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 05 课程名称 计算机图形学 主讲教师 熊卫丹 评分

学 号 2022155028 姓名 黄亮铭 专业年级 2022 级软件工程腾班

教师评语：

题目：

俄罗斯方块

中世纪村庄场景建模

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								

目录

1 工程架构

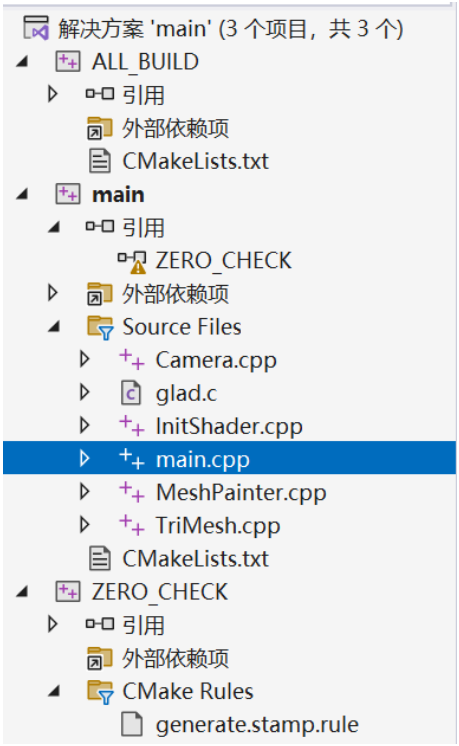

图 1 总体架构

项目的总体架构如图 1 所示。本次大作业涉及到的主要文件为：Camera、MeshPainter、TriMesh 和 main。此外，还包括顶点着色器和片元着色器。

接下来我将介绍每个文件在工程中的作用。

- **Camera**：主要用于在 3D 图形渲染中管理和控制相机的行为，提供了诸如视图矩阵和投影矩阵的计算方法，并支持键盘和鼠标交互来控制相机的移动和旋转等功能。
- **MeshPainter**：主要用于处理和渲染 3D 网格对象及其纹理、着色器、光照等，涉及到 OpenGL 的绘制、纹理加载、光照绑定等操作，常用于 3D 图形渲染应用程序。
- **TriMesh**：主要用于描述 3D 网格模型以及与之相关的光源，提供了对网格模型的几何形状、纹理、法线、颜色等属性的管理和处理方法，同时也包含了一些光照和变换功能。
- **InitShader**：提供了与 OpenGL 着色器相关的一些辅助函数，主要功能是从文件中读取着色器代码并创建一个 OpenGL 着色器程序。
- **Main**：提供了一个标准的框架来设置 OpenGL 环境、处理用户输入、进行渲染，并在窗口关闭时清理资源。

2 具体实现

在经过 2.1 的内容整合之后，所有的实现，如 2.2 的添加纹理，2.3 的添加光照、材质、阴影等效果，2.4 的任意角度浏览场景，2.5 的用户交互控制物体等都可以在 main 文件中进

行实现，而无需再次修改其他文件。

2.1 内容整合

上述工程的代码文件来自前面的各个实验。例如，MeshPainter 类来自实验 4，着色器（顶点着色器、片元着色器）来自实验 3，main 文件中的机器人建模来自实验补充二等等。

我将它们整合到一起，同时修改了部分的代码以更好地实现虚拟场景建模。以绘制 3D 网格图像为例，我在 MeshPainter 类中重载了 drawMesh 函数（图 2），让其更方便绘制纹理的时候同时绘制阴影（实验四的 MeshPainter 类没有提供阴影）。

```
// 增加bool变量：用于指定是否需要绘制阴影
void MeshPainter::drawMesh(int index, glm::mat4 modelMatrix, Light *light, Camera* camera, bool isShadow){
    openGLObject &object = opengl_objects[index];
    TriMesh* mesh = meshes[index];
    // 相机矩阵计算
    camera->updateCamera();
    camera->viewMatrix = camera->getViewMatrix();
    camera->projMatrix = camera->getProjectionMatrix(false);

    glBindVertexArray(object.vao);

    glUseProgram(object.program);

    // 传递矩阵
    glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniformMatrix4fv(object.viewLocation, 1, GL_FALSE, &camera->viewMatrix[0][0]);
    glUniformMatrix4fv(object.projectionLocation, 1, GL_FALSE, &camera->projMatrix[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(object.shadowLocation, 0);

    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

    // 将材质和光源数据传递给着色器
    bindLightAndMaterial(mesh, object, light, camera);

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

    // 绘制阴影
    if (isShadow)
    {
        modelMatrix = light->getShadowProjectionMatrix() * modelMatrix;
        glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
        glUniform1i(object.shadowLocation, 1);
        glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
    }

    glBindVertexArray(0);
```

图 2 重载的 drawMesh 函数

此外，我还对 main 文件中的所有 3D 网格对象进行了统一的处理，方便初始化和绘制（图 3）。例如，我将 3D 网格对象的索引和名字进行存储，然后将 3D 网格对象的缩放倍率、旋转角度等内容统一存储到一个数组中方便管理。通过上述措施，我就可以通过 3D 网格对象的名字直接调用对应的缩放倍率等内容，而无需为每个 3D 网格对象都新增一个缩放倍率变量。

```
// 物体的索引
std::map<std::string, int> meshIndexMap;
// 下标对应上述模型的索引
glm::vec3 meshScale[101] = { glm::vec3(0, 0, 0) };
GLfloat Theta[101] = { 0 };

// 可控制的模型的数量
const int canBeSelectedNum = 100;
// 可控制的模型
std::map<int, std::string> selectedMeshMap;
// 当前控制的mesh的索引
int selectedMesh = 0;
// 当前控制的mesh的名字
std::string selectedMeshName = "camera";
```

图 3 统一管理 3D 网格对象

2.2 添加纹理

添加纹理部分主要在 main 文件的 init 函数中实现，实现的思路如下所示。

1) 读入着色器（顶点着色器和片元着色器）的存储位置（图 4）。

```
std::string vshader, tfshader, noPhongfshader;
// 读取着色器并使用
vshader = "shaders/vshader.glsl";
tfshader = "shaders/tfshader.glsl";
noPhongfshader = "shaders/NoPhongfshader.glsl";
```

图 4 读取着色器

2) 初始化场景中所有的 3D 网格对象，如光源、地板、机器人的各个部位、建筑物和村民等。初始化也即是读取每个 3D 网格对象对应的 obj 文件（图 5）。

```
// 1 地板
meshIndexMap["plane"] = index++;
plane->setNormalize(false);
plane->readObj("assets/plane/plane.obj");
```

图 5 初始化对象（以地板为例）

3) 然后设置每一个对象的位置、旋转、缩放等属性（图 6）。

```
plane->setTranslation(glm::vec3(0.0, -0.1, 0.0));    //加偏移，防止跟阴影重合
plane->setRotation(glm::vec3(180, 0.0, 0.0));
meshScale[meshIndexMap["plane"]] = glm::vec3(25.0, 25.0, 25.0);
plane->setScale(meshScale[meshIndexMap["plane"]]);
```

图 6 设置对象属性

4) 再加载 3D 网格对象的纹理，并将纹理和对应的对象进行绑定操作（图 7）。

```
plane->setShininess(1.0); //高光系数
painter->addMesh(plane, "plane", "assets/plane/plane.png", vshader, tfshader);
```

图 7 加载和绑定

5) 接下来将 3D 网格对象添加到渲染管线中，方便后续的渲染过程。同时，将当前对象添加到对象列表中，方便后续回收内存，防止内存泄露（图 8）。

```
painter->addMesh(plane, "plane
meshList.push_back(plane);
```

图 8 添加到对象列表

6) 最后是设置场景的背景颜色。我将背景颜色设置为蓝色，目的是让背景颜色充当天空。

```
    glClearColor(0.34, 0.64, 0.98, 1.0);
}
```

图 8 设置背景颜色

2.3 添加材质、光照和阴影效果

以太阳为例，添加材质使用到了 setAmbient、setDiffuse 和 setSpecular 这个三个函数，分别代表环境光、漫反射和镜面反射，这三个函数接收的参数均为 glm::vec4（图 9）。

```
// 0 设置光源
meshIndexMap["light"] = index++;
light->readObj("assets/sun/sun.obj");
light->setTranslation(glm::vec3(20.0, 30.0f, 20.0));
light->setScale(glm::vec3(2, 2, 2));
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(0.5, 0.5, 0.5, 0.5)); // 镜面反射
painter->addMesh(light, "light", "assets/sun/sun.png", vshader, tfshader);
meshList.push_back(light);
```

图 9 设置材质

而添加光照部分则是在片元着色器中实现，即通过模拟光照与物体表面交互的过程来实现基本的光照效果（图 10）。具体来说，首先，计算环境光、漫反射光和镜面反射光三个分量。环境光模拟背景光，漫反射光反映光源照射到物体表面并散射的效果，而镜面反射光则模拟光线反射后在平滑表面形成的高光部分。然后着色器通过法线、光源方向和观察方向之间的关系，结合物体的材质属性（如漫反射、镜面反射系数和光泽度）计算出最终的光照颜色。最后，光照分量与纹理颜色相乘，得到物体的最终渲染效果。

```
// @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 H = normalize(L + V);

// 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(L, N), 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 s_t = pow(dot(N, H), material.shininess);
float specular_dot_pow = max(s_t, 0.0);
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
vec4 phone = I_a + I_d + I_s;
fColor = texture2D(texture, texCoord) * phone;
```

图 10 添加光照

阴影实际上在 2.1 的整合的文件中已经实现。在重载的 drawMesh 中，计算阴影矩阵和当前模视矩阵的乘积得到阴影的绘制位置，然后传递绘制阴影的变量到着色器中，将绘制的颜色改为黑色（图 11）。

```
// 绘制阴影
if (isShadow)
{
    modelMatrix = light->getShadowProjectionMatrix() * modelMatrix;
    glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniform1i(object.shadowLocation, 1);
    glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size());
}

glBindVertexArray(0);

glUseProgram(0);
```

图 11 添加阴影

2.4 任意角度浏览场景

该部分的实现在 Camera 的键盘响应函数和鼠标相应函数中。控制权传递到 Camera 的过程为（图 12）：键盘和鼠标的回调函数会通过判断当前控制的 3D 网格对象是否为相机，从而决定是否将控制权移交给 Camera。

```
if (selectedMeshMap[selectedMesh] != "camera") { ... }
else {
    camera->keyboard(window);
}
```

图 12 移交控制权判断条件

2.4.1 键盘

键盘部分通过 WASD 键和 CTRL、ALT 键实现相机的平移和上下移动。当按下 W 键时，相机会沿着视线方向（front）前进，而按下 S 键时，则会沿着相反方向后退。按下 A 键会使相机向左移动，按下 D 键则会使相机向右移动，这两个方向是通过计算相机视线方向（front）与上方向（up）的叉积获得的。此外，按下 CTRL 键时，相机会沿着上方向（up）上升，而按下 ALT 键时，相机会沿着下方向下降。为了防止相机越界，代码还对相机的位置进行了限制，确保相机始终在预定的场景范围内。具体的代码如图 13 所示。

```
void Camera::keyboard(GLFWwindow* window)
{
    // 键盘事件处理
    // WASD控制相机前后左右移动
    int W = glfwGetKey(window, GLFW_KEY_W), S = glfwGetKey(window, GLFW_KEY_S);
    int A = glfwGetKey(window, GLFW_KEY_A), D = glfwGetKey(window, GLFW_KEY_D);
    // CTRL和ALT控制相机上升和下降
    int CTRL = glfwGetKey(window, GLFW_KEY_LEFT_CONTROL), ALT = glfwGetKey(window, GLFW_KEY_LEFT_ALT);
    int SPACE = glfwGetKey(window, GLFW_KEY_SPACE);
    if (W == GLFW_PRESS || W == GLFW_REPEAT)
    {
        eye += cameraSpeed * front;
    }
    else if (S == GLFW_PRESS || S == GLFW_REPEAT)
    {
        eye -= cameraSpeed * front;
    }
    else if (A == GLFW_PRESS || A == GLFW_REPEAT)
    {
        glm::vec3 ftmp = front;
        glm::vec3 utmp = up;
        glm::vec4 delta = glm::vec4(cameraSpeed * glm::normalize(glm::cross(ftmp, utmp)), 1.0);
        eye -= delta;
    }
    else if (D == GLFW_PRESS || D == GLFW_REPEAT)
    {
        glm::vec3 ftmp = front;
        glm::vec3 utmp = up;
        glm::vec4 delta = glm::vec4(cameraSpeed * glm::normalize(glm::cross(ftmp, utmp)), 1.0);
        eye += delta;
    }
    else if (CTRL == GLFW_PRESS || CTRL == GLFW_REPEAT) {
        eye += cameraSpeed * up;
    }
    else if (ALT == GLFW_PRESS || ALT == GLFW_REPEAT) {
        eye -= cameraSpeed * up;
    }
    //空气墙
    eye.x = std::min(25.f, eye.x);
    eye.x = std::max(-25.f, eye.x);
    eye.y = std::min(35.00f, eye.y);
    eye.y = std::max(0.1f, eye.y);
    eye.z = std::min(25.f, eye.z);
    eye.z = std::max(-25.f, eye.z);
}
```

图 13 相机的键盘响应函数

2.4.2 鼠标

鼠标部分控制相机的旋转，通过改变相机的俯仰和偏航角度来实现。当鼠标移动时，计算出当前鼠标位置与上次位置的偏移量，分别应用于 水平旋转和垂直旋转。鼠标的水平移动控制相机的左右旋转，而垂直移动则控制相机的上下旋转。为了避免过度俯视或仰视，俯

仰角度被限制在 -89.9 到 89.9 度之间。通过这种方式，用户可以通过鼠标的移动精确控制相机视角，实现自由的场景浏览。具体的代码如图 14 所示。

```
// 鼠标控制摄像头方向，利用欧拉角
void Camera::mouse(double xpos, double ypos)
{
    // 如果鼠标是第一次移动，直接初始化lastX和lastY
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    // 注意要反转y轴，因为原点在左上方;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    if (pitch > 89.9f)
        pitch = 89.9f;
    if (pitch < -89.9f)
        pitch = -89.9f;
}
```

图 14 相机的键盘响应函数

2.5 用户交互控制物体

通过键盘控制物体，如机器人各个部件、三个村民和光源的位置和旋转。

用户可以通过数字键（1-9）选择不同的机器人部件（图 15），然后使用 CTRL 和 ALT 键来调整选中部件的旋转角度（图 16）。

```
        case GLFW_KEY_1:
            selectedMesh = robot.Torso;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_2:
            selectedMesh = robot.Head;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_3:
            selectedMesh = robot.LeftUpperArm;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_4:
            selectedMesh = robot.LeftLowerArm;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_5:
            selectedMesh = robot.Sword;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_6:
            selectedMesh = robot.RightUpperArm;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_7:
            selectedMesh = robot.RightLowerArm;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_8:
            selectedMesh = robot.LeftUpperLeg;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_9:
            selectedMesh = robot.LeftLowerLeg;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_0:
            selectedMesh = robot.RightUpperLeg;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
        case GLFW_KEY_MINUS:
            selectedMesh = robot.RightLowerLeg;
            selectedMeshName = selectedMeshMap[selectedMesh];
            break;
```

图 15 数字键选择部件

```
1222            // 通过按键旋转
1223            case GLFW_KEY_LEFT_CONTROL:
1224                if (action == GLFW_PRESS) {
1225                    KeyMap["CTRL"] = true;
1226                }
1227                else if (action == GLFW_RELEASE) {
1228                    KeyMap["CTRL"] = false;
1229                }
1230                break;
1231
1232            case GLFW_KEY_LEFT_ALT:
1233                if (action == GLFW_PRESS) {
1234                    KeyMap["ALT"] = true;
1235                }
1236                else if (action == GLFW_RELEASE) {
1237                    KeyMap["ALT"] = false;
1238                }
1239                break;
```

图 16 调整旋转角度

使用空格键选定控制的 3D 网格对象（图 17）后，按下 W、A、S、D 键控制物体在三维空间中的前后左右移动（图 18a），且每个移动都受到空气墙的限制。

```
// 相机
case GLFW_KEY_SPACE:
    if (action == GLFW_PRESS) {
        if (selectedMesh == 2) {
            //camera->radius = camera->radius + 10;
            selectedMesh = 0;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
        else if (selectedMesh == 0) {
            selectedMesh = 12;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
        else if (selectedMesh == 12) {
            selectedMesh = 13;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
        else if (selectedMesh == 13) {
            selectedMesh = 14;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
        else if (selectedMesh == 14) {
            selectedMesh = 1;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
        else {
            selectedMesh = 0;
            selectedMeshName = selectedMeshMap[selectedMesh];
        }
    }
    break;
```

图 17 空格键选定对象

按下 I、J、K、L 键则可以移动光源的位置（图 18b）。

```
// W/S控制前进后退，A/D控制左右转向
if (KeyMap["W"])
    translation += glm::vec3(dist * sin(glm::radians(angle)), 0.0, dist * cos(glm::radians(angle)));
if (KeyMap["S"])
    translation -= glm::vec3(dist * sin(glm::radians(angle)), 0.0, dist * cos(glm::radians(angle)));
if (KeyMap["A"])
{
    Theta[meshIndexMap[selectedMeshName]] += 2.0;
    if (Theta[meshIndexMap[selectedMeshName]] > 360.0f) Theta[meshIndexMap[selectedMeshName]] -= 360.0f;
}
if (KeyMap["D"])
{
    Theta[meshIndexMap[selectedMeshName]] -= 2.0;
    if (Theta[meshIndexMap[selectedMeshName]] < 0.0f) Theta[meshIndexMap[selectedMeshName]] += 360.0f;
}
```

图 18a WASD 控制网格对象

```
// 光源控制
if (KeyMap["I"] || KeyMap["J"] || KeyMap["K"] || KeyMap["L"])
{
    glm::vec3 translation = light->getTranslation();
    // 获得移动方向
    int zDirection = 0, xDirection = 0;
    if (KeyMap["I"]) {
        zDirection = -1;
    }
    else if (KeyMap["K"]) {
        zDirection = 1;
    }
    else if (KeyMap["J"]) {
        xDirection = -1;
    }
    else if (KeyMap["L"]) {
        xDirection = 1;
    }

    // 空气墙
    float dist = 0.1;
    if (translation.z + zDirection * dist < 25.f && translation.z + zDirection * dist > -25.f) translation.z += zDirection * dist;
    if (translation.x + xDirection * dist < 25.f && translation.x + xDirection * dist > -25.f) translation.x += xDirection * dist;
    light->setTranslation(glm::vec3(translation.x, translation.y, translation.z));
}
```

图 18b IJKL 控制光源

通过这些键盘输入，用户能够实时调整和操控物体的位置、方向和旋转，使得整个场景可以根据用户的需求进行动态变化。

2.6 机器人的层次建模

一个机器人可以抽象成如图 19 所示。

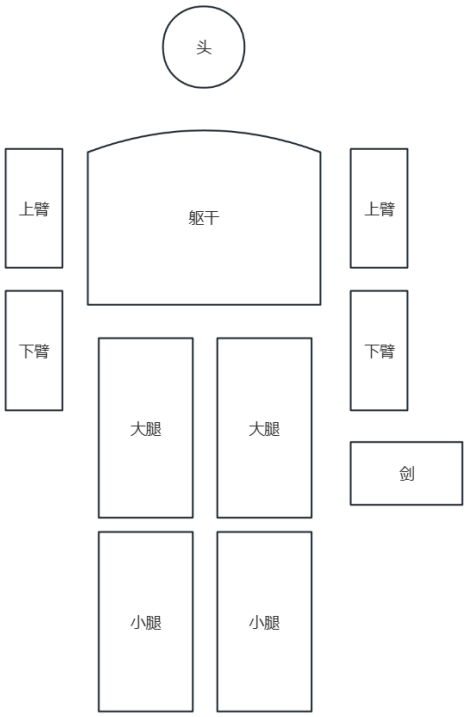

图 19 抽象机器人

层次建模的关键思想是将每个部件的位置、旋转和缩放与其父部件相对定位，从而形成一个父子关系的层次结构。

机器人的层级结构如图 20 所示。

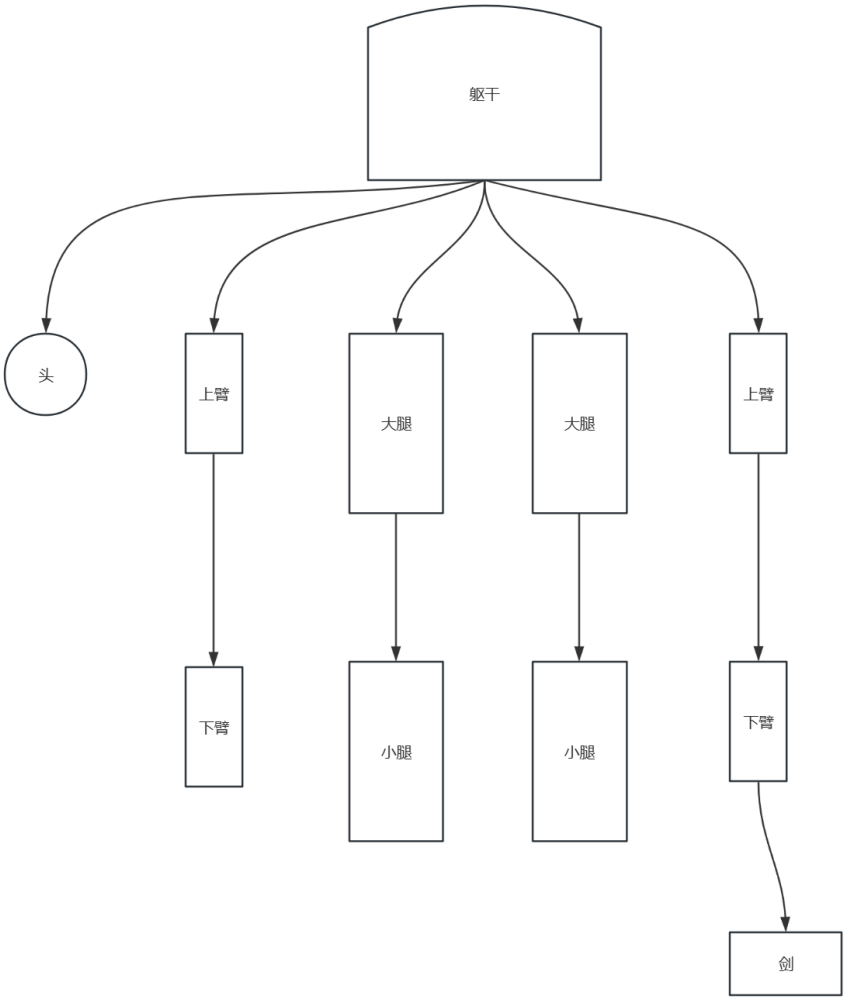

图 20 机器人的层次结构框架图

首先使用 mstack 作为一个堆栈来存储和恢复模型矩阵。每个部件，如躯干、头、四肢，都有自己的变换矩阵，这些矩阵通过堆栈进行管理。每当对子部件进行变换时，会先保存当前的模型矩阵（如果还有同一分支的同层次的模型），再应用相应的变换，最后从堆栈中弹出恢复之前的矩阵，确保层次结构正确。

然后是逐层变换，即每个部件的变换都是基于其父部件的变换进行的。通过使用堆栈保存当前的模型矩阵，逐层对每个部件应用相对变换。例如，躯干作为基准，所有其他部件都相对于躯干的位置进行平移、旋转和缩放。每次对子部件进行变换时，先保存当前矩阵，再应用变换后绘制部件，最后恢复父部件的变换矩阵。这种方式保证了每个部件的变换是层次化的，从而确保部件之间的正确相对定位和旋转，最终形成完整的模型。

最后，每次计算完一个部件的变换矩阵，使用 drawMesh 函数将该部件绘制出来。这个过程会按照层次结构依次执行，确保子部件在父部件变换的影响下进行渲染。

这里给出左上臂、左下臂和剑的层级建模（图 21）。

```
// 左大臂
modelMatrix = mstack.pop();
mstack.push(modelMatrix);
modelMatrix = glm::translate(modelMatrix, glm::vec3((Torso->getLength() + LeftUpperArm->getLength()) / 2, Torso->getHeight() - LeftUpperArm->getHeight(), 0.0));
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, LeftUpperArm->getHeight(), 0.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(Theta[meshIndexMap["LeftUpperArm"]]), glm::vec3(1.0, 0.0, 0.0));
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, -LeftUpperArm->getHeight(), 0.0));
//modelMatrix = glm::scale(modelMatrix, meshScale[meshIndexMap["LeftUpperArm"]]);
painter->drawMesh(meshIndexMap["LeftUpperArm"], modelMatrix, light, camera, 1);

// 左小臂
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, -LeftLowerArm->getHeight() + bias, 0.0));
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, LeftLowerArm->getHeight(), 0.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(Theta[meshIndexMap["LeftLowerArm"]]), glm::vec3(1.0, 0.0, 0.0));
modelMatrix = glm::translate(modelMatrix, glm::vec3(0.0, -LeftLowerArm->getHeight(), 0.0));
painter->drawMesh(meshIndexMap["LeftLowerArm"], modelMatrix, light, camera, 1);

// 剑
modelMatrix = glm::translate(modelMatrix, glm::vec3(0, 0. * (LeftLowerArm->getHeight() + LeftLowerArm->getHeight()), 5.0));
/*modelMatrix = glm::translate(modelMatrix, glm::vec3(-LeftLowerArm->getLength() * 7.5, Sword->getHeight() * 12, 0.0));*/
modelMatrix = glm::translate(modelMatrix, glm::vec3(0, -0. * (LeftLowerArm->getHeight() + LeftLowerArm->getHeight()), -5.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(45 + Theta[meshIndexMap["Sword"]]), glm::vec3(1.0, 0.0, 0.0));
//modelMatrix = glm::rotate(modelMatrix, glm::radians(-90.0f), glm::vec3(0.0, 1.0, 0.0));
/*modelMatrix = glm::translate(modelMatrix, glm::vec3(LeftLowerArm->getLength() * 7.5, -Sword->getHeight() * 12, 0.0));*/
modelMatrix = glm::translate(modelMatrix, glm::vec3(0, 0. * (LeftLowerArm->getHeight() + LeftLowerArm->getHeight()), 5.0));
modelMatrix = glm::scale(modelMatrix, glm::vec3(15, 15, 15));
painter->drawMesh(meshIndexMap["Sword"], modelMatrix, light, camera, 1);
```

图 21 机器人的左手层级建模

3 鼠标和键盘的交互使用方法

3.1 鼠标交互使用方法

鼠标可以上下左右移动，用于控制相机的视角方向，同时配合键盘按键移动相机位置。

3.2 键盘的交互使用方法

- **空格**：空格用于切换控制权，分别为相机控制权、三个村民的控制权以及机器人的控制权。
- **WASD**：控制相机、村民或者机器人的移动。

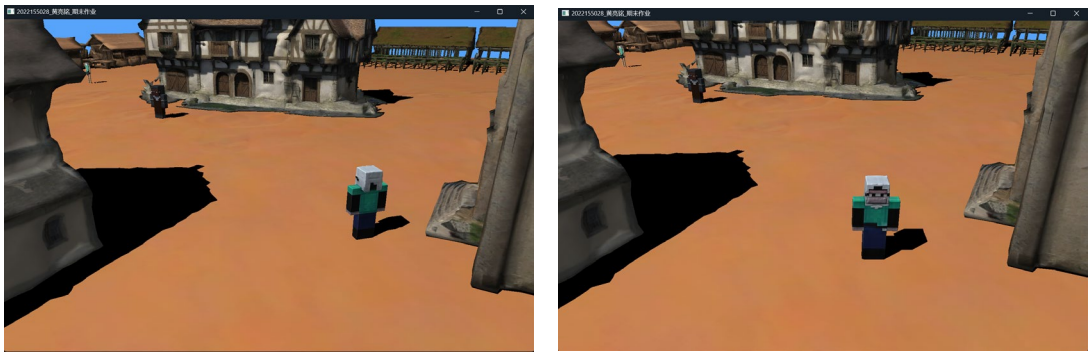

图 22 控制人物移动

- **CTRL 和 ALT**：1）控制相机的上升与下降，有利于提高相机的操控性，无需鼠标配合键盘就能实现相机的上升与下降；2）控制机器人身体的各个部位的旋转。

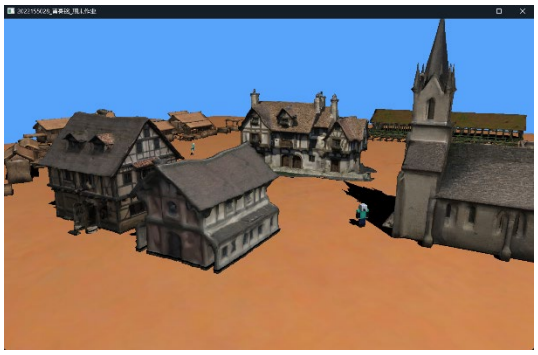


图 23 相机上升与下降

- **IJKL**：用于控制太阳模型移动，同时可以改变光照角度，进而改变阴影的朝向。

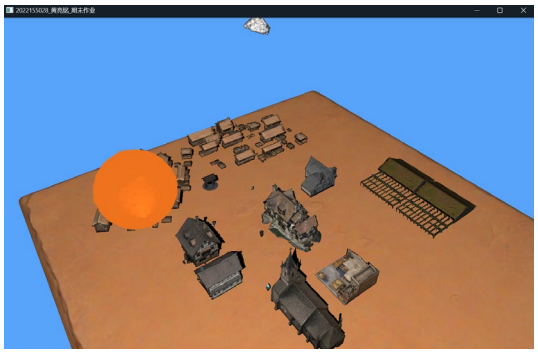


图 24 太阳移动


图 25 阴影变化

- **数字键以及“-”**：用于将控制权移交给机器人身体的各个部位，按照排列数字键 1-9、0、-，它们依次控制的部位为躯干、头部、左上臂、左下臂、剑、右上臂、右下臂、左大腿、左小腿、右大腿、右小腿。

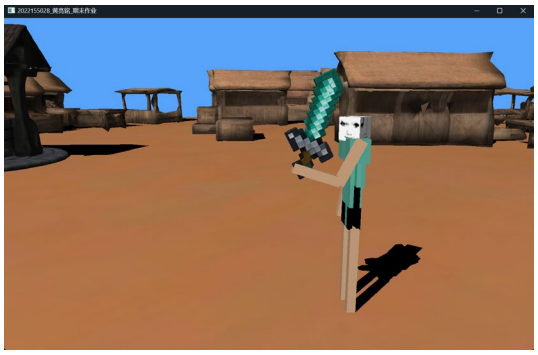


图 26 控制机器人

- **ESC**：退出程序。

4 模型绘制结果展示

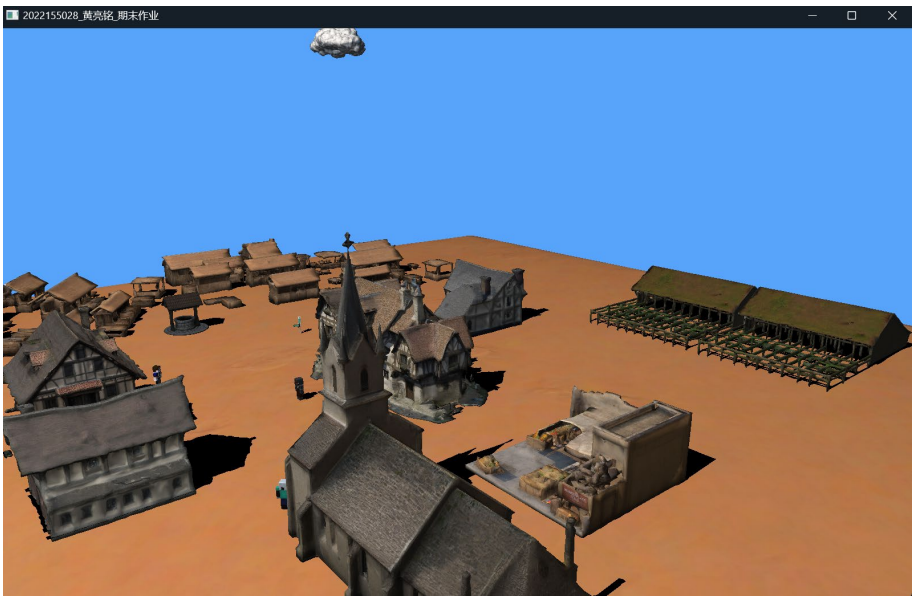

图 27 场景截图