深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇 二三 ～二〇 二四 学年度第 一 学期

课程编号 1500610003 课序号 03 课程名称 计算机图形学 主讲教师 周漾 评分

学 号 2021270184 姓名 曹婉楠 专业年级 计算机科学与技术(卓越班)

教师评语:

题目: 3d 小游戏逃出地下城

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								

3D 小游戏逃出地下城

在本次图形学大作业中，我选择制作一款 3D 小游戏，玩家需要控制一个小女孩找到钥匙打开大门，离开这个地下城。在冒险中，玩家会遇到很多怪物，玩家需要躲避或者攻击这些怪物，否则可能会受到伤害。此外，在地下城中还散落着许多宝石，玩家可以收集这些宝石，收集的宝石越多，最终得分就越高。

一、游戏展示

游戏最主要的组成部分如下：

1. 小女孩

小女孩是玩家操控的角色，玩家可以通过键盘来控制移动方向，通过鼠标移动视角方向。在战斗中，玩家可以来释放攻击对抗怪物。左上角显示玩家当前的血量，图 1 展示了该女孩的外表。

图 1 玩家控制的角色

2. 攻击的小球

玩家射击的小球外观如图 2 所示。

图 2 攻击的小球

3. 怪物

怪物总是想抓住小女孩，因此怪物总是朝着小女孩所在的位置移动。一旦怪物抓到了玩家，玩家就会受到伤害，于是玩家会以消耗自己生命为代价使该怪物消失；如果怪物被玩家攻击，怪物也

会消失。怪物如图 3 所示。

图 3 怪物

4. 小木屋

散落在地下城中的小木屋每隔一定时间就会产生新的怪物。如图 4 所示。小木屋的图片如图 4 所示。

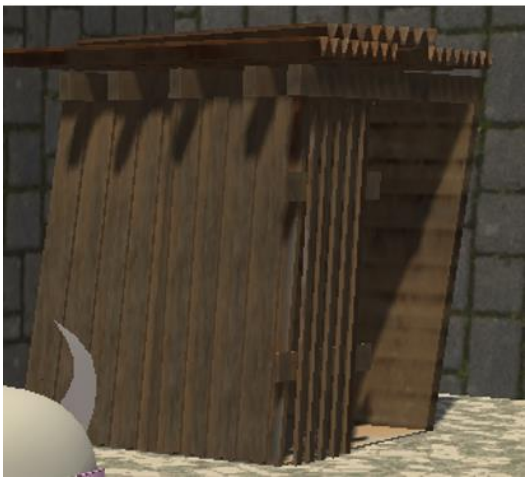


图 4 小木屋

5. 钥匙和宝石

地下城中有一把钥匙。玩家需要收集钥匙才能打开大门。并且地下城中有五颗宝石，收集到的宝石越多，玩家的分数就越高。但是，在这样的地下城中，收集更多的宝石意味着承受更多的风险。钥匙和宝石的照片如图 5 所示。

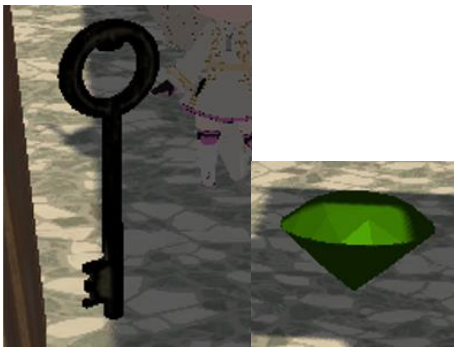


图 5 钥匙和宝石

6. 大门

游戏大门图片如图 6 所示。当玩家携带着钥匙，才能把大门打开，否则无法打开大门。

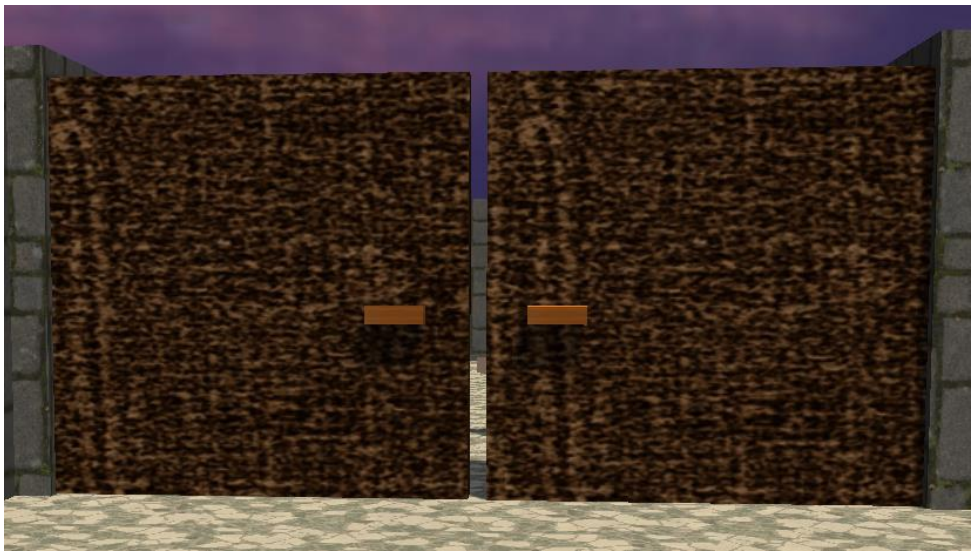


图 6 大门

7. 胜利和失败

当玩家成功携带钥匙逃出地下城时，游戏胜利，且屏幕上黄色星星的数量即为玩家收集到的宝石数量，如图 7 左图所示。当玩家不幸在地下城中生命值清 0，游戏结束，如图 7 右图所示。

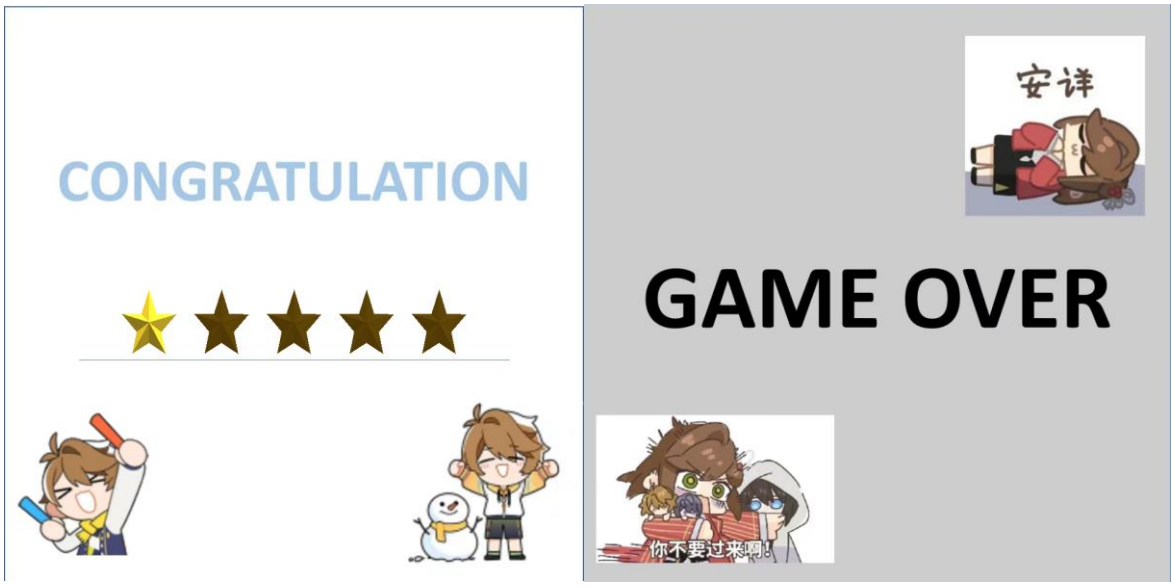


图 7 游戏胜利和失败

完整的游戏展示如上。下面来一步一步介绍其实现的步骤。

二、模型的导入与绘制

1. *Assimp*库的原理

在本次实验中，我学习了*LearnOpenGL*即 https://learnopengl-cn.github.io/ 上的模型导入章节，借助*Assimp*来实现模型的导入。*Assimp*很棒的一点在于，它抽象掉了加载不同文件格式的所有技术细节，只需要一行代码就能完成所有的工作。

$Assimp$能够导入多种不同的模型文件格式，包括obj格式和能保存骨骼动画的fbx格式，导入时会将所有的模型数据加载至$Assimp$的通用数据结构中。由于fbx格式的模型是有骨骼动画的，因此它与obj模型的导入有所区别，于是我们分别声明一个类$ffModel$用于导入fbx模型，$Model$类用于导入obj模型。

当使用$Assimp$导入一个模型的时候，它会将整个模型加载进一个场景$Scene$对象，它会包含导入的模型中的所有数据。$Assimp$会将场景载入为一系列的节点$Node$，每个节点包含了场景对象中所储存数据的索引，每个节点都可以有任意数量的子节点，组成了一个树形结构，如图 8 左半部分所示。$Assimp$会通过递归即dfs遍历所有节点，当我们想绘制某个节点时，$Assimp$会从文件中找到其对应的网格属性，包括顶点、法向量、纹理贴图、索引、材质等信息，然后将其保存下来，如图 8 右半部分所示。$Assimp$数据结构的简化模型如图 8 所示。

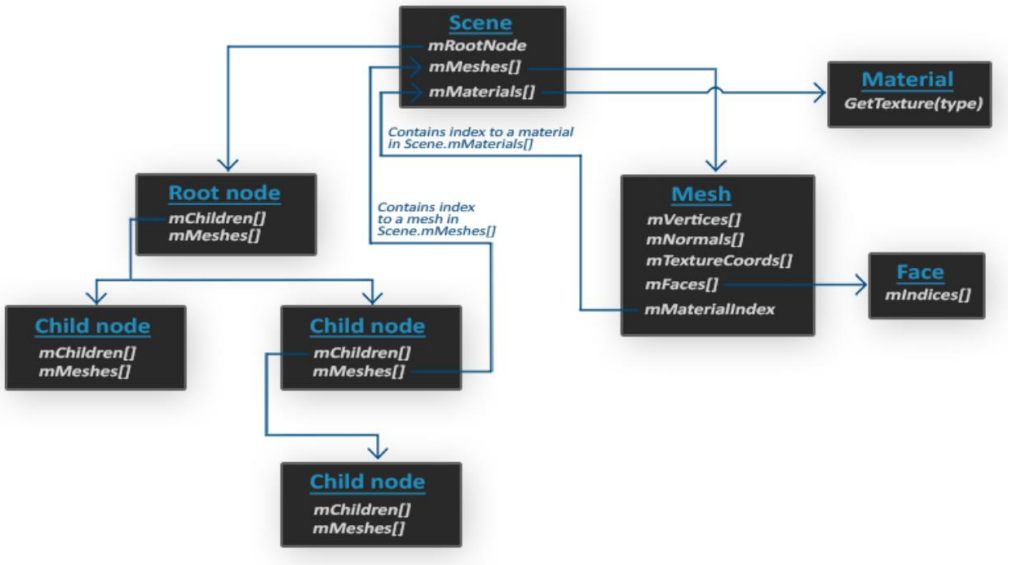


图 8 Assimp 简化模型

那么我们需要做的第一件事就是将物体加载到$Scene$对象中。我们首先声明一个$Assimp$命名空间下的$Importer$对象，并且调用它的$ReadFile$函数，这个函数的第一个参数是导入模型的路径；第二个参数是一些后期处理的选项，例如我们这里设定$aiProcess_Triangulate$，即告诉$Assimp$，如果模型不全由三角形组成，那么它需要将模型所有的图元形状变换为三角形；$aiProcess_FlipUVs$将翻转y轴的纹理坐标，因为在$OpenGL$中大部分图像的y轴是反的；$aiProcess_CalcTangentSpace$则是自动计算切线空间。

如果模型的路径不对或者模型的数据有问题、不完整，那么都会输出错误导入的提示信息。如果成功导入文件，我们就开始处理场景$Scene$下的每个$Node$节点。我们将第一个节点即根节点传入递归的$processNode$函数。如果在该节点下找到挂载的$Mesh$，就直接处理该$Mesh$数据并将其添加至$meshes$容器当中。代码如图 9 所示。

```
void Model::loadModel(std::string path){
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode){
        #ifdef Raven
            std::cout << "Assimp error when loading " << path << std::endl;
        #endif
        return;
    }
    processNode(scene->mRootNode, scene);
}
```

图 9 利用*Assimp*加载模型

*ffModel*的加载模型代码基本与*Model*一致，由于其还需要导入骨骼动画，骨骼动画存储在*aiScene*下的*aiAnimation*中，在我们加载*Scene*的同时就可以将模型动画同时加载进来，代码如图 10 所示。

```
for (unsigned int i = 0; i < scene->mNumAnimations; i++)    {
    aiAnimation* _aiAnimation = scene->mAnimations[i];
    ffAnimation* _animation = new ffAnimation(_aiAnimation, scene->mRootNode, m_boneInfoMap, m_boneCounter);
    ffAnimator* _animator = new ffAnimator(_animation);
    m_animators.push_back(_animator);
}
```

图 10 ffAssimp 模型还需要保存动画

接下来是递归函数*processNode*的书写，从*Scene*中获取根节点后，就可以从根节点开始，不断递归遍历其下的子节点，并且找到该节点下挂载的所有*Mesh*，直接处理*Mesh*数据并将其添加至*meshes*容器当中。最终做到处理完所有节点拥有的数据。代码如图 11 所示，*ffModel*该部分的代码与*Model*一致。

```
void Model::processNode(aiNode* node, const aiScene* scene){
    // process all the node's meshes (if any)
    for (unsigned int i = 0; i < node->mNumMeshes; i++){
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }
    for (unsigned int i = 0; i < node->mNumChildren; i++){
        processNode(node->mChildren[i], scene);
    }
}
```

图 11 遍历节点

在*processMesh*中，我们就可以利用*aiMesh*获取所有数据。我们首先加载出*aiMesh*中顶点属性的数据，包括顶点、法向量、纹理坐标，并将其保存到三个*vector*中，代码如图 12 所示。

```
Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene){
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    for (unsigned int i = 0; i < mesh->mNumVertices; i++){
        Vertex vertex;
        glm::vec3 vector;
        vector.x = mesh->mVertices[i].x; // 顶点
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;

        vector.x = mesh->mNormals[i].x; //法向量
        vector.y = mesh->mNormals[i].y;
        vector.z = mesh->mNormals[i].z;
        vertex.Normal = vector;

        if (mesh->mTextureCoords[0]){ //纹理坐标
            glm::vec2 vec;
            vec.x = mesh->mTextureCoords[0][i].x;
            vec.y = mesh->mTextureCoords[0][i].y;
            vertex.TexCoords = vec;
        }
        else{
            vertex.TexCoords = glm::vec2(0.0f, 0.0f);
        }
        vertices.push_back(vertex);
    }
```

图 12 处理网格的顶点属性

接下来需要处理网格的面索引和材质属性。每个网格都有一个*Face*数组，每个面代表了一个图元，由于我们在载入模型时使用了*aiProcess_Triangulate*选项(图 9 标红处)，因此模型的每一个面都是由三角形组成的。一个面包含了多个索引，我们遍历所有的面，并储存面的索引到*indices*这个*vector*中。

并且一个网格只包含了一个指向材质对象的索引，网格材质索引位于它的*mMaterialIndex*属性中。同样的，可以使用这个材质索引在*Scene*中查询到其对应的*aiMaterial*，里面存储了材质信息。*loadMaterialTextures*函数遍历了给定纹理类型的所有纹理位置，获取纹理的文件位置，并加载并和生成了纹理，将信息储存在了一个*Vertex*结构体中，详情可见代码附件。该段代码如图 13 所示。

```
for (unsigned int i = 0; i < mesh->mNumFaces; i++){ //索引
    aiFace face = mesh->mFaces[i];
    for (unsigned int j = 0; j < face.mNumIndices; j++){
        indices.push_back(face.mIndices[j]);
    }
}

if (mesh->mMaterialIndex >= 0){ //材质
    aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex];
    vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE, "texture_diffuse");
    textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
    vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR, "texture_specular");
    textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
}

return Mesh(vertices, indices, textures);
```

图 13 处理网格的索引、材质

$ffModel$中除了需要处理以上数据外，还需要从$aiNodeAnim$读取所有关键帧数据的单个骨骼，我们使用$loadBoneWeightForVertices$来完成，该函数还将根据当前动画时间在关键帧之间进行插值，即平移、缩放和旋转，详细代码可见附件。

```
loadBoneWeightForVertices(vertices, mesh, scene);

return ffMesh(vertices, indices, textures);
```

图 14 处理骨骼

至此，通过$Assimp$库导入模型的步骤已经完成。

2. 具体的绘制过程

如果我们想绘制导入的模型，调用$Model$下的$Draw$函数即可，该函数会遍历所有$Mesh$网格并进行绘制，代码如图 15 所示。

```
void Model::Draw(Shader * shader){
    for (unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}
```

图 15 绘制的$Draw$函数

其中以上我们所述的$Mesh$网格是我们声明的一个类，包含顶点、索引、纹理信息，每一个网格都绑定了其对应的顶点数组对象VAO、顶点缓冲对象VBO、顶点索引对象EBO，如图 16 所示。

```
class Mesh {
public:
    Mesh(float vertices[]);
    Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, std::vector<Texture> textures);
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;
    void Draw(Shader* shader);

private:
    unsigned int VAO, VBO, EBO;
    void setupMesh();
};
```

图 16 Mesh 类

在 Mesh 的构造函数中，我们会调用$setUpMesh$函数，该函数中我们先为VAO、VBO、EBO生成一个索引，并绑定该VAO，然后将顶点数据$vertices$与VBO绑定，将索引数据$indices$与EBO绑定。$glEnableVertexAttribArray$绑定顶点着色器中的位置属性，然后使用$glVertexAttribPointer$函数告诉$OpenGL$如何解析这些顶点数据，该函数第一个参数也是我们在顶点着色器中设置的位置属性，通常是 0 号代表顶点数据，1 号代表索引数据，2 号代表纹理数据；第二个参数指定属性大小，顶点属性和索引属性都是$vec3$，纹理属性为$vec2$；第三个参数指定数据类型，为GL_FLOAT；第四个参数是我们不需要正则化；第五个参数是偏移值。代码如图 17 所示。

```
void Mesh::setupMesh(){
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.size(), &vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &EBO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * indices.size(), &indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));

    glBindVertexArray(0);
}
```

图 17 链接顶点数据

最后就是$Mesh$下的$Draw$函数完成单个网格的绘制，该函数需要循环遍历与网格关联的纹理，并根据其类型(漫反射或镜面反射)设置纹理单元并绑定纹理。然后，在片段着色器中设置相应的$uniform$值，以指定漫反射和镜面反射纹理分别关联的纹理单元，为其贴上纹理，然后使用$glDrawElements$执行实际的绘制操作，参数选择三角形即可。最后解绑VAO，并将激活的纹理单元重置为默认值(GL_TEXTURE0)。代码及注释如图 18 所示。

```
void Mesh::Draw(Shader * shader){
    for (unsigned int i = 0; i < textures.size(); i++){
        if (textures[i].type == "texture_diffuse"){ // 在着色器中设置漫反射纹理的uniform值
            glActiveTexture(GL_TEXTURE0); // 激活纹理单元0并绑定纹理
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
            shader->SetUniform1i("material.diffuse", 0);
        }
        else if (textures[i].type == "texture_specular"){ // 在着色器中设置镜面反射纹理的uniform值
            glActiveTexture(GL_TEXTURE1); // 激活纹理单元1并绑定纹理
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
            shader->SetUniform1i("material.specular", 1);
        }
    }
    glBindVertexArray(VAO); // 绑定VAO
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0); // 使用三角形和指定的索引绘制网格
    glBindVertexArray(0); // 解绑顶点数组对象
    glActiveTexture(GL_TEXTURE0); // 重置激活的纹理单元为默认值GL_TEXTURE0
}
```

图 18 绘制网格并贴上纹理

至此，我们成功实现并封装了模型具体的绘制函数，绘制某一个模型直接调用$Model$下$Draw$即可，非常的便捷。

3. 小女孩

由于该游戏需要加载许多模型，且每个模型都有不同的参数，例如玩家操控的小女孩就还需要具有当前持有宝石数、钥匙数等变量，因此为了方便不同模型的管理，我们为每一个模型都建立一个类。对于小女孩我们建立一个$Character$类。其包含的成员变量如图 19 所示。包括$ffModel$，用于导入fbx模型。

```
class Character {
public:
    Character(Model _model, glm::vec3 _position, glm::vec3 _scale, float _speed);
    Character(ffModel _model, glm::vec3 _position, glm::vec3 _scale, float _speed);
    Model model;
    ffModel ffmodel;
    glm::vec3 initialDir = glm::vec3(0.0f, 0.0f, 1.0f);
    float rotate = 0;
    glm::vec3 position;  // 人物当前的位置
    glm::vec3 scale;     // 缩放比例
    glm::vec3 oriantation = glm::vec3(0.0f, 0.0f, 1.0f); // 人物朝向
    glm::vec3 forward = glm::vec3(0.0f, 0.0f, 1.0f);     // 人物向前的方向
    glm::vec3 right = glm::vec3(-1.0f, 0.0f, 0.0f);      // 人物的右方向
    glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);          // 人物的上方向
    glm::vec3 worldUp = glm::vec3(0.0f, 1.0f, 0.0f);     // 世界的上方向
    float radius = 1.0;   // 人物半径
    float speed;          // 人物速度
    unsigned int numKey = 0; // 当前钥匙的数量
    unsigned int numGem = 0; // 当前钻石的数量
    unsigned int numLife = 5;  //当前人物的生命值

    bool collision_x_p = false; // x正方向是否有碰撞
    bool collision_x_n = false; // x负方向是否有碰撞
    bool collision_z_p = false; // z正方向是否有碰撞
    bool collision_z_n = false; // z负方向是否有碰撞
    bool die = false;     // 角色是否死亡
    bool win = false;     // 角色是否胜利

    float senseForward = 0.15; // 向前移动率
    float senseRight = 0.15;   // 向右移动率
    //float senseUp = 0.15;    // 向上移动率，不需要
    float speedForward = 0;    // 其正负控制角色是向前走还是向后走
    float speedRight = 0;      // 其正负控制角色是向右走还是左后走
    //float speedUp = 0;       // 向上的速率，不需要
```

图 19 Character 类包含的成员变量

玩家操控的小女孩模型比较精细，具有层级建模的架构，该架构如图 20 所示。通过图 11 的 dfs代码，即遍历整个树完成整个模型的绘制，使用$Assimp$下的$aiNode$，会使整个过程更加清晰，而不是需要手动模拟整个过程。

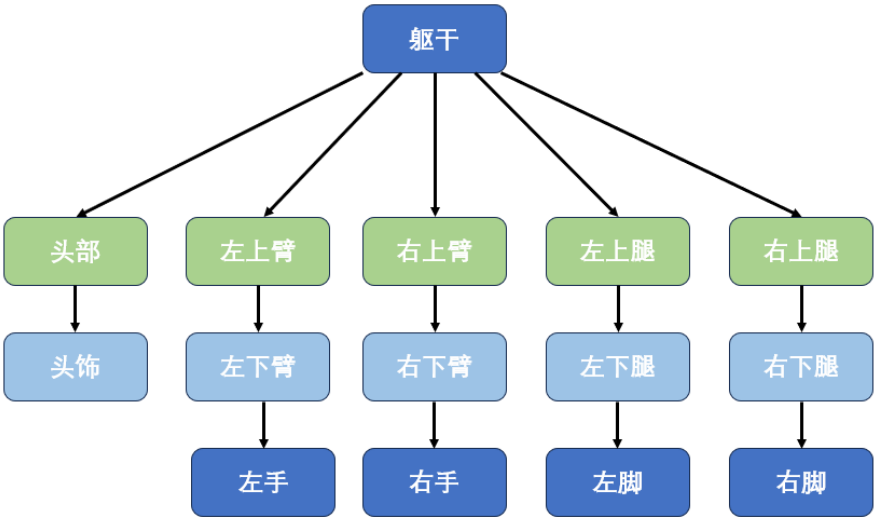


图 20 层级建模

接下来我们就来导入该 3d 模型，即在$main$函数中初始化一个$character$对象，其中包含的$ffModel$对象可以完成模型的导入，需要传入模型路径。并且我们需要人物初始化的位置(世界坐标)、缩放比例$scale$、人物速度，在构造函数中完成这些参数的初始化。如图 21 所示。

```
Character character(
    ffModel("asserts\\Lifu\\Lifu.fbx"),
    glm::vec3(-2.0f, 0.0f, 0.0f),      // position
    glm::vec3(5.0f, 5.0f, 5.0f),       // scale
    3                                  // speed
);
```

图 21 导入丽芙 fbx 模型

且通过 3d viewer 打开该模型，可以看到其包含 47 个动画，通过刚刚书写的导入函数即可将这些动画都导入到$m_animator$中，该模型如图 22 所示。

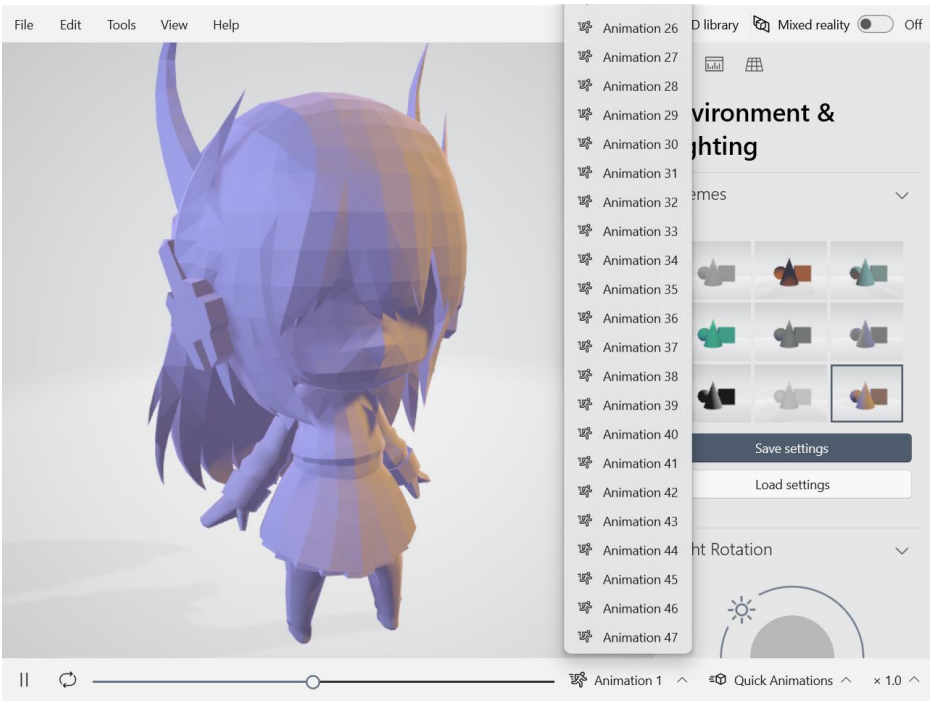


图 22 丽芙的 3d 模型包含多个动画

如果我们想播放人物的骨骼动画，可以调用$ffmodel$下的$update$来更新动画，第一个参数代表该动画播放的时间，第二个参数表示播放的动画编号，该编号可以在 3D viewer 中查询。如果当前角色在正在完成射击动作，则播放射击动画，如果人物正在移动，则播放走路动画，如果人物胜利，则播放一段胜利动画，否则人物静止。代码如图 23 所示。

```
if (shoot) {
    animationNumber = 22;  // 射击动画
    if (_currentTime - lastShootTime >= shootAnimationDuration) {
        shoot = false;
        character.ffmodel.m_animators[animationNumber]->playAnimation();
    }
}
else if (character.speedForward || character.speedRight) animationNumber = 38; // 走路
else if (character.win) animationNumber = 20; //胜利
else animationNumber = 0; // 站立不动
shader_shadowMap->SetMatrix("model", modelMat);
character.ffmodel.update(_deltaTime, animationNumber); // 更新动画
character.ffmodel.Draw(shader_shadowMap, animationNumber); // 绘制模型
```

图 23 更新动画且绘制模型

人物静止的骨骼动画如图 24 所示。

图 24 静止不动

人物走路的骨骼动画如图 25 所示。

图 25 走路

人物射击的骨骼动画如图 26 所示。

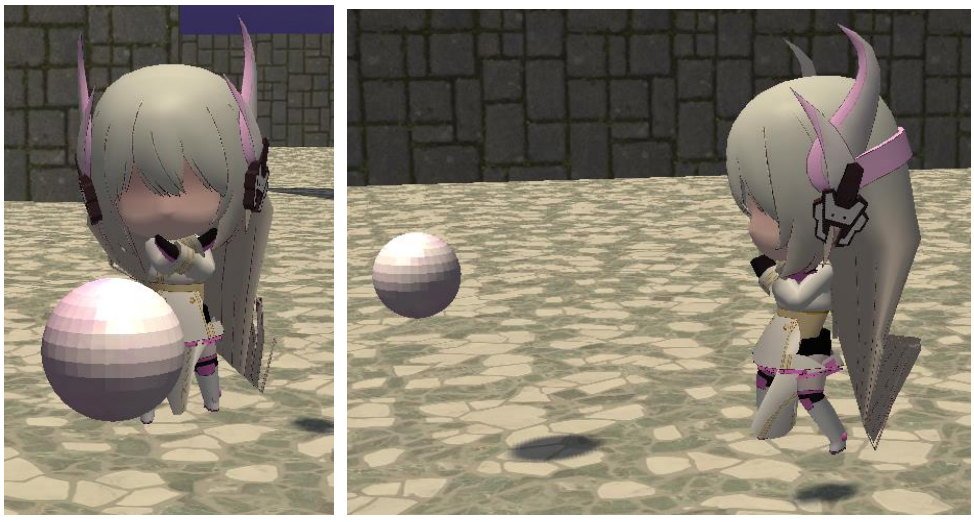


图 26 射击

人物胜利的骨骼动画如图 27 所示。

图 27 胜利

至此，我们完成了模型的导入、贴图的加载以及骨骼动画的播放。

4. 怪物

下面来导入怪物模型，怪物模型的参数和小女孩的参数差不多，因此也可以使用*Character*类。并且我们也为怪物模型添加一个简单的层级建模的结构，并且为了更加熟悉这个过程，我们通过将躯干、四肢都使用一些单独的*obj*文件进行导入，这样可以手动模拟层级建模的过程。如图 28 所示。

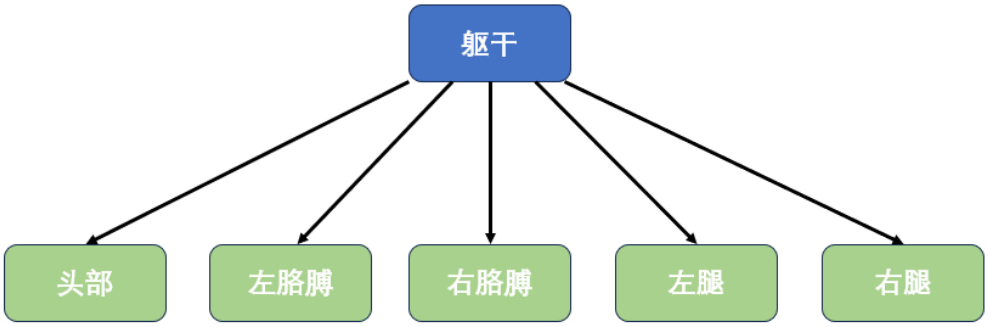


图 28 层级模型——怪物

接着我们使用*Model*类来导入怪物的*obj*模型，并且以怪物的躯干为中心，为其设定世界坐标、缩放大小以及速度大小的初始化，然后再导入怪物的四肢*obj*模型。代码如图 29 所示。

```
Model monsterBody_model("asserts\\squareman_seperate\\body\\body.obj");
vector<Character> monsters;
Character monster_0(
    monsterBody_model,
    glm::vec3(30.0f, 0.0f, 30.0f),     // position
    glm::vec3(1.5f, 1.5f, 1.5f),      // scale
    0.2                               // speed
);
monsters.push_back(monster_0);

Model monster_righthand("asserts\\squareman_seperate\\right_hand\\right_hand.obj");
Model monster_lefthand("asserts\\squareman_seperate\\left_hand\\left_hand.obj");
Model monster_rightleg("asserts\\squareman_seperate\\right_leg\\right_leg.obj");
Model monster_leftleg("asserts\\squareman_seperate\\left_leg\\left_leg.obj");
```

图 29 obj 模型

绘制怪物时，以左手和右手为例，我们需要创建一个变换矩阵，先通过怪物躯干的位置，对右

手进行平移和旋转，找到箱子人右手的所在位置，控制手臂的前后摇摆角度则通过时间来调整*time_rotate_monster*变量，该变量为[-1, 1]变化的值，由此控制摇摆角度。左手的控制逻辑与右手差不多，只是平移、旋转的方向略有不同，代码如图 30 所示。右脚和左脚的代码也基本一致，详细代码可见附件。

```
for (auto& monster : monsters) {
    shader_monster->use();
    trans = glm::scale(glm::mat4(1.0f), monster.scale);
    trans = glm::rotate(trans, monster.rotate, glm::vec3(0.0f, 1.0f, 0.0f)); //进行一定角度的旋转
    modelMat = glm::translate(glm::mat4(1.0f), monster.position);

    glm::mat4 trans_righthand = glm::translate(trans, glm::vec3(-0.15f, 1.5f, 0.0f)); //对右手进行旋转、平移找到箱子人右手的所在位置
    trans_righthand = glm::rotate(trans_righthand, glm::radians(22.5f * time_rotate_monster), glm::vec3(0.0f, 1.0f, 0.0f)); //进行一定角度的旋转
    trans_righthand = glm::translate(trans_righthand, glm::vec3(0.15f, -1.5f, 0.0f)); //进行一定角度的平移
    glUniformMatrix4fv(glGetUniformLocation(shader_monster->ID, "transform"), 1, GL_FALSE, glm::value_ptr(trans_righthand));
    monster_righthand.Draw(shader_monster);

    glm::mat4 trans_lefthand = glm::translate(trans, glm::vec3(0.15f, 1.5f, 0.0f)); //对左手进行旋转、平移找到箱子人左手的所在位置
    trans_lefthand = glm::rotate(trans_lefthand, glm::radians(22.5f * time_rotate_monster), glm::vec3(0.0f, 1.0f, 0.0f)); //进行一定角度的旋转
    trans_lefthand = glm::translate(trans_lefthand, glm::vec3(-0.15f, -1.5f, 0.0f)); //进行一定角度的平移
    glUniformMatrix4fv(glGetUniformLocation(shader_monster->ID, "transform"), 1, GL_FALSE, glm::value_ptr(trans_lefthand));
    monster_lefthand.Draw(shader_monster);
```

图 30 怪物手臂的旋转

至此我们完成了怪物模型的导入并且实现了一个简单的手臂、腿部的摇摆动作。

5. 小木屋、钻石、钥匙、球、大门

接下来，我们依次导入其他*obj*模型，包括小木屋、钻石、钥匙、球、大门等，并分别为其建立*Cabin*类，*Gem*类，*Key*类，*Ball*类，*Gate*类。

(1) 小木屋

*Cabin.cpp*包括位置、缩放大小以及旋转方向等参数，代码如图 31 所示。

```
class Cabin{
public:
    Cabin(Model _model, glm::vec3 _position, glm::vec3 _scale, float _rotation, glm::vec3 _rotation_dir) :
        model(_model),
        position(_position),
        scale(_scale),
        rotation(_rotation),
        rotation_dir(_rotation_dir){}


    Model model;
    glm::vec3 position; // 位置
    glm::vec3 scale; // 缩放大小
    float rotation; // 朝向
    glm::vec3 rotation_dir; // 旋转方向
};
```

图 31 Cabin 类

在 main 函数中，使用一个*vector*容器*cabins*存放所有的小木屋，导入模型以及初始化的代码如图 32 所示。

```
vector<Cabin> cabins;
Model cabin("asserts\\cabin\\cabin.obj");
Cabin newcabin = Cabin{
    cabin,
    glm::vec3(-29.0f, -0.3f, -27.0f),              // position
    //glm::vec3(0.00015f, 0.00035f, 0.00015f),      // scale
    glm::vec3(2.0f, 4.0f, 2.0f),
    glm::radians(90.0f),                          // rotate degree
    glm::vec3(0.0f, 1.0f, 0.0f)                   // rotate direction
};
cabins.push_back(newcabin);
```

图 32 cabins 的初始化

(2) 钻石

*Gem.cpp*包括位置、缩放大小以及半径参数，半径参数可以用来检测是否产生碰撞。代码如图33所示。

```
class Gem{
public:
    Gem(Model _model, glm::vec3 _position) :
        model(_model),
        position(_position)
    {}

    Model model;
    glm::vec3 position;
    glm::vec3 scale = glm::vec3(1.0f, 1.0f, 1.0f);
    float radius = 1.0;
};
```

图 33 Gem 类

在 main 函数中，使用一个*vector*容器*gems*存放所有的宝石，导入模型以及初始化的代码如图34所示。

```
vector<Gem> gems;
Model gem("asserts\\gem\\gem.obj");
Gem newGem = Gem(
    gem,
    glm::vec3(-36.0f, 1.0f, -71.0f)    // position
);
gems.push_back(newGem);
```

图 34 gems 的初始化

(3) 钥匙

*Key.cpp*与 Gem 一样，成员变量包括位置、缩放大小以及半径，代码如图 35 所示。

```
class Key{
public:
    Key(Model _model, glm::vec3 _position) :
        model(_model),
        position(_position){}

    Model model;
    glm::vec3 position;
    glm::vec3 scale = glm::vec3(3.0f, 3.0f, 3.0f);
    float radius = 1.0;
};
```

图 35 Key 类

在 main 函数中，使用一个*vector*容器*keys*存放钥匙，导入模型以及初始化的代码如图 36 所示。

```
vector<Key> keys;
Model key("asserts\\key\\key.obj");
Key newKey = Key(
    key,
    glm::vec3(35.0f, 2.0f, -72.0f)    // position
);
keys.push_back(newKey);
```

图 36 keys 的初始化

(4) 球

类似的，*Ball.cpp*也是包括位置、缩放大小、朝向以及旋转方向等参数，代码如图 37 所示。

```
class Ball{
public:
    Ball(Model _model, glm::vec3 _position, glm::vec3 _forward);

    Model model;
    glm::vec3 position;
    glm::vec3 forward;
    glm::vec3 worldUp = glm::vec3(0.0f, 1.0f, 0.0f);
    float radius = 0.5;
    float speed = 0.8;

    void updatePosition();
};
```

图 37 Ball 类

在 main 函数中，如果主角按下了攻击键，则会调用*ShootBall*函数，生成一个球体，也是将所有的球体都存到一个*vector*中，代码如图 38 所示。

```
void ShootBall(vector<Ball>& Attack_Ball, Model icePiton_model, glm::vec3 position, glm::vec3 forward){
    Ball Aball(
        icePiton_model,
        position,     // position
        forward       // forward direction
    );
    Attack_Ball.push_back(Aball);
}
```

图 38 ball 的初始化

(5) 大门

*Gate.cpp*包括位置、缩放大小以及x轴长度和z轴长度，作用类似于半径，可用于碰撞检测，代码如图 39 所示。

```
class Gate{
public:
    Gate(Model _model, glm::vec3 _position, glm::vec3 _scale) :
        model(_model),
        position(_position),
        scale(_scale){}

    Model model;
    glm::vec3 position;
    glm::vec3 scale;
    float x_length = 12.5;
    float z_length = 0.8;
};
```

图 39 Gate 类

在 main 函数中，使用一个*vector*容器*gates*存放所有的小木屋，导入模型以及初始化的代码如图 40 所示。

```
vector<Gate> gates;
Model gate("asserts\\gate\\gate.obj");
Gate newgate = Gate{
    gate,
    glm::vec3(52.5f, 0.0f, 39.0f),    // position
    glm::vec3(6.0f, 2.8f, 3.0f)       // scale
};
gates.push_back(newgate);
```

图 40 gate 的初始化

至此，我们完成了其余一些模型的导入。

6. 失败与胜利

当游戏胜利或者失败时，会显示一张铺满屏幕的图片，在加载图片之前，由于图片的uv坐标是

反着的，因此我们先讲y轴颠倒过来，然后调用$loadTextureImageToGPU$函数，加载图片并返回生成的纹理编号，这里特别注意jpg格式的图片是RGB三通道，png格式的图片是$RGBA$四通道，$textureSlot$是纹理的槽位。使用$glTexParameteri$设置纹理的环绕方式。代码如图 41 所示。

```
#pragma region Init Gameover Texture;
    std::string gameover_path = "asserts\\gameover\\gameover_flip.jpg";
    stbi_set_flip_vertically_on_load(true);
    unsigned int gameoverTex = LoadTextureImageToGPU(gameover_path.c_str() , GL_RGB, GL_RGB, 0);
#pragma endregion

#pragma region Init Congratulation Texture;
    std::string congratulation_path = "asserts\\congratulation\\congratulation_flip.jpg";

    stbi_set_flip_vertically_on_load(true);
    unsigned int congratulationTex = LoadTextureImageToGPU(congratulation_path.c_str(), GL_RGB, GL_RGB, 0);
#pragma endregion


#pragma region Texture Loading Function
unsigned int LoadTextureImageToGPU(const char* filename, GLint internalFormat, GLenum format, int textureSlot){
    unsigned int TexBuffer;
    glGenTextures(1, &TexBuffer);

    glActiveTexture(GL_TEXTURE0 + textureSlot);
    glBindTexture(GL_TEXTURE_2D, TexBuffer);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);   // set texture wrapping to GL_REPEAT (default
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    int width, height, nrChannel;
    unsigned char *data = stbi_load(filename, &width, &height, &nrChannel, 0);
    if (data)   {
        glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else{
    #ifdef Raven
        cout << "Texture image " << filename << " load failed." << endl;
    #endif
    }
    stbi_image_free(data);
    return TexBuffer;
}
#pragma endregion
```

图 41 导入失败/胜利纹理照片

绘制该图片时，不需要法向量坐标，位置坐标为屏幕的四个角，最后绘制$GL_TRIANGLE_STRIP$三角形即可。代码如图 42 所示。

```
#pragma region renderQuad
unsigned int quadVAO = 0;
unsigned int quadVBO;
void renderQuad(){
    if (quadVAO == 0){
        float quadVertices[] = {
            // positions        // texture Coords
            -1.0f,  1.0f, 0.0f, 0.0f, 1.0f,
            -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
             1.0f,  1.0f, 0.0f, 1.0f, 1.0f,
             1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
        };
        // setup plane VAO
        glGenVertexArrays(1, &quadVAO);
        glGenBuffers(1, &quadVBO);
        glBindVertexArray(quadVAO);
        glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 * sizeof(float)));
    }
    glBindVertexArray(quadVAO);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindVertexArray(0);
}
#pragma endregion
```

图 42 绘制失败/胜利纹理照片

7. 天空盒

在实时渲染中，如果要绘制非常远的物体，例如远处的山、天空等，随着观察者的距离的移动，

这个物体的大小是几乎没有什么变化的，这个时候可以考虑采用天空盒技术。所谓的天空盒其实就是将一个立方体展开，然后在六个面上贴上相应的贴图。如图 43 所示。

图 43 天空盒示意图

在实际的渲染中，将这个立方体始终罩在摄像机的周围，让摄像机始终处于这个立方体的中心位置，然后根据视线与立方体的交点的坐标，来确定究竟要在哪一个面上进行纹理采样。因此天空盒的图片需要是一张环绕的照片，否则有几个面拼不起来，就会出现割裂的效果，如图 44 所示。

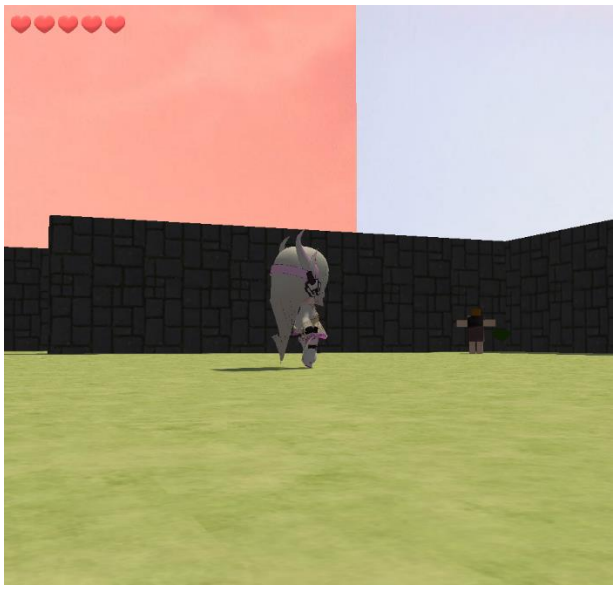


图 44 未使用环绕图

然后我们需要导入六张纹理照片，如图 45 所示。

```
#pragma region Init Skybox;
    vector<std::string> faces{
        "asserts\\skybox\\posx.jpg",
        "asserts\\skybox\\negx.jpg",
        "asserts\\skybox\\posy.jpg",
        "asserts\\skybox\\negy.jpg",
        "asserts\\skybox\\posz.jpg",
        "asserts\\skybox\\negz.jpg"
    };
    stbi_set_flip_vertically_on_load(false);
    unsigned int cubemapTexture = loadCubemap(faces);
#pragma endregion
```

图 45 导入天空盒的图片

绘制天空盒时，先通过$glDepthFunc(GL_LEQUAL)$设置深度测试的比较函数为小于等于，使其始终被后绘制的物体遮挡，从而呈现出正确的遮挡关系。然后就通过一系列矩阵变换来渲染天空盒。其中$viewMat$是观察矩阵，通过移除视图矩阵的平移部分，使天空盒不受相机位置的影响。绘制完毕后，通过$glDepthFunc(GL_LESS)$将深度测试的比较函数重新设置为小于，以恢复默认的深度测试条件，确保后续渲染操作使用正常的深度测试。代码如图 46 所示。

```
#pragma region Draw Skybox;
        glDepthFunc(GL_LEQUAL);
        shader_skybox->use();
        viewMat = glm::mat4(glm::mat3(camera.GetViewMatrix())); // remove translation from the view matrix
        modelMat = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -0.04f, 0.0f));
        modelMat = glm::rotate(modelMat, glm::radians(200.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        glUniformMatrix4fv(glGetUniformLocation(shader_skybox->ID, "modelMat"), 1, GL_FALSE, glm::value_ptr(modelMat));
        glUniformMatrix4fv(glGetUniformLocation(shader_skybox->ID, "viewMat"), 1, GL_FALSE, glm::value_ptr(viewMat));
        glUniformMatrix4fv(glGetUniformLocation(shader_skybox->ID, "projMat"), 1, GL_FALSE, glm::value_ptr(projMat));
        // skybox cube
        glBindVertexArray(skyboxVAO);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
        glDepthFunc(GL_LESS);
#pragma endregion
```

图 46 绘制天空盒

三、相机视角的切换

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。

图矩阵用于将场景中的物体坐标系变换到相机坐标系，以便进行后续的投影等操作，$lookAt$ 函数实现了视图矩阵的生成。视图矩阵通常由三个基本向量组成：右(x)轴，上(y)轴，和观察方向(z)轴。这三个向量构成了相机坐标系，如图 47 所示。

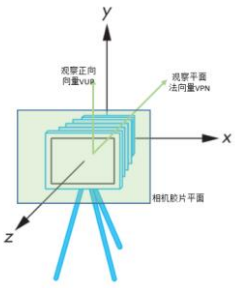


图 47 相机坐标系

我们再定义相机的位置，就可以创建一个视图矩阵了。不过glm中已经提供了$lookAt$函数，他需要三个参数，分别为：相机的位置、相机朝向的目标位置、世界空间的上方向，代码如图 48 所示。其中$cameraPos$为摄像机位置，方向是当前的位置加上刚刚的方向向量，第三个参数是世界上方向，用于计算相机的右方向，进而计算相机的上方向。如果想计算物体的$view\ \ Matrix$直接调用这个函数就可以了。

```
glm::mat4 Camera::GetViewMatrix(){
    return glm::lookAt(Position, Position + Forward, WorldUp);
}
```

图 48 视图矩阵 lookAt 代码

接着，我们只需要再写一个鼠标回调函数，用来调整相机的位置或者朝向，就可以实现相机的移动了。代码如图 49 所示。第一个回调函数式通过鼠标的x、y坐标的变化大小，从而计算相机需要变化的角度，从而改变相机所在的位置；第二个回调函数式通过移动鼠标滚轮，从而改变相机的fov参数，代表我们可以看到场景中多大的范围，这里把缩放级别限制在 3.0f 到 45.0f。

```
void mouse_callback_camera_ThirdPersonView(GLFWwindow* window, double xPos, double yPos){
    if (first_initialise_mouse == true) { // 第一次进入这个函数
        previous_xPos = xPos;
        previous_yPos = yPos;
        first_initialise_mouse = false;
        return;
    }

    double Delta_x = xPos - previous_xPos;
    double Delta_y = yPos - previous_yPos;

    previous_xPos = xPos;
    previous_yPos = yPos;

    //cout << Delta_x << "    " << Delta_y << endl;
    camera.ProcessMouseMovement_ThirdPersonView(Delta_x, Delta_y);
}

void scroll_callback_camera_ThirdPersonView(GLFWwindow* window, double xoffset, double yoffset){
    camera.fov -= (float)yoffset;
    if (camera.fov < 3.0f) camera.fov = 3.0f;
    if (camera.fov > 45.0f) camera.fov = 45.0f;
}
```

图 49 鼠标、滚轮回调函数

这样，我们就实现了相机随着鼠标的移动而移动，随着滚轮的放大缩小而放大缩小。但是我们由于人物的位置也会改变，而如果我们不把他们关联起来，可能就会产生问题，例如相机朝向和人物朝向不同，那么人物就会往其他方向走，甚至可能会离开摄像机的拍摄范围。因此，通过鼠标更新摄像机的朝向后，也要及时的更新人物的朝向；同时人物如果移动即位置改变后，也要根据人物的位置来决定照相机的位置；代码如图 50 所示，这里我设计了两个视角，一个是玩家的视角，另一个是顶端视角，顶端视角不同的地方在于，其位置是位于角色的正上方，然后朝下看的。

```
if (cameraMode == 1) {// 人物视角
    character.updateDirection_camera(camera.Forward, camera.Right);  //根据摄像机的朝向，更新人物的朝向
    camera.ThirdPersonView_LookAtCharacter(character); //同时人物移动后，也要根据人物的位置决定照相机的位置
}
else if (cameraMode == 2) {  // 顶端视角
    character.updateDirection_camera(camera.Forward, camera.Right);  //根据摄像机的朝向，更新人物的朝向
    camera.Position = character.position + glm::vec3(0.0f, 5.0f, 0.05f); //将照相机的位置置于角色上方
    camera.TopDownView_LookAtCharacter(character); //照相机朝向角色
}
```

图 50 更新相机的位置、朝向

而控制摄像机的两种模式，就可以通过键盘回调函数来解决，代码如图 51 所示。

```
if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS){  // Third person view
    cameraMode = 1;
    camera.Position = glm::vec3(0.0f, 5.0f, -50.0f);
}
if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS)  // Top down view
    cameraMode = 2;
```

图 51 键盘回调函数

这样，我们就实现了相机跟随人物视角移动以及相机视角的切换了。如果想将$view\ Matrix$变换到屏幕显示出来，可以直接调用glm下的$perspective$即透视变化函数，就不需要自己手写那个很复杂的矩阵了，代码如图 52 所示。该函数的第二个参数为屏幕的高宽比，第三个参数和第四个参数即为$near$和far参数，即近平面与远平面参数设置。

```
#pragma region Prepare MVP matrices
    glm::mat4 trans = glm::mat4(1.0f);
    glm::mat4 modelMat = glm::mat4(1.0f);
    glm::mat4 viewMat = glm::mat4(1.0f);
    glm::mat4 projMat;
    projMat = glm::perspective(glm::radians(45.0f), (float)window_height / (float)window_weight, 0.1f, 300.0f);
#pragma endregion
```

图 52 mvp 级联矩阵

四、光照与阴影

本次实验中，我们不使用$Phong$模型，而是使用$shadow\ map$阴影映射技术，基本原理如下：首先，这个游戏中的所有物体都会从光源位置的角度渲染一次。在此渲染中，深度信息将被记录下来，作为深度贴图。然后，所有物体将从摄像机位置的视点重新渲染。这一次，每个点的深度将被计算。如果深度大于刚刚在深度贴图中记录的相应深度，则该片段处于阴影中，那么渲染在阴影中的片段时，只使用环境光即$ambient$。

1. 生成深度贴图

第一步我们需要生成一张深度贴图。深度贴图是从光的透视图里渲染的深度纹理，可以用它来计算阴影。因为我们需要将场景的渲染结果储存到一个纹理中，所以需要用到帧缓冲。代码如图 53 所示。首先，我们要为渲染的深度贴图创建一个帧缓冲对象，对应代码的$glGenFrameBuffers$部分；其次创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用，并设置环绕方式即一些纹理参数；最后将深度贴图附加到帧缓冲对象即可。代码如图 53 所示。

```
#pragma region Init FBO for Shadow Map;
    unsigned int depthMapFBO; //创建帧缓冲对象
    glGenFramebuffers(1, &depthMapFBO);

    const unsigned int SHADOW_WIDTH = 2*4096, SHADOW_HEIGHT = 2*4096; //创建深度贴图纹理
    unsigned int depthMap;
    glGenTextures(1, &depthMap);
    glBindTexture(GL_TEXTURE_2D, depthMap);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); // 设置深度贴图的纹理参数
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);


    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0); // 将深度贴图附加到帧缓冲对象
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);

#pragma endregion
```

图 53 生成深度贴图

2. 光源空间的变换

因为我们使用的是一个所有光线都平行的定向光。因此我们为光源设定$ortho$正交投影矩阵，将场景投影到一个平行光的视图空间中，投影矩阵的参数指定了平行光视锥体的六个面的位置。接着再创建一个观察矩阵，将光源的位置设置为相对于角色位置的偏移量，并指定观察方向以及上向量。这样可以模拟光源随着角色的移动而改变位置。最后将投影矩阵和观察矩阵相乘，得到光空间矩阵，我们再把它传给顶点着色器的$uniform$变量，代码如图 54 所示。

```
float near_plane = 0.0f, far_plane = 200.5f;
glm::mat4 lightProjection = glm::ortho(-150.0f, 150.0f, -150.0f, 150.0f, near_plane, far_plane);
// 创建一个观察矩阵，将光源的位置设置为相对于角色位置的偏移量，并指定观察方向以及上向量。这样可以模拟光源随着角色的移动而改变位置。
glm::mat4 lightView = glm::lookAt(lightD.position + character.position,
    character.position + glm::vec3(0.0f, 0.0f, 100.0f),
    glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
shader_shadowMap->use();
shader_shadowMap->SetMatrix("lightSpaceMatrix", lightSpaceMatrix);
```

图 54 光源空间的变换

3. 渲染阴影

正确地生成深度贴图以后我们就可以开始生成阴影了。我们首先在顶点着色器中进行从图 54 中传来的光空间矩阵的变换，代码如图 55 所示。

```
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;
layout(location = 3) in vec4 inBoneWeights;
layout(location = 4) in ivec4 inBoneIds;

const int MAX_BONES = 250;
const int MAX_BONE_INFLUENCE = 4;
uniform mat4 _finalBoneMatrices[MAX_BONES];
uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main(){
    vec4 _totalPos = vec4(0.0f);

    for (int i = 0; i < MAX_BONE_INFLUENCE; i++){
        if (inBoneIds[i] < 0){
            continue;
        }
        if(inBoneIds[i] >= MAX_BONES){
            _totalPos = vec4(aPos, 1.0);
            break;
        }

        _totalPos += inBoneWeights[i] * _finalBoneMatrices[inBoneIds[i]] *  vec4(aPos, 1.0);
    }
    gl_Position = lightSpaceMatrix * model * _totalPos;
}
```

图 55 光源空间的变换

片段着色器可以什么都不用实现，代码如图 56 所示。

```
#version 330 core

void main(){
    gl_FragDepth = gl_FragCoord.z;
    //gl_FragDepth += gl_FrontFacing ? 0.01 : 0.0;
}
```

图 56 片段着色器

绘制时，我们将最终的模型变换矩阵传递给阴影映射着色器，然后使用指定的着色器程序进行模型的渲染，以绘制小木屋的阴影为例，代码如图 57 所示。其他物体的绘制阴影过程也是一模一样的，其中$Draw$函数我们已经在图 15 中实现了。

```
trans = glm::scale(glm::mat4(1.0f), cabins[i].scale);
trans = glm::rotate(trans, cabins[i].rotation, cabins[i].rotation_dir);
modelMat = glm::translate(glm::mat4(1.0f), cabins[i].position);
modelMat = modelMat * trans;
shader_shadowMap->SetMatrix("model", modelMat);
cabin.Draw(shader_shadowMap);
```

图 57 绘制阴影

最终实现的效果如图 58 所示。

图 58 阴影效果

五、碰撞检测

在本次大作业中，我实现了角色与钻石、钥匙、怪物、大门、墙，以及怪物与攻击的小球之间的碰撞检测、其中角色与钻石、钥匙、怪物，以及怪物与小球的碰撞原理都差不多，拿角色与钻石碰撞为例，我们首先需要获取角色和宝石的位置，然后通过$length$计算他们之间的距离，如果该距离小于两个物体的半径之和，说明产生了碰撞，角色当前钻石数+1。这里的坐标都是物体中心的坐标，代码及注释如图 59 所示。

```
#pragma region Chacter-Gem Collision
    for (int j = 0; j < gems.size(); j++) {
        // 获取角色和宝石的位置信息
        glm::vec3 pos_character = glm::vec3(character.position.x, 0.0f, character.position.z);
        glm::vec3 pos_gem = glm::vec3(gems[j].position.x, 0.0, gems[j].position.z);
        // 计算角色和宝石之间的距离
        float distance = glm::length(pos_character - pos_gem);
        // 如果距离小于两个物体的半径之和，说明发生了碰撞
        if (distance <= (character.radius + gems[j].radius)) {
            gems.erase(gems.begin() + j);  //vector中erase掉这个钻石
            character.numGem += 1; //钻石数+1
        }
    }
#pragma endregion
```

图 59 角色与钻石的碰撞

角色与墙或者大门碰撞检测时，我区分了x和z这两个分量，如果不区分的话，那么角色与墙一碰撞角色就动不了了。碰撞检测的原理为：看看大门和角色x轴、z轴之间的距离是否小于一个阈值，

如果小于则碰撞。$collision_x_n$角色的x分量有碰撞，其余变量含义也差不多。代码如图 60 所示。

```
#pragma region Chacter-Gate Collision

        for (unsigned int i = 0; i < gates.size(); i++) {
            bool gate_collision = false;

            if (abs(character.position.x - gates[i].position.x) <= gates[i].x_length + 0.5
                && abs(character.position.z - gates[i].position.z) <= gates[i].z_length + 0.5)
                gate_collision = true; //与大门碰撞

            if (gate_collision) {
                if (character.numKey) { //如果角色有钥匙
                    gates.erase(gates.begin() + i);
                    character.win = true; //设置为胜利
                    congratulationTime = glfwGetTime(); //一个时间间隔，可以加载胜利的动画
                    break;
                }

                if (character.position.x < (gates[i].position.x - gates[i].x_length + 0.5 + 0.5))
                    character.collision_x_n = true;
                else if (character.position.x > (gates[i].position.x + gates[i].x_length - 0.5 - 0.5))
                    character.collision_x_p = true;

                if (character.position.z < (gates[i].position.z - gates[i].z_length + 0.5 + 0.5))
                    character.collision_z_n = true;
                else if (character.position.z > (gates[i].position.z + gates[i].z_length - 0.5 - 0.5))
                    character.collision_z_p = true;
            }
        }

#pragma endregion
```

图 60 角色与大门的碰撞

效果如图 61 所示。

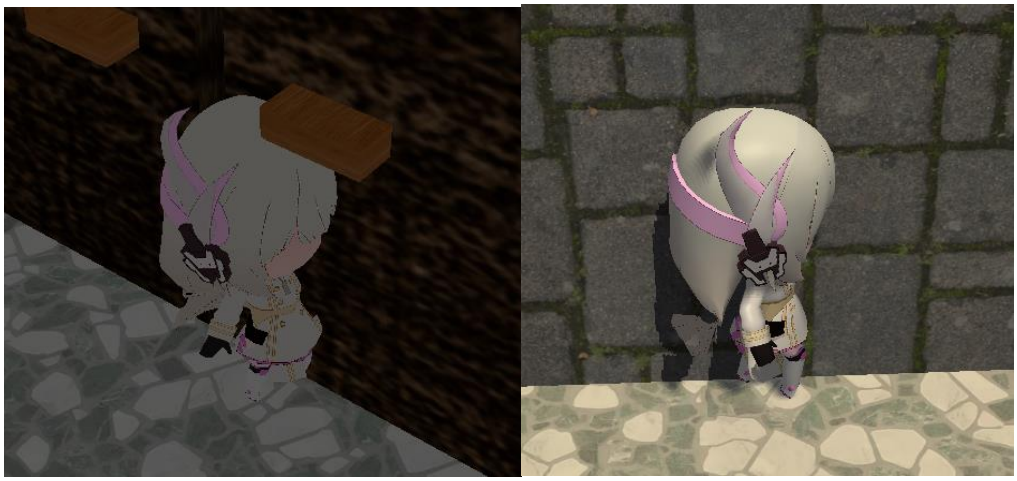


图 61 角色与大门、墙的碰撞效果

由于墙、大门他们是立方体，因此其x、z分量的值是一个特定的值，但是如果检测obj与obj之间的碰撞就非常麻烦了，例如角色与小木屋之间的碰撞检测就非常难实现。

六、物体的移动

主要包括人物、怪物和小球的运动。

1. 人物的移动

我们通过$character$对象的成员变量$speedForward$和$speedRight$的正负，决定角色是往前/后走，还是右/左走，通过键盘回调函数进行控制，代码如图 62 所示。

```
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS){
    character.speedForward = 1;
}
else if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS){
    character.speedForward = -1;
}
else character.speedForward = 0;

if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS){
    character.speedRight = 1;
}
else if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS){
    character.speedRight = -1;
}
else character.speedRight = 0;
```

图 62 控制运动方向

然后调用*updatePosition*函数，*displacement*作为偏移量，可以来更新角色的位置，其中*sense*是变化率，*speedForward*其来控制正负，*forward*即角色向前的朝向，*speed*是角色的速率。这里我添加了一个碰撞检测，如果角色的x方向分量或者z方向分量产生碰撞的话(通过第五部分的碰撞检测来判断)，就将*displacement*的这个方向设为 0，代码如图 63 所示。

```
glm::vec3 Character::updatePosition_key(){
    glm::vec3 displacement = senseForward * speedForward * forward * speed + senseRight * speedRight * right * speed;

    // if collide with walls
    if (collision_x_p && displacement.x < 0)
        displacement.x = 0;
    if (collision_x_n && displacement.x > 0)
        displacement.x = 0;
    if (collision_z_p && displacement.z < 0)
        displacement.z = 0;
    if (collision_z_n && displacement.z > 0)
        displacement.z = 0;

    position += displacement;

    glm::mat4 trans = glm::mat4(1.0f);
    trans = glm::rotate(trans, rotate, glm::vec3(0.0f, 1.0f, 0.0f));
    oriantation =  trans * glm::vec4(initialDir, 1.0f);

    return displacement;
}
```

图 63 角色的移动

2. 怪物的移动

由于设定怪物是自动朝着玩家所在位置走的，我们先对两个变量作差，就能得到怪物当前的朝向，然后再更新其朝向即可，代码如图 64 所示。

```
monster.updateDirection_follow(glm::normalize(character.position - monster.position));

void Character::updateDirection_follow(glm::vec3 followDirection){
    forward = glm::normalize(followDirection);
    right = glm::normalize(cross(forward, worldUp));
}
```

图 64 更新怪物的朝向

怪物的移动代码跟图 63 差不多，因为怪物已经朝向了玩家，因此只需控制*monster*往他的朝向走即可，并且也会检测该方向上是否有碰撞，有的话就将该分量置为 0 代码如图 65 所示。

```
monster.updatePosition_follow(character.position);
```

```
glm::vec3 Character::updatePosition_follow(glm::vec3 destination){
    glm::vec3 displacement;

    if (glm::length(destination - position) >= 0.3) {
        displacement = senseForward * forward * speed; //位移的方向

        if (collision_x_p && displacement.x < 0) //如果有碰撞，就将这个分量设为0
            displacement.x = 0;
        if (collision_x_n && displacement.x > 0)
            displacement.x = 0;
        if (collision_z_p && displacement.z < 0)
            displacement.z = 0;
        if (collision_z_n && displacement.z > 0)
            displacement.z = 0;

        position += displacement; //更新角色位置
    }
    else{
        displacement = glm::vec3(0.0f);
    }

    return displacement;
}
```

图 62 怪物的移动

3. 小球的移动

我们调用*ShootBall*函数(图 38)生成一个小球时，其位置和朝向和角色的位置、朝向一致，那么我们只需要控制小球往该朝向以一定速率进行移动即可，代码如图 63 所示。

```
void Ball::updatePosition(){
    position += forward * speed;
}
```

图 63 小球的移动

至此，整个大作业的主要代码已经实现。

七、补充

1. 游戏操作说明

其中整个大作业实现的交互键如图 64 所示。

```
#pragma region Help
void print_help() {
    std::cout << "============================================" << std::endl;
    std::cout << "Welcome to this world full of monsters!\n" << std::endl <<
                 "As a little girl, you need to find the key to open the door, \n" << std::endl <<
                 "At the meanwhile, you need to escape from this place," << std::endl <<
                 "and please avoid the attack of monsters, of course, you can also attack monsters!" << std::endl <<
                 "But when your blood runs out..." << std::endl;
    std::cout << "============================================" << std::endl << std::endl;

    std::cout << "Guidance" << std::endl;
    std::cout <<
        "[Window]" << std::endl <<
        "ESC:       Exit" << std::endl <<
        "h:         Print help message" << std::endl <<
        "[Keyboard]" << std::endl <<
        "w:         go ahead" << std::endl <<
        "a:         turn left" << std::endl <<
        "s:         go backward" << std::endl <<
        "d:         turn right" << std::endl <<
        "SPACE:     attack" << std::endl <<
        "[Mouse]" << std::endl <<
        "mouse:             controll the camera direction" << std::endl <<
        "mouseScroll_up:    enlarge view" << std::endl <<
        "mouseScroll_down:  reduce view" << std::endl <<
        "n:       hidden the cursor" << std::endl <<
        "m:       display the cursor" << std::endl <<
        "[Camera]" << std::endl <<
        "1:     ThirdPersonView" << std::endl <<
        "2:     TopDownView" << std::endl << std::endl;
}
#pragma endregion
```

图 64 游戏帮助

2. 代码运行说明

由于该大作业配置了$assimp$库，而起初该课程提供的$vcpkg$中$installed\x64-windows\include$是不包含$assimp$库的。这里我提供了一份新的$vcpkg$文件，其包含了运行我代码所需要的库，如果需要运行代码，需先打开cmd，并在$vcpkg.exe$所在的路径下执行.$\vcpkg\ integrate\ install$命令，如图 65 所示。出现绿色的那行代表配置成功。

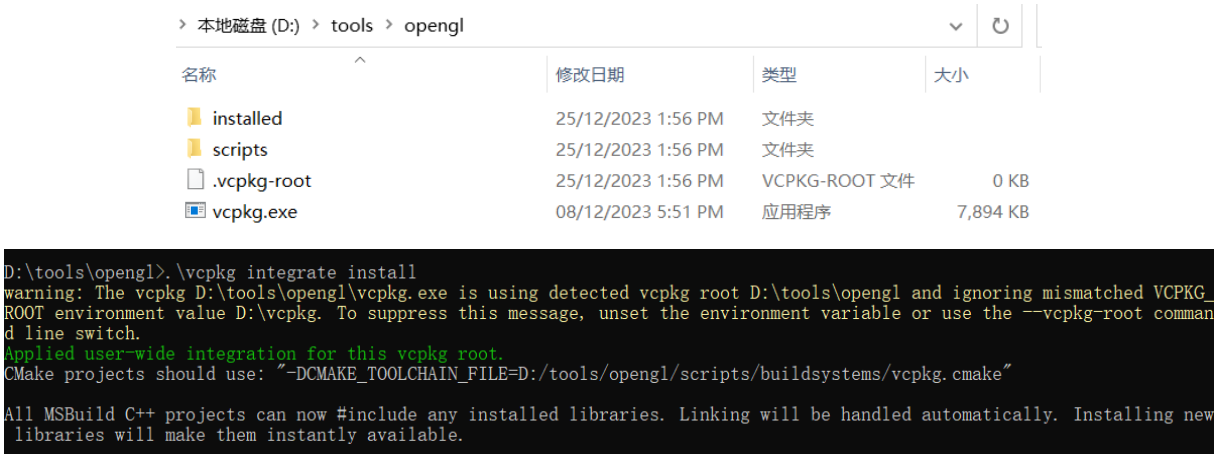
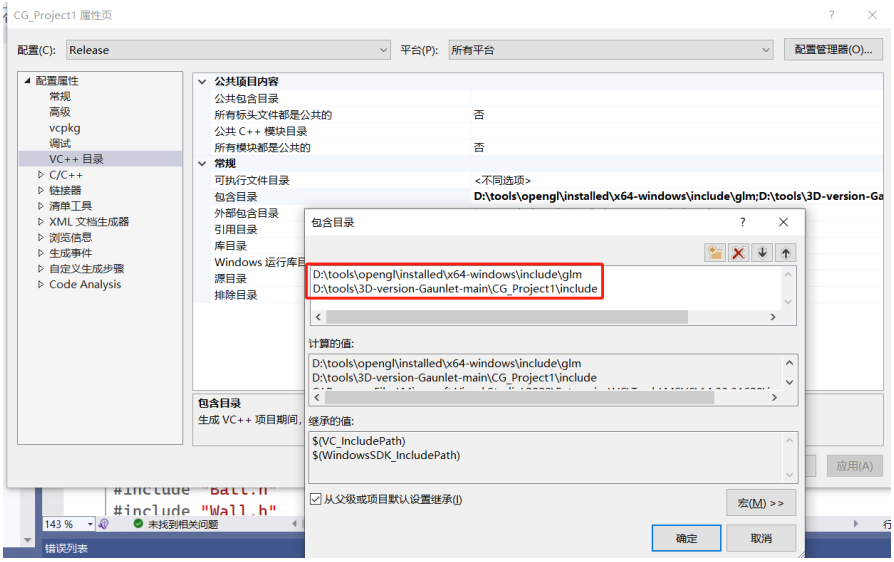
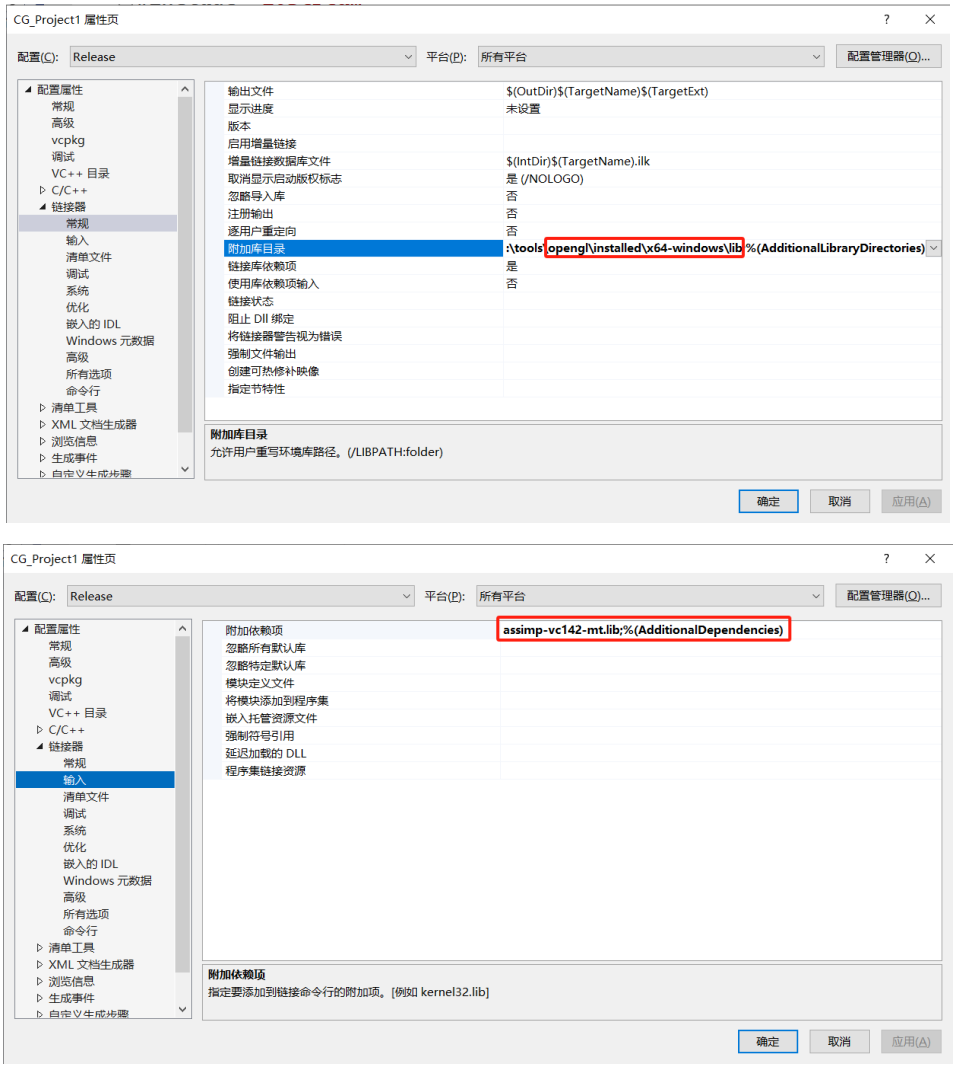


图 65 关于环境配置(1)

此时右键$CG_Project1$(项目名称)，打开最下面的属性，将$VC++$的包含目录添加$..\opengl\installed\x64-windows\include\glm$和..\项目路径$\include$，链接器的附加库目录添加.$.\opengl\installed\x64-windows\lib$，最后在链接器的输入中添加$assimp-vc142-mt.lib$即可，如图 66 所示。最后方案配置可能需要使用$Release$。




CG_Project1 属性页
配置(C): Release
平台(P): 所有平台
配置管理器(O)...
配置属性
常规
高级
vcpkg
调试
VC++ 目录
C/C++
链接器
常规
输入
清单文件
调试
系统
优化
嵌入的 IDL
Windows 元数据
高级
所有选项
命令行
清单工具
XML 文档生成器
浏览信息
生成事件
输出文件 $(OutDir)$(TargetName)$(TargetExt)
显示进度 未设置
版本
启用增量链接
增量链接数据库文件 $(IntDir)$(TargetName).ilk
取消显示启动版权标志 是 (/NOLOGO)
忽略导入库 否
注册输出 否
逐用户重定向 否
附加库目录 :\tools\opengl\installed\x64-windows\lib;%(AdditionalLibraryDirectories)
链接库依赖项 是
使用库依赖项输入 否
链接状态
阻止 DII 绑定
将链接器警告视为错误
强制文件输出
创建可热修补映像
指定节特性
附加库目录
允许用户重写环境库路径。(/LIBPATH:folder)
确定
取消
应用(A)

CG_Project1 属性页
配置(C): Release
平台(P): 所有平台
配置管理器(O)...
配置属性
常规
高级
vcpkg
调试
VC++ 目录
C/C++
链接器
常规
输入
清单文件
调试
系统
优化
嵌入的 IDL
Windows 元数据
高级
所有选项
命令行
清单工具
XML 文档生成器
浏览信息
生成事件
附加依赖项 assimp-vc142-mt.lib;%(AdditionalDependencies)
忽略所有默认库
忽略特定默认库
模块定义文件
将模块添加到程序集
嵌入托管资源文件
强制符号引用
延迟加载的 DLL
程序集链接资源
附加依赖项
指定要添加到链接命令行的附加项。[例如 kernel32.lib]
确定
取消
应用(A)

图 66 关于环境配置(2)