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 2022150130 姓名 林宪亮 专业年级 计算机科学与技术 大三

教师评语：

题目： **宫殿遗迹**

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								

概览：

一 完成的功能

二 场景介绍

三 实现细节

一 完成的功能

1. 场景设计和显示：

使用层次建模建立了两个物体，一个是末影守卫（四层层次建模），主角（两层层次建模）。建立了一个遗迹宫殿，该宫殿拥有超过十五个虚拟物体，包含地面和天空。

2. 添加纹理：

我为场景中的所有物体都添加了纹理贴图，贴图数量超过十五个。

3. 添加光照，材质，阴影效果：

我为场景中除了游戏设计所需的物体天骄了光照材质和阴影效果。

4. 用户交互实现视角切换完成对场景的任意角度浏览：

我设计了四个相机视角，可以完成对场景的任意角度浏览。

5. 通过交互控制物体：

可以通过键盘鼠标交互控制主角以及末影龙，包括移动，跳跃，加速，攻击,旋转等等。

6. 动画效果

末影守卫在宫殿遗迹中自动移动巡逻。

达到了所有的要求！！！

二 场景介绍

召唤师，您好，欢迎来到古龙族的宫殿遗迹。您是一位从 2098 年穿越而来的时空冒险者，这次你的冒险目的地是龙族的宫殿遗迹，十万年前，这是龙王的宫殿，但现在却只是一座废墟，但你需要注意龙王残念所化的末影龙和末影守卫。您将在这开启一段惊险刺激的冒险，在开始我们的探险前，请允许我为你介绍这座宫殿遗迹。

1. 宫殿遗迹介绍：

1.1 宫殿遗迹全览图

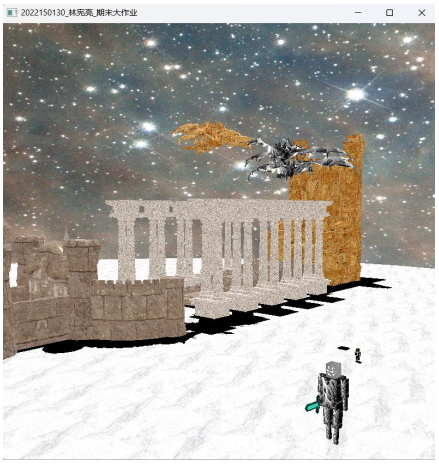

图 1 第三视角

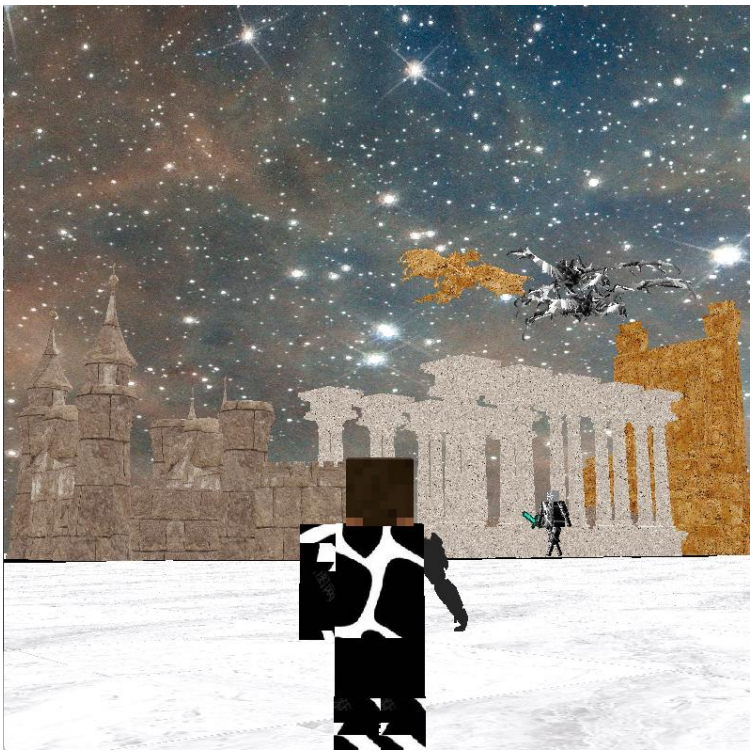

图 2 第一视角

如图 1 和图 2 分别是第三视角和第一视角看到的宫殿全览图。

1.2 宫殿守护者

图 3 末影龙

如图为宫殿的守护者 1，末影龙，它是远古龙王留下的一丝残念所化，因为是末影龙，所以它是没有影子的。

图 4 末影守卫

这是宫殿的守护者 2，末影守卫，同理，它也是没有影子的。

1.3 宫殿建筑

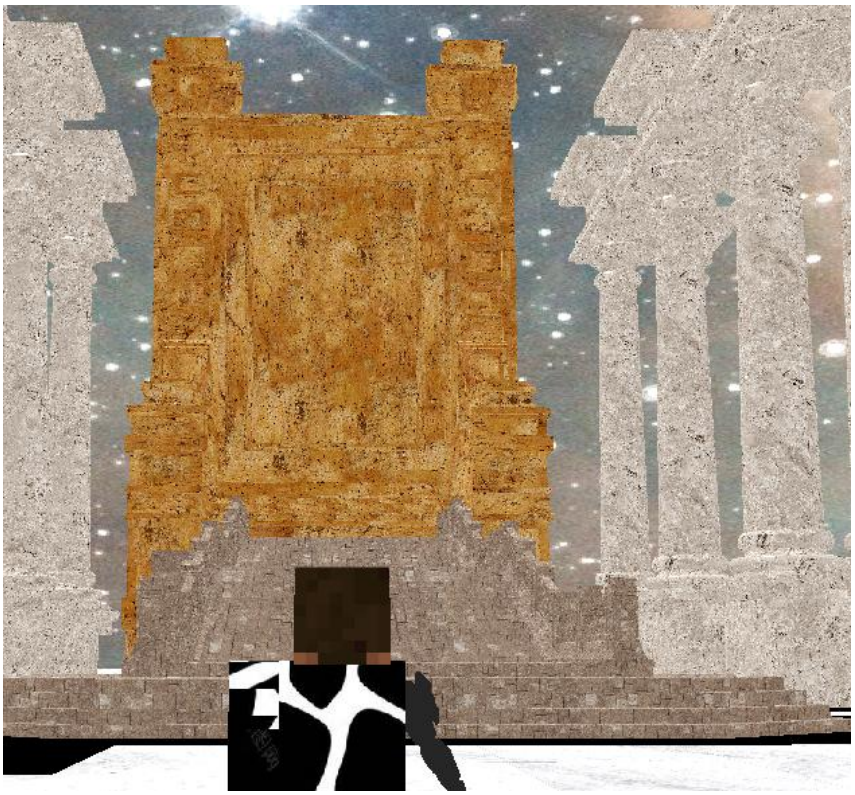

图 5 龙椅和龙梯

如图，这是远古龙王的龙椅，以及登上龙椅的阶梯。

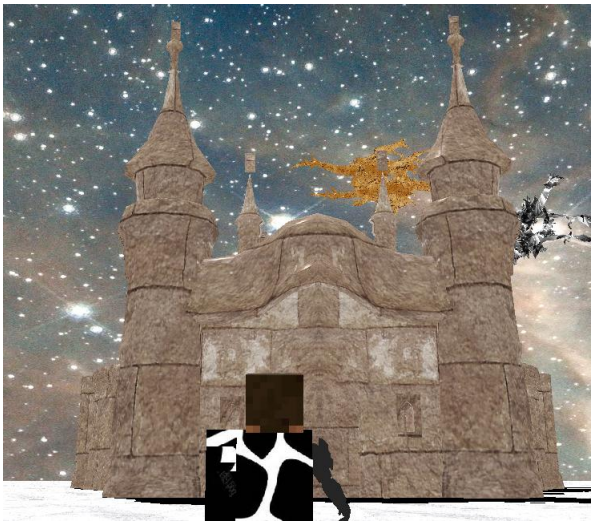

图 6 龙堡大门

这是龙堡的宫殿大门。

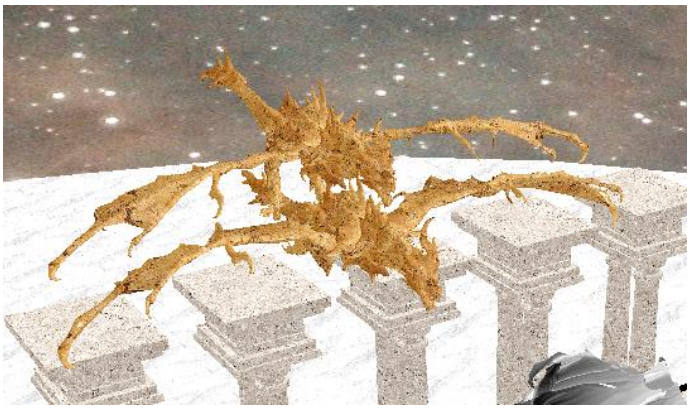

图 7 龙雕

如图，这是黄金龙雕。

图 8 龙柱

这是撑起龙族世界的龙柱。

三 实现细节

1. 场景搭建

```
// 主光源
light->setTranslation(lightPos);
light->setAmbient(glm::vec4(1.2, 1.2, 1.2, 1.0)); // 环境光稍强
light->setDiffuse(glm::vec4(1.0, 1.0, 1.0, 1.0));
light->setSpecular(glm::vec4(1.0, 1.0, 1.0, 1.0));
light->setAttenuation(1.0, 0.02, 0.001);
// 辅助光源（模拟反射光）
auxLight->setTranslation(glm::vec3(0.0, 10.0, 0.0));
auxLight->setAmbient(glm::vec4(0.4, 0.4, 0.4, 1.0)); // 提供浅色阴影效果
auxLight->setDiffuse(glm::vec4(0.3, 0.3, 0.3, 1.0));
auxLight->setSpecular(glm::vec4(0.3, 0.3, 0.3, 1.0));
auxLight->setAttenuation(1.0, 0.1, 0.01);
//初始化人物各个部分
//躯干
Torso->setNormalize(false);
Torso->generateCube();
Torso->setTranslation(glm::vec3(0.0, -10.0, 0.0));
Torso->setRotation(glm::vec3(0.0, 0.0, 0.0));
Torso->setScale(glm::vec3(1.0, 1.0, 1.0));
Torso->setAmbient(glm::vec4(0.05f, 0.05f, 0.05f, 1.0f)); // 环境光
Torso->setDiffuse(glm::vec4(0.5f, 0.5f, 0.5f, 1.0f)); // 漫反射
Torso->setSpecular(glm::vec4(0.7f, 0.7f, 0.7f, 1.0f)); // 镜面反射
Torso->setShininess(32.0f); //高光系数
painter->addMesh(Torso, "Torso", "./assets/sword/OIP.png", vshader, fshader2);
......
// 初始化角色模型的骨架结构
lxlman = new LXLMAN();
// 创建角色右腿并设置其相对位置和比例
lxlmanRightLeg = new treenode(lxlman->RightLeg, glm::vec3(-0.13, -0.71, 0.0), glm::vec3(0.0, 0.0, 0.0),
 glm::vec3(0.9, 0.9, 0.9),
 NULL, NULL, glm::translate(glm::mat4(1.0), glm::vec3(0.0, -0.35, 0.0)));
// 创建角色左腿
lxlmanLeftLeg = new treenode(lxlman->LeftLeg, glm::vec3(0.13, -0.71, 0.0), glm::vec3(0.0, 0.0, 0.0), gl
m::vec3(0.9, 0.9, 0.9),
 NULL, lxlmanRightLeg, glm::translate(glm::mat4(1.0), glm::vec3(0.0, -0.35, 0.0)));
......
// 创建 jieti 模型
addTriMesh("objStair", glm::vec3{ 37.0, 2.57, 25.0 }, glm::vec3{ 0.0, 180.0, 0.0 }, glm::vec3{ 35, 35,
35 },
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.5, 0.5, 0.5, 1.0 }, 225,
 true);
// 创建天空盒
addTriMesh("sky", glm::vec3{ 35.0, 0.0, 5.0 }, glm::vec3{ 0.0, 0.0, 0.0 }, glm::vec3{ 300.0, 300.0, 300.
0 },
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.0, 0.0, 0.0, 1.0 }, 225,
 false);
```

```
// 创建宫殿
addTriMesh("Palace", glm::vec3{ 37.0, 10.62, -15 }, glm::vec3{ 0.0, 0.0, 0.0 }, glm::vec3{ 45, 45, 45 }
,
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.5, 0.5, 0.5, 1.0 }, 225,
 true);
//long
addTriMesh("dalong", glm::vec3{ 32.0, 27, 10 }, glm::vec3{ 0.0, -90.0, 0.0 }, glm::vec3{ 22.0, 22.0, 22.
0 },
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.5, 0.5, 0.5, 1.0 }, 96.
078431, true);
......
//柱子
addTriMesh("objPillar", glm::vec3{ 27.0, 7.9, 7.0 }, glm::vec3{ 0.0, 0.0, 0.0 }, glm::vec3{ 20.0, 20.0,
 20.0 },
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.5, 0.5, 0.5, 1.0 }, 96.
078431, true);
......
addTriMesh("objPillar", glm::vec3{ 47.0,  7.9, 7.0 }, glm::vec3{ 0.0, 0.0, 0.0 }, glm::vec3{ 20.0, 20.0,
 20.0 },
 glm::vec4{ 0.0, 0.0, 0.0, 0.0 }, glm::vec4{ 0.2, 0.2, 0.2, 1.0 }, glm::vec4{ 0.5, 0.5, 0.5, 1.0 }, 96.
078431, true);
......
// 创建龙椅
addTriMesh("objChair", glm::vec3{37, 9.7, 47.0}, glm::vec3{0.0, 180.0, 0.0}, glm::vec3{60.0, 60.0, 60.0}
,
   glm::vec4{0.0, 0.0, 0.0, 0.0}, glm::vec4{0.2, 0.2, 0.2, 1.0}, glm::vec4{0.5, 0.5, 0.5, 1.0}, 96.0784
31, true);
glClearColor(0.68f, 0.85f, 0.9f, 1.0f);
```

在这段代码中，我正在为一个三维场景设置光源、模型、纹理和结构。首先，我初始化了主光源和辅助光源，并设置了它们的光照属性，如环境光、漫反射、镜面反射和衰减系数。主光源的环境光强度较高，以提供更亮的整体照明，而辅助光源则用于模拟反射光，创造更自然的阴影效果。接下来，我初始化了一个虚拟角色的各个部件，包括躯干、头部、手臂和腿部。每个部件都经过了旋转、缩放和位置调整，还设置了环境光、漫反射、镜面反射和高光系数，以模拟材质的光照反应。每个部件都通过 painter->addMesh 方法被加入到场景中，并且都附带了相应的纹理。

在设置角色模型时，我还通过 treenode 类初始化了角色的骨架结构，定义了各个部件之间的父子关系，并对它们的位置和比例进行了适当调整。这种层级结构确保了模型各部件在动画和变换时能够协调一致地运动。

另外，我还为场景添加了一个 30x30 的草方块地面，这些草方块被分布在一个大的网格中，作为游戏环境的一部分。每个方块也有自己的光照设置，包括环境光、漫反射和镜面反射。

除此之外，我还创建了多个场景中的静态物体，比如一个楼梯模型、天空盒和宫殿建筑。它们的设置包括位置、旋转、缩放以及相应的光照参数。每个物体也都有适当的纹理，以增强视觉效果。通过这种方式，我将所有的元素都精心设置，确保场景中各个部分相互协作，营造出一个丰富且有层次的三维环境。

2. 层次建模

末影守卫

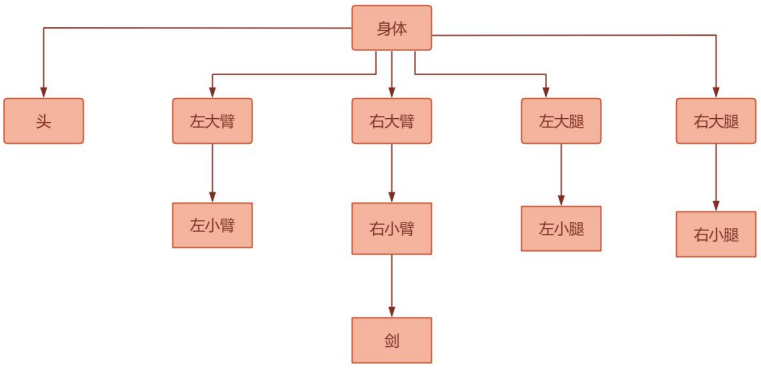

图 9 末影守卫层次建模

主角

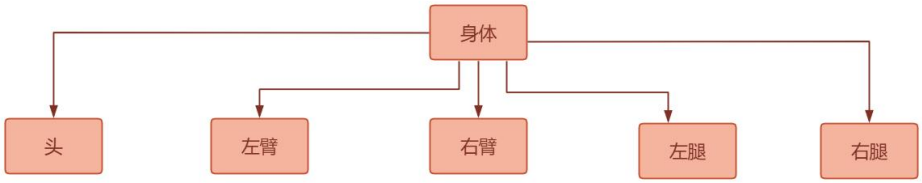

图 10 主角层次建模

```
// 如果当前节点为空，直接返回
if (node == NULL) return;
// 保存当前的变换矩阵到栈中
mstack.push(modelMatrix);
// 应用当前节点的平移变换
modelMatrix = glm::translate(modelMatrix, node->translation);
// 应用当前节点的旋转变换（Z 轴、Y 轴、X 轴顺序）
modelMatrix = glm::rotate(modelMatrix, glm::radians(node->rotation[2]), glm::vec3(0.0, 0.0, 1.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(node->rotation[1]), glm::vec3(0.0, 1.0, 0.0));
modelMatrix = glm::rotate(modelMatrix, glm::radians(node->rotation[0]), glm::vec3(1.0, 0.0, 0.0));
// 应用当前节点的缩放变换
modelMatrix = glm::scale(modelMatrix, node->scale);
// 设置当前节点的材质属性
lxlman->triMesh[node->nodeId]->setAmbient(glm::vec4{ 0.0, 0.0, 0.0, 0.0 });  // 设置环境光
lxlman->triMesh[node->nodeId]->setDiffuse(glm::vec4{ 0.2, 0.2, 0.2, 1.0 });  // 设置漫反射
lxlman->triMesh[node->nodeId]->setSpecular(glm::vec4{ 0.5, 0.5, 0.5, 1.0 }); // 设置镜面反射
lxlman->triMesh[node->nodeId]->setShininess(225);                            // 设置高光系数
// 如果当前节点需要绘制，则进行渲染
if (node->drawFlag == true) {
 painter->drawMeshWithM(
  lxlman->triMesh[node->nodeId],              // 当前节点的三角网格
  modelMatrix * node->selfTransformM,         // 当前节点的变换矩阵
  lxlman->object[node->nodeId],               // 渲染对象
  light,                                      // 光源
```

```
 camera,                                    // 摄像机
 lxlman->lxlmanPos,                         // 模型位置
 lxlman->lxlmanDir                          // 模型方向
 );
}
// 如果当前节点有子节点，递归遍历子节点
if (node->child != NULL) {
 traverse(mstack, node->child, modelMatrix);
}
// 恢复变换矩阵
modelMatrix = mstack.pop();
// 如果当前节点有兄弟节点，递归遍历兄弟节点
if (node->sibling != NULL) {
 traverse(mstack, node->sibling, modelMatrix);
}
```

在这段代码中，我实现了一个递归遍历并更新模型变换的功能。主要通过一个 traverse 函数来处理每个节点的变换和渲染。首先，我会检查当前节点是否为空，如果为空则返回并结束递归。接着，我会将当前变换矩阵保存到栈中，并依次应用当前节点的平移、旋转和缩放变换，更新变换矩阵。

对于每个节点，我还会根据其材质属性设置相应的环境光、漫反射、镜面反射和高光系数。如果该节点需要绘制，我会调用渲染函数绘制该节点的三角网格。

在遍历节点的过程中，如果当前节点有子节点，我会递归地遍历它们，并确保在完成子节点的处理后恢复原来的变换矩阵。此外，如果当前节点有兄弟节点，我还会继续遍历兄弟节点，确保整个渲染都能够正确处理。

3. 动画效果

```
// 定义运动参数
float armSwingAmplitude = 30.0f;   // 手臂摆动幅度（单位：度）
float legSwingAmplitude = 30.0f;   // 腿部摆动幅度（单位：度）
float swingFrequency = 2.0f;       // 摆动频率（单位：Hz）
// 获取时间
float time = glfwGetTime();
float swingAngle = armSwingAmplitude * sin(swingFrequency * time); // 摆动角度
// 定义位移（沿圆周路径）
float movementRadius = 20.0f;
float movementSpeed = 0.05f;
float angle = movementSpeed * time;
place.x = movementRadius * cos(angle);
place.z = movementRadius * sin(angle);
// =========== 躯干 ===========
glm::mat4 modelMatrix2 = glm::mat4(1.0);
modelMatrix2 = glm::translate(modelMatrix2, place);
modelMatrix2 = glm::rotate(modelMatrix2, glm::radians(robot.theta[robot.Torso]), glm::vec3(0.0, 1.0, 0.
0));
modelMatrix2 = glm::scale(modelMatrix2, glm::vec3(0.5, 0.5, 0.5));
```

```
torso(modelMatrix2);
MatrixStack mstack2;
mstack2.push(modelMatrix2); // 保存躯干变换矩阵
......
```

在这段代码中，我实现了两种动画效果：手臂和腿部的摆动，以及物体沿圆周路径的运动。首先，我定义了手臂和腿部的摆动幅度为 30.0f 度，并设置了摆动频率为 2.0f Hz，这意味着每秒钟会进行两次摆动。接着，我通过 glfwGetTime() 获取当前的时间，并利用正弦函数计算出当前的摆动角度。这让摆动角度随时间变化，形成周期性的动作。

然后，我实现了物体沿圆周路径的运动。我定义了圆周的半径 20.0f 和运动速度 0.05f，通过将时间与速度相乘，计算出物体在圆周上的角度。根据这个角度，我更新了物体的 x 和 z 坐标，使物体沿着圆周路径运动。

通过这种方式，我能够同时模拟手臂和腿部的自然摆动以及物体在平面上的运动。

4. 视角变换

```
if(cameraMode == 1){
  camera->mouse(deltaX, deltaY);
 }
 else if(cameraMode == 2 || cameraMode == 3){
  if (deltaX > 0) {
lxlmanBody->rotation[1] -= abs(deltaX) * (cameraMode == 2 ? 0.6f : 0.3f);  // 鼠标向右滑动
  }
  else if (deltaX < 0) {
lxlmanBody->rotation[1] += abs(deltaX) * (cameraMode == 2 ? 0.6f : 0.3f);  // 鼠标向左滑动
  }
  if (deltaY > 0) {
lxlmanHead->rotation[0] = glm::min(lxlmanHead->rotation[0] + abs(deltaY) * (cameraMode == 2 ? 0.05f : 0.
1f), (cameraMode == 2 ? 89.9f : 50.0f));
  }
  else if (deltaY < 0) {
lxlmanHead->rotation[0] = glm::max(lxlmanHead->rotation[0] - abs(deltaY) * (cameraMode == 2 ? 0.05f : 0.
1f), -89.9f);
  }
 }
 else if(cameraMode == 4){
  if (deltaX > 0) {
      lxlmanBody->rotation[1] += abs(deltaX) * 0.7f;  // 鼠标向右滑动
  }
  else if (deltaX < 0) {
lxlmanBody->rotation[1] -= abs(deltaX) * 0.7f;  // 鼠标向左滑动
  }
  if (deltaY > 0) {
lxlmanHead->rotation[0] = glm::min(lxlmanHead->rotation[0] + abs(deltaY) * 0.1f, 89.9f);
  }
  else if (deltaY < 0) {
lxlmanHead->rotation[0] = glm::max(lxlmanHead->rotation[0] - abs(deltaY) * 0.1f, -89.9f);
  }
 }
```

在这段代码中，我通过不同的 cameraMode 来控制视角的旋转，具体来说，我让鼠标的移动影响角色或相机的旋转。首先，当 cameraMode 为 1 时，我直接通过 camera->mouse(deltaX, deltaY) 控制相机的视角。这意味着鼠标的移动会影响相机的方向。
当 cameraMode 为 2 或 3 时，鼠标的水平移动（deltaX）会让我控制角色身体的旋转。具体来说，如果鼠标向右移动（deltaX > 0），角色会向左旋转；如果鼠标向左移动（deltaX < 0），角色会向右旋转。而垂直移动（deltaY）则控制角色头部的旋转，向上滑动会让头部向上转，向下滑动则让头部向下转。为了避免头部旋转超过一定的角度，我使用了 glm::min 和 glm::max 来限制旋转范围，分别限制为 89.9 度和 -89.9 度。

此外，cameraMode == 2 和 cameraMode == 3 之间的差异在于旋转速度，cameraMode == 2 的旋转较慢，而 cameraMode == 3 的旋转较快，分别通过不同的速度系数来调整。
最后，当 cameraMode 为 4 时，旋转速度更快，水平旋转幅度设为 0.7f，垂直旋转幅度为 0.1f，并且同样设置了头部旋转的角度限制。

5. 动作交互

```
// 如果 W, S, A, D 其中一个键被按下
if (isWPressed || isSPressed || isAPressed || isDPressed) {
 // 如果视角模式不是 3，旋转左臂
 if (camera->getPerspectiveMode() != 3)
  lxlmanLeftArm->rotation[0] += lxlman->roateDir[2] * (isShiftPressed ? 15 : 9); // 根据是否按
下 Shift 键调整速度
 if (lxlmanLeftArm->rotation[0] >= 45 || lxlmanLeftArm->rotation[0] <= -45)
  lxlman->roateDir[2] = -lxlman->roateDir[2]; // 当旋转角度达到阈值时，改变旋转方向
......
 // 旋转右腿
 lxlmanRightLeg->rotation[0] += lxlman->roateDir[5] * (isShiftPressed ? 15 : 9);
 if (lxlmanRightLeg->rotation[0] >= 45 || lxlmanRightLeg->rotation[0] <= -45)
  lxlman->roateDir[5] = -lxlman->roateDir[5];
 // 获取角色的移动速度
 float currentSpeed = lxlman->speed;
 if (isShiftPressed) {
  currentSpeed *= 2; // 按下 Shift 键时加速
 }
......
 // 如果按下空格键，触发跳跃
 if (isSpacePressed) {
  if (!lxlman->jumping) {
   lxlman->jumping = true;
   lxlman->jumpingDirection = 1; // 启动跳跃
  }
 }
}
// 控制“达龙”的飞行位置（类似飞行控制）
if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
 dalongPos.z += dalongmoveSpeed;  // 向前飞行
}
if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS) {
 dalongPos.z -= dalongmoveSpeed;  // 向后飞行
```

```
}
if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
 dalongPos.x += dalongmoveSpeed;  // 向左飞行
}
if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS) {
 dalongPos.x -= dalongmoveSpeed;  // 向右飞行
}
```

这段代码是用来处理我的角色和“达龙”飞行控制的。首先，针对 W、S、A、D 键的按下，我控制了角色的移动和摆臂动作。当 W、S、A、D 中的任何一个键被按下时，我会根据当前是否按下 Shift 键来调整角色四肢（左臂、右臂、左腿、右腿）的旋转速度。如果旋转角度达到一定的阈值（比如 45 度），我会反转旋转的方向，确保角色的动作不至于一直单向旋转。

然后，角色的前进、后退、左移和右移都基于角色当前的朝向进行计算，角色的速度也会根据 Shift 键是否按下而变化，按下 Shift 键可以加速移动。

当我按下空格键时，角色会跳跃，并启动跳跃的方向，确保跳跃操作不会重复触发。

最后，我还实现了“达龙”的飞行控制。通过上下左右箭头键，我可以控制“达龙”的飞行方向和速度，类似于飞行控制器的操作。

```
//控制挥剑
void DoSword()
{
 //挥剑
 if (doSorw) {
  //右臂摆到最高点，则往下摆
  if (Upper >= maxUpper && Lower >= maxLower)
  {
   movestep = -movestep;
  }
  //往上摆时
  if (movestep > 0)
  {
   //上臂未到最高点
   if (Upper < maxUpper)
    Upper += movestep;
   else if (Lower < maxLower)
    Lower += movestep;
  }
  else {
   //下臂未到最低点
   if (Lower > 0)
    Lower += movestep;
   else if (Upper > 0)
    Upper += movestep;
  }
  //挥剑结束
  if (movestep < 0 && Upper <= 0 && Lower <= 0)
  {
   Upper = 0;
   Lower = 0;
```

```
   doSorw = false;
   movestep = -movestep;
  }
 }
}
```

这段代码是用来实现挥剑动画的控制。首先，我通过判断 doSorw 是否为 true 来决定是否执行挥剑动作。如果挥剑动作开始，我就会控制上臂和下臂的角度变化。在挥剑的过程中，我判断上臂和下臂的角度是否达到了最大或最小值，以此来决定是否改变挥剑的方向。具体来说，如果上臂和下臂都达到最大角度，我就会改变动作的方向，开始从上往下摆剑。反之，如果动作还没有到达最大角度，我会继续增加上臂或下臂的角度。在挥剑结束时，如果上臂和下臂都回到起始位置，我就会停止挥剑动作，并将 doSorw 设置为 false，准备开始下一次挥剑。

```
// 鼠标按键回调函数 - 用于处理鼠标按键事件
void mouseButtonCallback(GLFWwindow* window, int button, int action, int mods) {
 // 检查是否按下鼠标左键
 if (action == GLFW_PRESS) {
  if (button == GLFW_MOUSE_BUTTON_LEFT) {
   // 如果角色当前没有在挥手状态，则设置为挥手状态
   if (lxlman->waving == false) {
    lxlman->waving = true;  // 设置挥手状态为 true
    lxlman->wavingState = 0; // 初始化挥手状态
    // 如果不是特殊视角模式，则重置右臂的旋转
    if (camera->getPerspectiveMode() != 3) {
     lxlmanRightArm->rotation = glm::vec3(0.0, 0.0, 0.0);
    }
   }
  }
 }
```

在这段代码中，我实现了一个鼠标按键的回调函数，用于处理鼠标按键的事件。具体来说，这个函数主要是监听鼠标左键的按下事件，并根据事件的触发状态执行相关的操作。

首先，我判断是否按下了鼠标左键（GLFW_MOUSE_BUTTON_LEFT）。如果鼠标左键被按下，我接着检查角色是否处于挥手状态（lxlman->waving == false）。如果角色当前不在挥手状态，我会将其设置为挥手状态（lxlman->waving = true）并初始化挥手的状态（lxlman->wavingState = 0），这样角色就开始进入挥手的动作。

此外，我还检查了当前视角模式。如果视角模式不是特殊的视角模式（即不等于 3），我会将右臂的旋转角度重置为默认值（glm::vec3(0.0, 0.0, 0.0)），确保角色在挥手时，右臂从初始位置开始旋转。

```
// 跳跃
void update2() {
 // 检查是否处于跳跃状态
 if (lxlman->jumping) {
  // 根据跳跃方向更新角色的垂直位置（y 轴）
  lxlman->lxlmanPos[1] += lxlman->jumpingDirection * 0.1f;
  // 如果角色到达最高点或最低点，切换跳跃状态
```

```
 if (lxlman->lxlmanPos[1] >= 1.5f || lxlman->lxlmanPos[1] == 1.0f) {
  // 如果角色达到最高点，开始下落
  if (lxlman->jumpingDirection == 1 && lxlman->lxlmanPos[1] >= 1.5f) {
   lxlman->jumpingDirection = -1;  // 从上升状态切换到下降状态
  }
  // 如果角色已经到达最低点，跳跃结束
  else if (lxlman->jumpingDirection == -1 && lxlman->lxlmanPos[1] <= 1.0f) {
   lxlman->jumping = false;  // 跳跃结束，重置跳跃状态
  }
 }
}
}
```

在这段代码中，我实现了角色跳跃的逻辑。通过 update2 函数，我根据角色的跳跃状态（lxlman->jumping）控制角色在垂直方向（y 轴）的位置变化。

首先，我检查角色是否处于跳跃状态。如果是，角色的垂直位置会根据跳跃方向（lxlman->jumpingDirection）更新。跳跃方向的值为 1 时表示角色正在上升，值为 -1 时表示角色正在下降。在这里，我让角色的垂直位置按每次 0.1f 的步长进行更新。

然后，我判断角色的垂直位置是否已经达到跳跃的最高点或最低点。当角色的 y 轴位置大于等于 1.5f（最高点）时，我会让跳跃方向变为下降（lxlman->jumpingDirection = -1）。如果角色已经下降到 1.0f（最低点），跳跃就结束了，此时将 lxlman->jumping 设置为 false，表示跳跃完成，角色恢复到初始状态。

这 样，通过简单的逻辑判断，我模拟了角色的跳跃过程，并且确保角色在达到最高点后开始下落，最终回到地面，跳跃完成。这个控制逻辑能够让角色在游戏中表现出更加自然的跳跃动作。

```
//控制人物移动时手臂摆动
void walk() {
 if (Zmove || Xmove)
 {
  //第一次摆动手臂
  if (firstwalk)
  {
   robot.theta[robot.RightUpperArm] = 0;
   robot.theta[robot.RightUpperLeg] = 360;
   robot.theta[robot.LeftUpperArm] = 360;
   robot.theta[robot.LeftUpperLeg] = 0;
   firstwalk = false;
   type = 'R';
  }
  ......
   else if (!Up) {
    robot.theta[robot.RightUpperArm] -= movespeed;
    robot.theta[robot.LeftUpperArm] += movespeed;
    robot.theta[robot.RightUpperLeg] += movespeed;
    robot.theta[robot.LeftUpperLeg] -= movespeed;
   }
   //摆动到最高点
   if (robot.theta[robot.RightUpperArm] >= 20)
```

```
  {
   Up = false;
  }
 ......
  if (!Up && robot.theta[robot.LeftUpperArm] <= 0)
  {
   type = 'R';
   robot.theta[robot.RightUpperArm] = 0;
   robot.theta[robot.RightUpperLeg] = 360;
   robot.theta[robot.LeftUpperArm] = 360;
   robot.theta[robot.LeftUpperLeg] = 0;
   Up = true;
  }
 }
}
//停止行走
else {
 firstwalk = true;
 robot.theta[robot.RightUpperArm] = 0;
 robot.theta[robot.RightUpperLeg] = 0;
 robot.theta[robot.LeftUpperArm] = 0;
 robot.theta[robot.LeftUpperLeg] = 0;
}
}
```

在这段代码中，我负责控制机器人在移动时手臂的摆动。具体来说，当机器人开始移动时，手臂会按照一定的规律摆动，模拟自然的行走动作。

首先，当机器人第一次开始行走时，我初始化了四肢的角度：右上臂设置为 0 度，左上臂设置为 360 度，右上腿设置为 360 度，左上腿设置为 0 度。这样，机器人就准备好开始摆动了。

接下来，我通过判断 type 来决定是右臂还是左臂在前。当 type 为'R'时，右臂在前，我通过 Up 来控制手臂的摆动方向。当 Up 为 true 时，右臂向前摆动，左臂向后摆动；当 Up 为 false 时，右臂向后摆动，左臂向前摆动。通过这种方式，手臂的摆动会交替进行，模拟出自然的行走动作。

此外，我还设置了条件来控制手臂摆动的范围。当右臂摆到一定角度时，Up 会变为 false，这时机器人会开始摆动左臂，左臂也会按照类似的方式摆动。这些动作会交替进行，直到机器人停止移动为止。如果机器人停止移动，我会将所有的角度重置为初始值，准备下一次的手臂摆动。

6. 阴影绘制

```
if(mesh->getDrawShade()){
    // 绘制阴影
    glm::mat4 shadowMatrix = light->getShadowProjectionMatrix();  // 获取阴影矩阵
    // 转置阴影矩阵以适应 OpenGL 的列主序存储方式
    shadowMatrix = glm::transpose(shadowMatrix);
    // 将阴影矩阵应用于模型变换
    shadowMatrix = shadowMatrix * modelMatrix;
    // 传递矩阵到着色器
    glUniformMatrix4fv(object.modelLocation, 1, GL_FALSE, &shadowMatrix[0][0]); // 模型矩阵
    glUniformMatrix4fv(object.viewLocation, 1, GL_FALSE, &camera->viewMatrix[0][0]); // 视图矩阵
```

```
    glUniformMatrix4fv(object.projectionLocation, 1, GL_FALSE, &camera->projMatrix[0][0]); // 投影矩阵
    // 设置阴影标志位为1，表示绘制阴影
    glUniform1i(object.shadowLocation, 1);
    // **传递阴影亮度参数**，设置为 0.5 表示阴影亮度为环境光的 50%
    glUniform1f(glGetUniformLocation(object.program, "shadowBrightness"), 0.5f);
    // 绘制阴影
    glDrawArrays(GL_TRIANGLES, 0, mesh->getPoints().size()); // 绘制阴影的几何体
}
```

在这段代码中，我首先检查对象是否需要绘制阴影。如果需要，我会执行一系列步骤来计算并渲染阴影。首先，我通过 light->getShadowProjectionMatrix() 获取阴影投影矩阵，这个矩阵是用来映射光源投射的阴影的。由于 OpenGL 使用的是列主序存储矩阵，而返回的矩阵是行主序，我需要使用 glm::transpose() 来转置它，以确保正确的矩阵格式。

接下来，我将转置后的阴影矩阵与当前的模型变换矩阵相乘，得到最终的阴影矩阵，这个矩阵将影响物体的阴影渲染。然后，我通过 glUniformMatrix4fv 把模型矩阵、视图矩阵和投影矩阵传递给着色器，确保阴影能够正确映射到物体表面。

最后，我设置阴影的亮度参数，并通过 glDrawArrays 绘制阴影几何体。这些步骤共同保证了阴影能够被正确渲染到场景中，增强了视觉效果。

7. 主函数

```
int main(int argc, char** argv)
{
 // 初始化 GLFW 库，必须是应用程序调用的第一个 GLFW 函数
 glfwInit();
 // 配置GLFW 窗口的OpenGL 版本
 glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);  // 主版本设置为3
 glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);  // 次版本设置为3
 glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);  // 设置OpenGL 核心配置文件
 // 在MacOS 上需要额外配置，使OpenGL 兼容
#ifdef __APPLE__
 glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);  // 启用前向兼容
#endif
 //PlaySound("./music/music.wav", NULL, SND_FILENAME | SND_ASYNC);
 // 创建窗口，设置窗口大小和标题
 GLFWwindow* window = glfwCreateWindow(800, 800, u8"2022150130_林宪亮_期末大作业", NULL, NULL);
 if (window == NULL)
 {
  // 如果窗口创建失败，输出错误信息并终止程序
  std::cout << "Failed to create GLFW window" << std::endl;
  glfwTerminate();
  return -1;
 }
 glfwMakeContextCurrent(window);  // 设置当前窗口为上下文
 // 设置回调函数处理用户输入事件
 glfwSetKeyCallback(window, key_callback);
 glfwSetMouseButtonCallback(window, mouseButtonCallback);
 glfwSetCursorPosCallback(window, mouse_callback);  // 设置鼠标位置回调函数
 glfwSetScrollCallback(window, scroll_callback);  // 设置滚轮回调函数
```

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);  // 隐藏鼠标指针
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);  // 处理窗口尺寸变化
// 加载OpenGL 函数指针
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
 // 如果GLAD 加载失败，输出错误信息并终止程序
 std::cout << "Failed to initialize GLAD" << std::endl;
 return -1;
}
// 初始化网格、着色器和缓冲区等资源
init();
// 输出帮助信息，指导用户如何操作
printHelp();
// 启用深度测试，确保渲染正确的物体层次
glEnable(GL_DEPTH_TEST);
// 主循环，直到窗口关闭
while (!glfwWindowShouldClose(window))
{
 // 更新游戏状态，例如角色移动、动画更新等
 update(window);
 // 更新跳跃动画
 update2();
 // 更新挥手动画
 update3();
 // 渲染当前帧
 display();
 // 交换缓冲区，显示渲染结果
 glfwSwapBuffers(window);
 // 处理所有窗口事件，如按键、鼠标等
 glfwPollEvents();
}
// 清理数据，释放所有资源
cleanData();
return 0;
}
```

在这段代码中，我首先初始化了 GLFW 库，确保它可以正确运行。我通过 glfwInit() 函数完成了这个操作。接着，我配置了 OpenGL 的版本，要求使用 OpenGL 3.3，并指定了核心配置文件。这样，我可以确保只使用 OpenGL 的现代特性，不会包含过时的功能。

为了兼容 MacOS 系统，我使用了一个条件编译的宏来启用前向兼容性，这对于在 Mac 上使用 OpenGL 至关重要。然后，我创建了一个 800x800 像素的窗口，并为其设置了标题。万一窗口创建失败，我会输出错误信息并提前终止程序。

接下来，我通过 glfwMakeContextCurrent() 设置了当前的 OpenGL 上下文，确保接下来的 OpenGL 渲染操作能够与窗口相关联。接着，我设置了多个回调函数，用于处理键盘、鼠标、滚轮等输入事件。这些回调函数在程序运行时会被调用，确保我能够实时响应用户的操作。

在加载完 OpenGL 的函数指针之后，我调用了 init() 函数来初始化所有渲染所需的资源，如网格、着色器和缓冲区等。此外，我还调用了 printHelp() 函数显示帮助信息，让用户了解如何操作程序。在主循环中，我不断地更新游戏状态，如角色的移动、动画等，同时执行渲染操作。通过

glfwSwapBuffers() 刷新窗口，将当前帧的渲染结果显示出来。我还使用了 glfwPollEvents() 来处理用户的输入事件，确保交互体验流畅。

当窗口关闭时，我调用了 cleanData() 函数来释放资源，确保程序能够干净地退出。通过这样的流程，我能够确保程序在渲染时的高效性和稳定性。