数 void torso(glm::mat4 modelMatrix)：

```
// 躯体
void torso(glm::mat4 modelMatrix)
{
    // 本节点局部变换矩阵
    glm::mat4 instance = glm::mat4(1.0);
    instance = glm::translate(instance, glm::vec3(0.0, 0.5 * robot.TORSO_HEIGHT, 0.0));
    instance = glm::scale(instance, glm::vec3(robot.TORSO_WIDTH, robot.TORSO_HEIGHT, robot.TORSO_WIDTH));

    // 乘以来自父物体的模型变换矩阵，绘制当前物体
    drawMesh(modelMatrix * instance, Torso, TorsoObject);
}
```

在上述局部变换的构造过程中，instance 初始化为单位矩阵（glm::mat4(1.0)），然后通过一系列操作调整物体的姿态和形状。glm::translate 将模型沿 Y 轴方向平移，以确保躯干的位置正确位于局部坐标的中心偏上部，体现了对局部坐标系原点的合理布局。而 glm::scale 则依据机器人躯干的宽度和高度对物体进行非均匀缩放，直接控制几何体的最终尺寸。

将局部变换矩阵 instance 与全局变换矩阵 modelMatrix 相乘时，遵循了矩阵乘法的规则：全局变换会先应用，而局部变换则在其基础上进一步细化。这种方式保证了躯干的变换可以继承父节点的旋转、缩放和平移，同时还可以叠加其自身的变换。

最后，drawMesh 函数调用是实际的绘制步骤，modelMatrix * instance 提供了完整的变换矩阵，确保了模型正确地定位到世界坐标系中的目标位置，而 Torso 和 TorsoObject 则是用于定义模型几何体和相关资源的参数。

如果，我们想要调整各部位的位置以及大小，我们可以通过调整各部位对应的绘制函数中的 glm::translate 以及 glm::scale。这里我们定义了一个结构体 Robot robot 来方便我们对其进行管理，于是，我们只需要修改结构体中的参数即可调整位置、大小，如下图所示：

```
struct Robot
{
    // 关节大小
    float TORSO_HEIGHT = 4.0 * 0.1;
    float TORSO_WIDTH = 2.5 * 0.1;
    float UPPER_ARM_HEIGHT = 2.5 * 0.1;
    float LOWER_ARM_HEIGHT = 1.8 * 0.1;
    float UPPER_ARM_WIDTH =  0.8 * 0.1;
    float LOWER_ARM_WIDTH =  0.5 * 0.1;
    float UPPER_LEG_HEIGHT = 2.8 * 0.1;
    float LOWER_LEG_HEIGHT = 2.2 * 0.1;
    float UPPER_LEG_WIDTH =  1.0 * 0.1;
    float LOWER_LEG_WIDTH =  0.5 * 0.1;
    float HEAD_HEIGHT = 1.8 * 0.1;
    float HEAD_WIDTH = 1.5 * 0.1;
    float EYE_HEIGHT = 0.4 * 0.1;
    float EYE_WIDTH = 0.2 * 0.1;
```

然后，我们需要在 init 函数中调用 generateCube（）方法生成各个关节部分的模型，并设置对应的颜色，在将顶点数据传递到着色器中，准备下一步的绘制操作。

这里的颜色为一个 rgb 值，在 opengl 中为一个 3 维变量，第一维度代表 red，第二维度代表 green，第三维度代表 blue。

```
// 设置机器人颜色
Torso->generateCube(Blue);
Head->generateCube(Red);
RightUpperArm->generateCube(Yellow);
LeftUpperArm->generateCube(Yellow);
RightUpperLeg->generateCube(Brown);
LeftUpperLeg->generateCube(Brown);
RightLowerArm->generateCube(Green);
LeftLowerArm->generateCube(Green);
RightLowerLeg->generateCube(Sliver);
LeftLowerLeg->generateCube(Sliver);
left_sword->generateCube(DarkSliver);
LeftEye->generateCube(Black);
RightEye->generateCube(Black);


// 将物体的顶点数据传递
bindObjectAndData(Torso, TorsoObject, vshader, fshader);
bindObjectAndData(Head, HeadObject, vshader, fshader);
bindObjectAndData(RightUpperArm, RightUpperArmObject, vshader, fshader);
bindObjectAndData(LeftUpperArm, LeftUpperArmObject, vshader, fshader);
bindObjectAndData(RightUpperLeg, RightUpperLegObject, vshader, fshader);
bindObjectAndData(LeftUpperLeg, LeftUpperLegObject, vshader, fshader);
bindObjectAndData(RightLowerArm, RightLowerArmObject, vshader, fshader);
bindObjectAndData(LeftLowerArm, LeftLowerArmObject, vshader, fshader);
bindObjectAndData(RightLowerLeg, RightLowerLegObject, vshader, fshader);
bindObjectAndData(LeftLowerLeg, LeftLowerLegObject, vshader, fshader);
bindObjectAndData(left_sword, left_sword_Object, vshader, fshader);
bindObjectAndData(LeftEye, LeftEyeObject, vshader, fshader);
bindObjectAndData(RightEye, RightEyeObject, vshader, fshader);
```

最后，在我们定义完了所有组成部分的绘制函数之后，我们便可以在 display 函数中，根据树的思想，利用堆栈对机器人的各个组成部分进行绘制。

堆栈的过程需要严格按照树形结构来，不然会导致机器人的层级结构与我们的预期不同，首先绘制第一层的物体，即躯干，然后再按照 dfs 的方式绘制每一颗子树。

```
// 物体的变换矩阵
glm::mat4 modelMatrix;
modelMatrix = glm::mat4(1.0);
// 保持变换矩阵的栈
MatrixStack mstack;

// 躯干（这里我们希望机器人的躯干只绕Y轴旋转，所以只计算了RotateY）
modelMatrix = glm::translate(modelMatrix, robotPos);
modelMatrix = glm::rotate(modelMatrix, glm::radians(robot.theta[robot.Torso]), glm::vec3(0.0, 1.0, 0.0));
torso(modelMatrix);

mstack.push(modelMatrix); // 保存躯干变换矩阵
// 头部（这里我们希望机器人的头部只绕Y轴旋转，所以只计算了RotateY）
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, robot.TORSO_HEIGHT, 0.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(robot.theta[robot.Head]), glm::vec3(0.0, 1.0, 0.0));
head(modelMatrix);
// 左眼
glm::mat4 temp = modelMatrix;
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.03, robot.EYE_HEIGHT+0.03, -0.5 * robot.HEAD_WIDTH));
modelMatrix = glm::rotate(modelMatrix, glm::radians(robot.theta[robot.left_eye]), glm::vec3(0.0, 1.0, 0.0));
left_eye(modelMatrix);
// 右眼
modelMatrix = temp;
modelMatrix = glm::translate(modelMatrix, glm::vec3(-0.03, robot.EYE_HEIGHT + 0.03, -0.5 * robot.HEAD_WIDTH));
modelMatrix = glm::rotate(modelMatrix, glm::radians(robot.theta[robot.right_eye]), glm::vec3(0.0, 1.0, 0.0));
right_eye(modelMatrix);
modelMatrix = mstack.pop(); // 恢复躯干变换矩阵

// ========== 左臂 ==========
mstack.push(modelMatrix);   // 保存躯干变换矩阵
```

入栈

出栈

在此过程中，如果我们需要对机器人进行旋转，只需要调整躯干的旋转角度即可，因为剩余部分都会被躯干的变换矩阵所影响到。具体来说，我们可以修改 glm::rotate 函数的第二个参数，第三个参数指定旋转轴，这里为 y 轴。

由于我们将各部分的旋转角度封装在了结构体 robot 中，于是我们只需要去修改 robot 里的 theta 数组的初始值：

```
// 关节角大小
GLfloat theta[13] = {
    -90.0,    // Torso
    0.0,    // Head
    0.0,    // RightUpperArm
    0.0,    // RightLowerArm
    0.0,    // LeftUpperArm
    0.0,    // LeftLowerArm
    0.0,    // RightUpperLeg
    0.0,    // RightLowerLeg
    0.0,    // LeftUpperLeg
    0.0,     // LeftLowerLeg
    30.0,     // left_sword
    0.0,     // left_eye
    0.0      // right_eye
};
```

（2）绘制地面

实验还要求应包含地面，起初我们采用了 generateSquare 函数的方法生成一个正方形图像，直接绘制作为地面，值得注意的是为了使后面物体的阴影能展现出来，所以设置正方形位置的时候在 y 故意设为-0.01 使其低于阴影平面。

```
//// 创建正方形平面，给它一个其他颜色
//plane->generateSquare(glm::vec3(0.6, 0.8, 0.0));
//// 设置正方形的位置和旋转，注意这里我们将正方形平面下移了一点点距离，
//// 这是为了防止和阴影三角形重叠在同个平面上导致颜色交叉
//plane->setRotation(glm::vec3(90, 0, 0));
//plane->setTranslation(glm::vec3(0, -0.01, 0));
//plane->setScale(glm::vec3(4, 4, 4));
//bindObjectAndData(plane, plane_object, vshader, fshader);
```

但在后来，我们发现这种方案的效果并不佳，于是决定使用带有纹理的地板模型作为地面，从而增加场景的美观度。我们在网上检索了不少 obj 模型及其对应的纹理贴图，并保存到本地。

首先，我们创建了一个地板对象： `TriMesh* myFloor = new TriMesh();  // 地板`

然后，我们使用 TriMesh 类的相关函数去读取地板模型，并为模型添加纹理图片，并加入到 painter 对象中，以便后续对纹理模型进行统一绘制：

```
// 读取地板模型
myFloor->setNormalize(true);
myFloor->readObj("./assets/floor/floor.obj");
// 设置物体的旋转位移
myFloor->setTranslation(glm::vec3(0.0, -0.01, 0));
myFloor->setRotation(glm::vec3(0.0, 0.0, 0.0));
myFloor->setScale(glm::vec3(25.0, 1.0, 25.0));
// 加到painter中
painter->addMesh(myFloor, "mesh_m", "./assets/floor/floor.jpg", vshader, fshader);  // 指定纹理与着色器
// 我们创建的这个加入一个容器内，为了程序结束时将这些数据释放
meshList.push_back(myFloor);
```

此外，我们还需要对地板的位置、大小进行参数调整，以使其出现在恰当的位置。

（3）绘制其他的虚拟物体

为了构建出一个“隐士修所”主题的室外虚拟场景，我们还需要绘制更多具有时代风格的虚拟物体，如：木桌、木屋、路灯、茶壶、茶杯等等。其中，纹理模型的绘制方法几乎一致，与地板的

绘制过程相似。

在此，我们仅以桌子这一个模型来举例说明。

首先，定义一个 TriMesh 对象来保存 table 的信息： `TriMesh* table = new TriMesh(); // 桌子。`

然后，读取桌子的 obj 模型，并添加纹理图片，接着加入到 painter 中，等待统一绘制：

```
// @TODO: Task2 读取桌子模型
table->setNormalize(true);
table->readObj("./assets/bigTable.obj");
// 设置物体的旋转位移
table->setTranslation(glm::vec3(0.0, 0.15, 1.0));
table->setRotation(glm::vec3(0.0, 0.0, 0.0));
table->setScale(glm::vec3(1.2, 2.0, 2.5));
// 加到painter中
painter->addMesh(table, "mesh_a", "./assets/bigTable.bmp", vshader, fshader);   // 指定纹理与着色器
// 我们创建的这个加入一个容器内，为了程序结束时将这些数据释放
meshList.push_back(table);
```

最后，我们加入到 painter 中的一系列模型，都会在 display 函数中被绘制出来，这归功于 painter->drawMeshes 方法：

```
// 绘制一系列带纹理物体
painter->drawMeshes(light, camera);
```

2、添加纹理

（1）读取带纹理的 obj 文件

为物体添加纹理，主要在于 readObj 函数。这部分是读取并存储包括顶点坐标 vertex_positions、顶点法线数据 vertex_normals、顶点纹理数据或者说 UV 坐标 vertex_textures，先分别读取数据再 push_back 压入这些向量中。

```
if (type == "v")
{
    sin >> _x >> _y >> _z;
    vertex_positions.push_back(glm::vec3(_x, _y, _z));
}
if (type == "vn")
{
    sin >> _x >> _y >> _z;
    vertex_normals.push_back(glm::vec3(_x, _y, _z));
    //vertex_colors.push_back(glm::vec3(_x, _y, _z));
}
if (type == "vt")
{
    sin >> _x >> _y >> _z;
    vertex_textures.push_back(glm::vec2(_x, _y));
}
```

下面这部分负责读取并存储以“f”开头的坐标信息，包括片面的顶点索引数据 faces、三角面片顶点的法线索引下标 normal_index 以及三角面片顶点的纹理坐标索引下标。其中值得注意的是在 obj 文件中索引下标是从 1 开始的，所以需要在传入的时候减 1。

```
if (type == "f")
{
    sin >> a0 >> slash >> b0 >> slash >> c0;
    sin >> a1 >> slash >> b1 >> slash >> c1;
    sin >> a2 >> slash >> b2 >> slash >> c2;
    //sin >> a3 >> slash >> b3 >> slash >> c3;
    faces.push_back(vec3i(a0 - 1, a1 - 1, a2 - 1));
    texture_index.push_back(vec3i(b0 - 1, b1 - 1, b2 - 1));
    normal_index.push_back(vec3i(c0 - 1, c1 - 1, c2 - 1));
}
```

在读取完 obj 内的数据后，用法向量的值赋予 vertex_color 和 color_index 并调用 storeFacesPoints 函数存储数据。

```
// 其中vertex_color和color_index可以用法向量的数值赋值
vertex_colors = vertex_normals;
color_index = normal_index;
storeFacesPoints();
```

（2）完善数据的读取

在 storeFacesPoints 函数，主要需要完成归一化和四个信息的存储。

```
void TriMesh::storeFacesPoints() {
    norm();
    // 计算法向量
    if (vertex_normals.size() == 0)
        computeVertexNormals();
    // 根据每个三角面片的顶点下标存储要传入GPU的数据
    for (int i = 0; i < faces.size(); i++)
    {
        // 坐标
        points.push_back(vertex_positions[faces[i].x]);
        points.push_back(vertex_positions[faces[i].y]);
        points.push_back(vertex_positions[faces[i].z]);
        // 颜色
        colors.push_back(vertex_colors[color_index[i].x]);
        colors.push_back(vertex_colors[color_index[i].y]);
        colors.push_back(vertex_colors[color_index[i].z]);
        // 法向量
        if (vertex_normals.size() != 0)
        {
            normals.push_back(vertex_normals[normal_index[i].x]);
            normals.push_back(vertex_normals[normal_index[i].y]);
            normals.push_back(vertex_normals[normal_index[i].z]);
        }
        // 纹理
        if (vertex_textures.size() != 0)
        {
            textures.push_back(vertex_textures[texture_index[i].x]);
            textures.push_back(vertex_textures[texture_index[i].y]);
            textures.push_back(vertex_textures[texture_index[i].z]);
        }
    }
}
```

这里自定义了 norm 函数用以归一化，而四个信息的存储或者准确说将三角面片的顶点索引传入 GPU 中，分别是坐标 points、颜色 colors、法向量 normals、纹理 textures，由于前面已经将这些信息存储到 vertex 中，所以这里只需要用 for 循环遍历，按角标索引以及各个点的信息逐个压入即可。

```
void TriMesh:: norm() {
    if (do_normalize_size) {
        // 记录物体包围盒大小，可以用于大小的归一化
        // 先获得包围盒的对角顶点
        float max_x = -FLT_MAX;
        float max_y = -FLT_MAX;
        float max_z = -FLT_MAX;
        float min_x = FLT_MAX;
        float min_y = FLT_MAX;
        float min_z = FLT_MAX;
        for (int i = 0; i < vertex_positions.size(); i++) {
            auto& position = vertex_positions[i];
            if (position.x > max_x) max_x = position.x;
            if (position.y > max_y) max_y = position.y;
            if (position.z > max_z) max_z = position.z;
            if (position.x < min_x) min_x = position.x;
            if (position.y < min_y) min_y = position.y;
            if (position.z < min_z) min_z = position.z;
        }
        up_corner = glm::vec3(max_x, max_y, max_z);
        down_corner = glm::vec3(min_x, min_y, min_z);
        center = glm::vec3((min_x + max_x) / 2.0, (min_y + max_y) / 2.0, (min_z + max_z) / 2.0);

        diagonal_length = length(up_corner - down_corner);

        for (int i = 0; i < vertex_positions.size(); i++) {
            vertex_positions[i] = (vertex_positions[i] - center) / diagonal_length;
        }
    }
}
```

（3）绘制纹理物体

这里主要以 table 为例展现具体绘制含纹理物体的过程，先确认归一化，然后调用 readObj 函数读取 bigTable.obj 文件，设置物体的旋转位移，最后加到选择需要添加的纹理再加入到 painter 中准备绘制。

```
table->setNormalize(true);
table->readObj("./assets/bigTable.obj");
// 设置物体的旋转位移
table->setTranslation(glm::vec3(0.0, 0.15, 1.0));
table->setRotation(glm::vec3(0.0, 0.0, 0.0));
table->setScale(glm::vec3(1.2, 2.0, 2.5));
// 加到painter中
painter->addMesh(table, "mesh_a", "./assets/bigTable.bmp", vshader, fshader);   // 指定纹理与着色器
// 我们创建的这个加入一个容器内，为了程序结束时将这些数据释放
meshList.push_back(table);
```

加入到 painter 后，调用 drawMesh 函数进行绘制。其中将 isShadow 设置为 2，表示根据纹理绘制。

```
void MeshPainter::drawMesh(TriMesh* mesh, openGLObject& object, Light* light, Camera* camera, bool hasShadow) {

    // 相机矩阵计算
    camera->updateCamera();
    camera->viewMatrix = camera->getViewMatrix();
    camera->projMatrix = camera->getProjectionMatrix(false); // 采用透视投影（false）

#ifdef __APPLE__    // for MacOS
    glBindVertexArrayAPPLE(object.vao);
#else
    glBindVertexArray(object.vao);
#endif
    glUseProgram(object.program);

    // 物体的变换矩阵
    glm::mat4 modelMatrix = mesh->getModelMatrix();
    // 传递矩阵
    glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniformMatrix4fv(object.viewLocation, 1, GL_TRUE, &camera->viewMatrix[0][0]);
    glUniformMatrix4fv(object.projectionLocation, 1, GL_TRUE, &camera->projMatrix[0][0]);
    // 将着色器 isShadow 设置为2，表示根据纹理绘制
    glUniform1i(object.shadowLocation, 2)
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, object.texture);// 该语句必须，否则将只使用同一个纹理进行绘制
    // 传递纹理数据 将生成的纹理传给shader
    glUniform1i(glGetUniformLocation(object.program, "texture"), 0);
    // 将材质和光源数据传递给着色器
    bindLightAndMaterial(mesh, object, light, camera);
    // 绘制
    glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
```

除此之外，由于还需要根据光照绘制阴影，所以在 drawMesh 函数后半部分添加了绘制阴影的方法， 具体实现将放到第三个实验要求的部分详细说明，这里暂时截图展示一下。

```
// @TODO: 根据光源位置，计算阴影投影矩阵
glm::vec3 light_position = light->getTranslation();
float lx = light_position[0];
float ly = light_position[1];
float lz = light_position[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);
// 计算阴影的模型变换矩阵。
modelMatrix = shadowProjMatrix * modelMatrix;
// 传递 isShadow 变量，3表示黑色。
glUniform1i(object.shadowLocation, 3);
// 传递 uniform 关键字的矩阵数据。
glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
// 绘制
if(hasShadow)
    glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
```

最终实现的效果如下图所示：（桌上的物体为后续添加的效果）

3、添加光照、材质、阴影效果

（1）设置光照效果

在 init 函数中，即初始化物体之前，设置光源的位置以及各类系数：

```
// 设置光源位置
light->setTranslation(glm::vec3(-5.0, 15.0, 10.0));
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
light->setAttenuation(1.0, 0.045, 0.0075); // 衰减系数
```

（2）添加材质效果

在 mesh_init 函数中，设置 mesh 物体的旋转、位移、材质并传递顶点数据：

```
void mesh_init() {
    /*
    feat: 3.添加材质效果
    */
    std::string vshader, fshader;
    // 读取着色器并使用
    vshader = "shaders/vshader.glsl";
    fshader = "shaders/fshader.glsl";
    // 设置物体的旋转位移
    mesh->setTranslation(glm::vec3(0.9, 0.34, 1.0));
    mesh->setRotation(glm::vec3(0.0, -90.0, 0.0));
    mesh->setScale(glm::vec3(1.0, 1.0, 1.0));
    // 设置材质
    mesh->setAmbient(glm::vec4(0.24752f, 0.1995f, 0.0745f, 1.0)); // 环境光
    mesh->setDiffuse(glm::vec4(0.75164f, 0.60648f, 0.22648f, 1.0)); // 漫反射
    mesh->setSpecular(glm::vec4(0.62828f, 0.555802f, 0.366065f, 1.0)); // 镜面反射
    mesh->setShininess(1.0); //高光系数
    // 将物体的顶点数据传递
    bindObjectAndData(mesh, mesh_object, vshader, fshader);
}
```

调用 draw_obj 函数绘制图像，将 isShadow 设置为 0 表示根据光照绘制，并传递光照和材质数据：

```
void draw_obj(TriMesh* mesh, openGLObject&mesh_object, int a) {
    glm::mat4 modelMatrix;
    // 绘制光照物体
    glBindVertexArray(mesh_object.vao);
    glUseProgram(mesh_object.program);
    modelMatrix = mesh->getModelMatrix();
    glUniformMatrix4fv(mesh_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniformMatrix4fv(mesh_object.viewLocation, 1, GL_TRUE, &camera->viewMatrix[0][0]);
    glUniformMatrix4fv(mesh_object.projectionLocation, 1, GL_TRUE, &camera->projMatrix[0][0]);
    // 将isShadow设置为a，根据传入的参数绘制
    glUniform1i(mesh_object.shadowLocation, a);
    bindLightAndMaterial(mesh, mesh_object, light, camera);
    glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
}
```

（3）添加阴影效果

调用 draw_sha 函数绘制阴影，确定投影矩阵，将 isShadow 设置为 3 表示黑色（阴影）：

```
void draw_sha(TriMesh* m, openGLObject& mesh_object, int sha) {
    // 根据光源位置，计算阴影投影矩阵
    light_position = light->getTranslation();
    float lx = light_position[0];
    float ly = light_position[1];
    float lz = light_position[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);
    // 计算阴影的模型变换矩阵。
    glm::mat4 modelMatrix = m->getModelMatrix();
    modelMatrix = shadowProjMatrix * modelMatrix;
    // 传递 isShadow 变量，3表示黑色。
    glUniform1i(mesh_object.shadowLocation, sha);
    // 传递 uniform 关键字的矩阵数据。
    glUniformMatrix4fv(mesh_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    // 绘制
    glDrawArrays(GL_TRIANGLES, 0, m->getPoints().size());
}
```

最后在 display 函数中调用上述两个函数进行绘制：

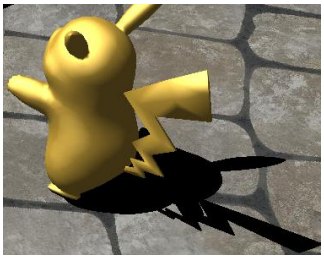
```
// 绘制光照模型及阴影
draw_obj(mesh, mesh_object, 0);
draw_sha(mesh, mesh_object, 3);
```

这里对根据光照绘制物体的 fshader 着色器进行一下补充说明，根据 phong 光照模型主要有三个分量，分别为环境光分量 I_a，漫反射分量 I_d，镜面反射分量 I_s，将这三个光照分量加在一起得到 fColor = I_a + I_d + I_s 得到根据光照效果的绘制。

```
// @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 = 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 = 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;
```

（4）实现效果

在添加了光照、材质、阴影后，我的皮卡丘便有了立体感，具体效果如下图所示：



4、用户交互实现视角切换完成对场景的任意角度浏览

这里主要涉及到 Camera.cpp 中 lookat 函数的编写。在该函数中分别定义 n，u，n 等向量得到 viewMatrix 矩阵；由于在最开始，我们还需要将相机从坐标原点移动到视点，所以还需要一个平移矩阵并添加平移矩阵 T。

参考公式：

（1）观察平面法向量（VPN：View-plane Normal）

$$VPN = e - a$$

（2）归一化得到：

$$\boldsymbol{n} = \frac{VPN}{|VPN|}$$

（3）与 VUP 和 VPN 都垂直的方向向量

$$\boldsymbol{u} = \frac{VUP \times \boldsymbol{n}}{|VUP \times \boldsymbol{n}|}$$

（4）VUP 在照相机胶片平面上的投影

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

（5）相机观察矩阵 viewMatrix

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

首先，定义好一个 lookAt 函数，用于生成观察矩阵：

```
glm::mat4 Camera::lookAt(const glm::vec4& eye, const glm::vec4& at, const glm::vec4& up)
{
    // @TODO: Task1:请按照实验课内容补全相机观察矩阵的计算
    // 获得相机方向。
    glm::vec4 n = glm::normalize(eye - at);
    // 获得右(x)轴方向。
    glm::vec3 up_3 = up;
    glm::vec3 n_3 = n;
    glm::vec4 u = glm::normalize(glm::vec4(glm::cross(up_3, n_3), 0.0));
    // 获得上(y)轴方向。
    glm::vec3 u_3 = u;
    glm::vec4 v = glm::normalize(glm::vec4(glm::cross(n_3, u_3), 0.0));

    glm::vec4 t = glm::vec4(0.0, 0.0, 0.0, 1.0);
    glm::mat4 c = glm::mat4(u, v, n, t);

    // 处理相机位置向量。
    glm::mat4 p = glm::mat4(1.0f);
    p[0].w = -(eye.x);
    p[1].w = -(eye.y);
    p[2].w = -(eye.z);

    glm::mat4 view = p * c;
    return view;                    // 计算最后需要沿-eye方向平移
}
```

然后，在 Camera::updateCamera()中设置相机位置和方向：

```
// 计算相机的位置
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);
```

接着，在 display 函数中调用 get 函数计算得到相机矩阵 viewMatrix 和 projMatrix。其中在 getProjectionMatrix 函数中传入 true 变量表示采用正交投影：

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 相机矩阵计算
    camera->updateCamera();
    camera->viewMatrix = camera->getViewMatrix();
    camera->projMatrix = camera->getProjectionMatrix(true);
```

最后，我们将利用键盘回调函数实现视角的切换：

在实现任意角度变化前需要先了解相机中的各个参数及实际意义：

```
// 相机位置参数
float radius;       //前后远近变动视角角度
float rotateAngle;  //环绕中心左右转动视角角度
float upAngle;      //环绕中心上下转动视角角度
```

在了解各个参数的意义之后，在 keyboard 函数中设置通过按键改变相机参数，实现从任意角度浏览：

```
void Camera::keyboard(int key, int action, int mode)
{
    // 键盘事件处理
    // 通过按键改变相机和投影的参数
    if (key == GLFW_KEY_U && mode == 0x0000) {
        rotateAngle += 5.0;
        if (rotateAngle > 180)
            rotateAngle = rotateAngle - 360;
    }
    else if (key == GLFW_KEY_U && mode == GLFW_MOD_SHIFT) {
        rotateAngle -= 5.0;
        if (rotateAngle < -180)
            rotateAngle = rotateAngle + 360;
    }
    else if (key == GLFW_KEY_I && mode == 0x0000) {
        upAngle += 5.0;
        if (upAngle >= 180)
            upAngle = upAngle - 360;
    }
    else if (key == GLFW_KEY_I && mode == GLFW_MOD_SHIFT) {
        upAngle -= 5.0;
        if (upAngle <= -180)
            upAngle = upAngle + 360;
    }
    else if (key == GLFW_KEY_O && mode == 0x0000) {
        radius += 0.1;
    }
    else if (key == GLFW_KEY_O && mode == GLFW_MOD_SHIFT) {
        radius -= 0.1;
    }
    // 空格键初始化所有参数
    else if (key == GLFW_KEY_SPACE && mode == 0x0000) {
        initCamera();
    }

}
```

此外，我们还定义了一个鼠标滚轮的回调事件，从而实现滚轮调整相机视角的大小：

```
// 通过鼠标滚轮上下改变相机的距离大小
void Camera::scroll(double xoffset, double yoffset)
{
    radius += yoffset * 0.1;
}
```

具体操作说明：

- u / U 分别控制视角的逆时针/顺时针旋转
- i / I 分别控制视角的上/下移动
- o / 0 分别控制视角的拉远/近
- 鼠标滚轮向上/向下分别控制视角的拉远/近

效果展示：

5、通过交互控制物体

这部分主要涉及到 main.cpp 主函数中键盘回调 key_callback、鼠标点击回调 mouse_button_callback 以及鼠标滚轮回调 mouse_scroll_callback 三个回调函数的编写。实现了若干功能，包括选择需要控制的物体、自由移动、改变光照位置、视角转动、视角缩放、旋转平移、材质物体切换等。

以下将进行详细说明：

（1）通过按键 0~9 选择需要控制的机器人的哪个关节部分。

```
case GLFW_KEY_ESCAPE: exit(EXIT_SUCCESS); break;
case GLFW_KEY_Q: exit(EXIT_SUCCESS); break;
case GLFW_KEY_GRAVE_ACCENT: Selected_mesh = robot.left_sword; break;
case GLFW_KEY_1: Selected_mesh = robot.Torso; break;
case GLFW_KEY_2: Selected_mesh = robot.Head; break;
case GLFW_KEY_3: Selected_mesh = robot.RightUpperArm; break;
case GLFW_KEY_4: Selected_mesh = robot.RightLowerArm; break;
case GLFW_KEY_5: Selected_mesh = robot.LeftUpperArm; break;
case GLFW_KEY_6: Selected_mesh = robot.LeftLowerArm; break;
case GLFW_KEY_7: Selected_mesh = robot.RightUpperLeg; break;
case GLFW_KEY_8: Selected_mesh = robot.RightLowerLeg; break;
case GLFW_KEY_9: Selected_mesh = robot.LeftUpperLeg; break;
case GLFW_KEY_0: Selected_mesh = robot.LeftLowerLeg; break;
```

（2）通过按键 a / s 控制机器人逆/顺时针旋转。

```
// 通过按键旋转机器人
case GLFW_KEY_A:
    robot.theta[Selected_mesh] += 5.0;
    if (robot.theta[Selected_mesh] > 360.0)
        robot.theta[Selected_mesh] -= 360.0;
    break;
case GLFW_KEY_S:
    robot.theta[Selected_mesh] -= 5.0;
    if (robot.theta[Selected_mesh] < 0.0)
        robot.theta[Selected_mesh] += 360.0;
    break;
```

（3）通过按键 V、B、N 切换具有光照、材质、阴影效果的物体。

```
// 切换具有光照、材质、阴影效果的物体
case GLFW_KEY_V:
    mesh = new TriMesh();
    mesh->readOff("./assets/sphere.off");
    mesh_init();
    break;
case GLFW_KEY_B:
    mesh = new TriMesh();
    mesh->readOff("./assets/cow.off");
    mesh_init();
    break;
case GLFW_KEY_N:
    mesh = new TriMesh();
    mesh->readOff("./assets/Squirtle.off");
    mesh_init();
    break;
```

（4）选择需要控制的物体。

```
// 通过按钮 T 和 M 来分别选择机器人（第一人称）或物体（第三人称）的控制权
case GLFW_KEY_T:
    isRobot = true; // 控制机器人
    obj = Torso;
    /*
    feat：5.通过交互控制物体
    此处实现了第一人称和第三人称的切换。第一人称隐藏鼠标，第三人称显示鼠标。
    */
    if (camera->getviewmodel() == 1)
    {
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        camera->setviewmodel(3);
    }
    else
    {
        camera->setviewmodel(1);
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);//第一人称隐藏鼠标
    }
    break;
case GLFW_KEY_M:
    isRobot = false;// 不控制机器人
    obj = mesh;
    break;
```

在此处，第一次使用按键 T 时，将使用第一人称对机器人进行操控，重复使用按键 T 时，将切换回第三人称视角。当使用按键 M 时，将使用第三人称视角对材质物体进行操控。

这里补充一下第一人称视角的实现方法：

首先，需要在 Camera 类中补充一些重要的成员变量及方法：

```
//人称视角参数与函数
int viewmodel = 3;
void setviewmodel(int model);   //设置视角模式
int getviewmodel();             //获取视角模式
glm::mat4 permatrix;            //人称视角矩阵
glm::mat4 getpermatrix();       //获取人称视角矩阵
void setmatrix(glm::mat4 matrix);   //设置人称视角矩阵
glm::vec3 direction;            //人称视角方向
```

然后，在 main 方法中，在机器人头部的绘制函数中利用头部的模型矩阵生成相机的人称视角矩阵，因为第一人称视角是根据机器人的视角来实现的：

```
// 头部
void head(glm::mat4 modelMatrix)
{
    // 本节点局部变换矩阵
    glm::mat4 instance = glm::mat4(1.0);
    instance = glm::translate(instance, glm::vec3(0.0, 0.5 * robot.HEAD_HEIGHT, 0.0));
    instance = glm::scale(instance, glm::vec3(robot.HEAD_WIDTH, robot.HEAD_HEIGHT, robot.HEAD_WIDTH));

    camera->setmatrix(modelMatrix); // 设置人称视角矩阵

    // 乘以来自父物体的模型变换矩阵，绘制当前物体
    drawMesh(modelMatrix * instance, Head, HeadObject);
}
```

接着，初始化一个相机视角方向的矢量变量： `glm::vec3 cameraDirection(1, 0, 0);//相机视线方向` ，并且在机器人旋转的时候要对其进行修改：

```
    // 控制物体的旋转
case GLFW_KEY_R:
    if (isRobot) {
        if (int(robot.theta[0]) % 90 == 0)
            robot.theta[0] -= 90;
        else
            robot.theta[0] -= int(robot.theta[0]) % 90 + 90;
        if (cameraDirection.x == 1)
            cameraDirection = glm::vec3(0, 0, 1);
        else if (cameraDirection.z == 1) {
            cameraDirection = glm::vec3(-1, 0, 0);
        }
        else if (cameraDirection.x == -1)
            cameraDirection = glm::vec3(0, 0, -1);
        else if (cameraDirection.z == -1)
            cameraDirection = glm::vec3(1, 0, 0);
        break;
    }
    temp = obj->getRotation();
    if (mode == GLFW_MOD_SHIFT)
        temp.y += 90.0;
    else
        temp.y -= 90.0;
    obj->setRotation(temp);
    break;
```

此外，还要在使用按键 T 时切换相机的人称视角模式：

```
case GLFW_KEY_T:
    isRobot = true; // 控制机器人
    obj = Torso;
    /*
    feat: 5.通过交互控制物体
    此处实现了第一人称和第三人称的切换。第一人称隐藏鼠标，第三人称显示鼠标。
    */
    if (camera->getviewmodel() == 1)
    {
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        camera->setviewmodel(3);
    }
    else
    {
        camera->setviewmodel(1);
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);//第一人称隐藏鼠标
    }
    break;
```

最后，在 display 函数中，当人称模式为第一人称时，需要将相机的方向进行更新，更新为当前的 cameraDirection：

```
if (camera->getviewmodel() == 1)
{
    camera->direction = cameraDirection;
}
```

这样一来，在 `Camera::updateCamera()` 中，就可以根据人称视角矩阵 `permatrix`、相机方向 `direction` 来生成机器人第一人称的相应 eye、up、at 矢量，从而也就实现了相机视角的变换。

```
void Camera::updateCamera()
{
    // 使用相对于at的角度控制相机的时候，注意在upAngle大于90的时候，相机坐标系的u向量会变成相反的方向，
    // 要将up的y轴改为负方向才不会发生这种问题

    if (viewmodel != 3)
    {
        eye = glm::vec4(permatrix[3][0], permatrix[3][1] + 1.2, permatrix[3][2], 1.0);
        at = glm::vec4(permatrix[3][0] + direction.x, permatrix[3][1] + 0.4 + direction.y, permatrix[3][2] + direction.z, 1.0);
        up = glm::vec4(0.0, 1.0, 0.0, 0.0);
        return;
    }
    up = glm::vec4(0.0, 1.0, 0.0, 0.0);
    if (upAngle > 90) {
        up.y = -1;
    }
```

（5）控制物体旋转。

```
case GLFW_KEY_R:
    if (isRobot) {
        if (int(robot.theta[0]) % 90 == 0)
            robot.theta[0] -= 90;
        else
            robot.theta[0] -= int(robot.theta[0]) % 90 + 90;
        if (cameraDirection.x == 1)
            cameraDirection = glm::vec3(0, 0, 1);
        else if (cameraDirection.z == 1) {
            cameraDirection = glm::vec3(-1, 0, 0);
        }
        else if (cameraDirection.x == -1)
            cameraDirection = glm::vec3(0, 0, -1);
        else if (cameraDirection.z == -1)
            cameraDirection = glm::vec3(1, 0, 0);
        break;
    }
    temp = obj->getRotation();
    if (mode == GLFW_MOD_SHIFT)
        temp.y += 90.0;
    else
        temp.y -= 90.0;
    obj->setRotation(temp);
    break;
```

（6）控制物体在平面上任意移动。

```
case GLFW_KEY_LEFT:
    if (isRobot) {
        mapAngle == -3 || mapAngle == 5 ? robotPos[abs(mapAngle % 3)==0 ? 2 : 0]+= 0.2 : robotPos[abs(mapAngle % 3) == 0 ? 2 : 0] -= 0.2;
        break;
    }
    temp = obj->getTranslation();
    mapAngle == -3 || mapAngle == 5 ? temp[abs(mapAngle % 3) == 0 ? 2 : 0] += 0.2 : temp[abs(mapAngle % 3) == 0 ? 2 : 0] -= 0.2;
    obj->setTranslation(temp);
    break;
case GLFW_KEY_RIGHT:
    if (isRobot) {
        mapAngle == -3 || mapAngle == 5 ? robotPos[abs(mapAngle % 3) == 0 ? 2 : 0] -= 0.2 : robotPos[abs(mapAngle % 3) == 0 ? 2 : 0] += 0.2;
        break;
    }
    temp = obj->getTranslation();
    mapAngle == -3 || mapAngle == 5 ? temp[abs(mapAngle % 3) == 0 ? 2 : 0] -= 0.2 : temp[abs(mapAngle % 3) == 0 ? 2 : 0] += 0.2;
    obj->setTranslation(temp);
    break;
case GLFW_KEY_UP:
    if (isRobot) {
        mapAngle > 0 ? robotPos[ abs(mapAngle % 3) ] += 0.2 : robotPos[abs(mapAngle % 3)] -= 0.2;
        break;
    }
    temp = obj->getTranslation();
    mapAngle > 0 ? temp[abs(mapAngle % 3)] += 0.2 : temp[abs(mapAngle % 3)] -= 0.2;
    obj->setTranslation(temp);
    break;
case GLFW_KEY_DOWN:
    if (isRobot) {
        mapAngle > 0 ? robotPos[abs(mapAngle % 3)] -= 0.2 : robotPos[abs(mapAngle % 3)] += 0.2;
        break;
    }
    temp = obj->getTranslation();
    mapAngle > 0 ? temp[abs(mapAngle % 3)] -= 0.2 : temp[abs(mapAngle % 3)] += 0.2;
    obj->setTranslation(temp);
    break;
```

（7）控制光照位置变动。

使用 x / X 按键分别控制光源位置向移动 X 轴负/正方向移动：

```
case GLFW_KEY_X:    // 沿着X轴移动光源
    light_position = light->getTranslation();
    if (mode == GLFW_MOD_SHIFT)
        light_position[0] += move_step_size;
    else
        light_position[0] -= move_step_size;
    light->setTranslation(light_position);
    break;
```

使用 y / Y 分别控制光源位置向移动 Y 轴负/正方向移动：

```
case GLFW_KEY_Y:    // 沿着Y轴移动光源
    light_position = light->getTranslation();
    if (mode == GLFW_MOD_SHIFT)
        light_position[1] += move_step_size;
    else {
        light_position[1] -= move_step_size;
        if (light_position[1] <= 1.0) {
            light_position[1] += move_step_size;
        }
    }
    light->setTranslation(light_position);
    break;
```

使用 z / Z 分别控制光源位置向移动 Z 轴负/正方向移动：

```
case GLFW_KEY_Z:    // 沿着Z轴移动光源
    light_position = light->getTranslation();
    if (mode == GLFW_MOD_SHIFT)
        light_position[2] += move_step_size;
    else
        light_position[2] -= move_step_size;
    light->setTranslation(light_position);
    break;
```

总结而言，键盘交互功能可见控制台输出以及使用说明书：

```
[Part choose]
1:              Torso
2:              Head
3:              RightUpperArm
4:              RightLowerArm
5:              LeftUpperArm
6:              LeftLowerArm
7:              RightUpperLeg
8:              RightLowerLeg
9:              LeftUpperLeg
0:              LeftLowerLeg

[Robot action]
a/A:    Increase rotate angle
s/S:    Decrease rotate angle

[Model choose]
v/V:    read off 'sphere'
b/B:    read off 'cow'
n/N:    read off 'squirtle'

[Control]
t/T:    control 'robot'(the first sight)
m/M:    control 'mesh'(the third sight)
left:   move to the left
right:  move to the right
up:     move to the up
down:   move to the down
r/R:    rotate the current mesh

[Mouse]
Mouse_left_button:      rotate the earth
Mouse_right_button:     stop rotation

[Scroll]
scroll up:      Increase the camera radius
scroll down:    Decrease the camera radius

[Camera]
SPACE:          Reset camera parameters
u/U:            Increase/Decrease the rotate angle
i/I:            Increase/Decrease the up angle
o/O:            Increase/Decrease the camera radius

[Light]
x/X:            move the light along X positive/negative axis
y/Y:            move the light along Y positive/negative axis
z/Z:            move the light along Z positive/negative axis
```

四、实验总结与体会

实验结论：

在本次计算机图形学的期末大作业中，我成功构建了一个名为“隐士修所”的虚拟场景，并实现了用户交互、光照效果、阴影效果、纹理映射等任务。我不仅加深了对三维图形渲染的理解，还提升了解决实际问题的能力。

在实验过程中，我成功应用了层级建模技术构建了一个复杂四层次的机器人模型，并实现了其在场景中的绘制。这一过程让我深刻理解了三维模型的构建和变换原理，以及如何通过矩阵操作来实现物体的精确定位和变换。

此外，我还为场景中的主要物体添加了纹理贴图，增强了场景的真实感，同时也掌握了如何在 OpenGL 中应用纹理映射技术。光照、材质与阴影效果的实现是本次实验的另一个挑战。通过对光照模型、材质属性和阴影效果的深入研究和实践，我成功地为场景中的物体添加了逼真的光照效果，这让我对光照模型有了更深入的认识，并学会了如何在 OpenGL 中实现复杂的光照效果。

用户交互的实现也是本次实验的一个重要部分。我实现了用户通过键盘和鼠标与虚拟场景的交互，包括人称视角的切换以及物体的控制权切换。这一过程锻炼了我的编程能力，也让我理解了用户交互在图形学应用中的重要性。

总的来说，这次实验不仅让我巩固了计算机图形学的理论知识，还提升了解决实际问题的能力，为我未来的学习和研究打下了坚实的基础。通过实践，我对 OpenGL 的理解和应用能力得到了显著提升，同时也对计算机图形学中的模型构建、纹理映射、光照处理和用户交互有了更深刻的认识。

实验难点：

1. **层级建模的复杂性：**在构建机器人模型时，我遇到了层级建模的复杂性问题。需要精确控制每个部件的变换矩阵，以确保整体结构的正确性。

通过深入学习矩阵变换和 OpenGL 的模型视图矩阵，同时参考了补充实验 2 之后，我最终解决了这一问题。

2. **光照模型的实现：**在实现光照效果时，我遇到了理解 Phong 光照模型的难点。

通过线上知识检索、实验 3 参考和实践，我逐步掌握了环境光、漫反射和镜面反射的计算方法，并成功应用到场景中。

3. **用户交互的编程逻辑：**在实现用户交互功能时，需要处理键盘、鼠标事件和相应的图形变换逻辑。

这要求我不仅要有扎实的编程基础，还要对 OpenGL 的交互机制有深入的理解。通过不断调试和优化代码，我提高了代码的健壮性和用户体验。