

# Final Report of Computer Graphics

---

第17小组

组员：陈柏林、黄雍明、李天扬、吕星龙

## 项目链接

---

我们的项目在Github上已经完成开源开发，可参考[链接](#)。

## 实验内容

---

### 小组选题

使用OpenGL搭建并渲染一个MC城堡的场景。

### 实现内容

- **模型文件加载**: 使用Assimp库加载静态模型文件。
- **动画模型加载**: 使用Assimp库加载骨骼动画模型文件。
- **实例化**: 使用实例化渲染大片重复内容。
- **天空盒背景**: 使用立方体贴图实现天空盒背景。
- **Blinn-Phong光照**: 实现Blinn-Phong光照明模型。
- **交互摄像机**: 通过键盘输入可以实现照相机的移动，鼠标滚轮实现拉近效果。

### 实验环境

- **操作系统**: Windows 11
- **开发工具**: Visual Studio 2022
- **编程语言**: C++17及以上
- **OpenGL版本**: OpenGL 3.3及以上
- **依赖库**:
  - **GLFW**: 用于创建窗口和管理OpenGL上下文。
  - **glad**: OpenGL扩展加载库。
  - **glm**: 用于矩阵、向量、变换等数学计算的库。
  - **assimp**: 用于加载静态和动画模型文件。
  - **stb\_image**: 用于加载纹理图片。

## 编译assimp源码

---

Assimp (全称Open Asset Import Library) 是一个开源的模型导入库，用于导入和处理3D模型文件，支持多种3D模型文件格式。Assimp提供了丰富的功能，包括场景图解析、网格处理、材质和纹理加载等，广泛应用于游戏开发、可视化和其他需要加载3D模型的领域。

本次课程项目，我们在底层使用Assimp库来导入模型到场景中。

首先在github找到assimp对应版本。本次项目使用的版本为 `assimp-5.4.2`，链接[点击此处](#)。在该网页的最下端，可以下载到源码：

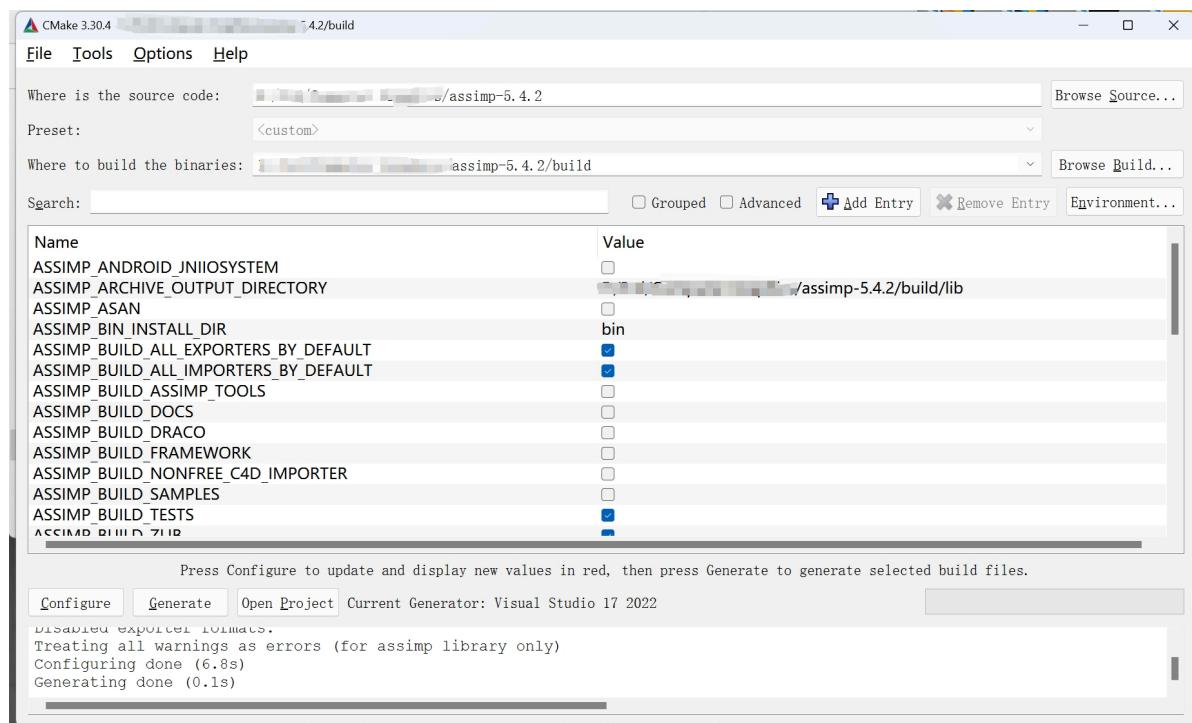


The screenshot shows a GitHub repository page for 'Assets'. There are two download links: 'Source code (zip)' and 'Source code (tar.gz)'. Both links were uploaded on Jul 4.

解压安装到任意目录，打开对应的 `assimp-5.4.2` 的文件夹，在该目录下创建build文件夹，用于存放 `cmake` 的生成和配置内容：

名称	修改日期	类型	大小
build	2024/11/27 14:29	文件夹	
cmake-modules	2024/07/04 3:37	文件夹	
code	2024/07/04 3:37	文件夹	
contrib	2024/07/04 3:37	文件夹	
doc	2024/07/04 3:37	文件夹	
fuzz	2024/07/04 3:37	文件夹	
include	2024/07/04 3:37	文件夹	
packaging	2024/07/04 3:37	文件夹	
port	2024/07/04 3:37	文件夹	
samples	2024/07/04 3:37	文件夹	
scripts	2024/07/04 3:37	文件夹	
test	2024/07/04 3:37	文件夹	
tools	2024/07/04 3:37	文件夹	
.clang-format	2024/07/04 3:37	CLANG-FORMAT ...	4 KB

然后打开cmake-gui版，选择如图对应路径，源代码位于 `/assimp-5.4.2`，生成内容位于 `/assimp-5.4.2/build`：

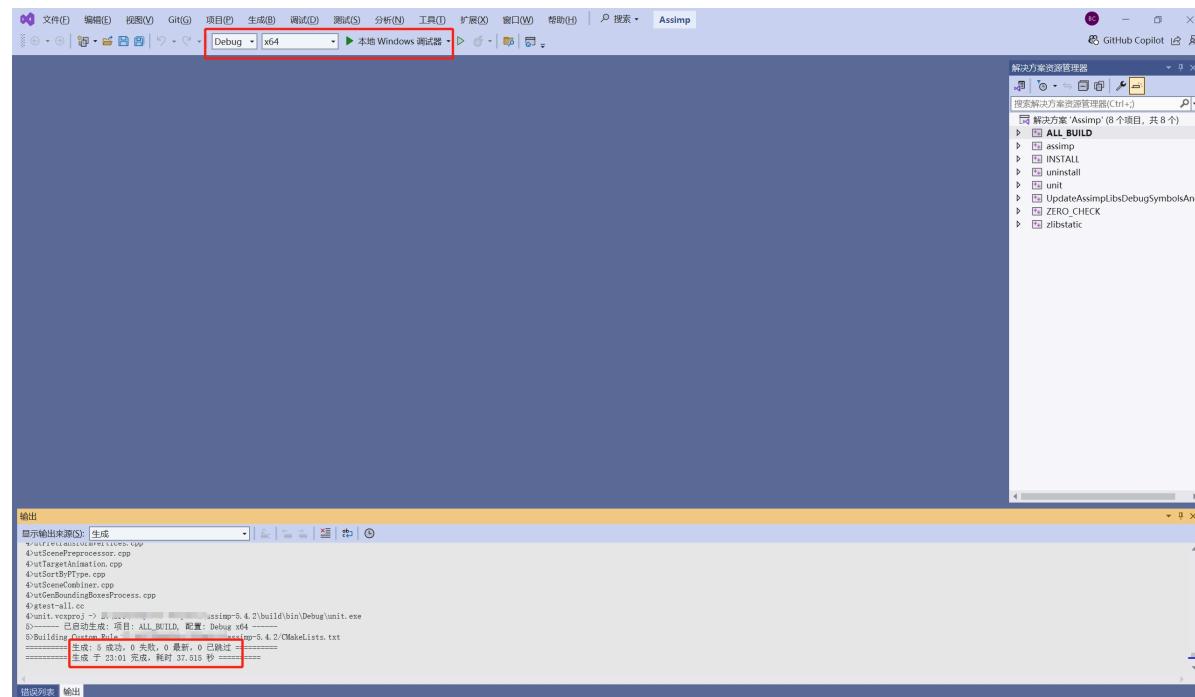


注意我们使用的是VS2022版本，并且选择x64的配置环境。

然后分别点击上图的 configure 和 Generate , 对应的 cmake 生成文件就位于 build 目录下了:

名称	修改日期	类型	大小
CMakeFiles	2024/12/15 22:59	文件夹	
code	2024/12/15 22:58	文件夹	
contrib	2024/12/15 22:58	文件夹	
generated	2024/12/15 22:59	文件夹	
include	2024/12/15 22:58	文件夹	
test	2024/12/15 22:58	文件夹	
ALL_BUILD.vcxproj	2024/12/15 22:58	VC++ Project	45 KB
ALL_BUILD.vcxproj.filters	2024/12/15 22:58	FILTERS 文件	1 KB
assimp.pc	2024/12/15 22:58	PC 文件	1 KB
Assimp.sln	2024/12/15 22:58	Visual Studio Soluti...	8 KB
cmake_install.cmake	2024/12/15 22:58	CMake 源文件	6 KB
cmake_uninstall.cmake	2024/12/15 22:58	CMake 源文件	1 KB
CMakeCache.txt	2024/12/15 22:58	文本文档	19 KB
INSTALL.vcxproj	2024/12/15 22:58	VC++ Project	11 KB

然后使用VS2022打开上图的Assimp.sln文件，接下来开始编译 assimp :



如图所示，选择 Debug 和 x64 ，然后运行：

可以看到成功编译，然后就会弹出**没有Debug内容**的提示，忽略即可，因为我们所需要的只有编译链接过程中产生的 .lib 和 .dll 文件。

assimp\_vc143-mtd.lib 位于 /assimp-5.4.2/build/lib/Debug 中，

在 Debug 中搜索			
名称	修改日期	类型	大小
assimp-vc143-mtd.exp	2024/12/15 23:00	Exports Library File	240 KB
assimp-vc143-mtd.lib	2024/12/15 23:00	Object File Library	402 KB

而 `assimp-vc143-mtd.dll` 位于 `/assimp-5.4.2/build/bin/Debug` 中：

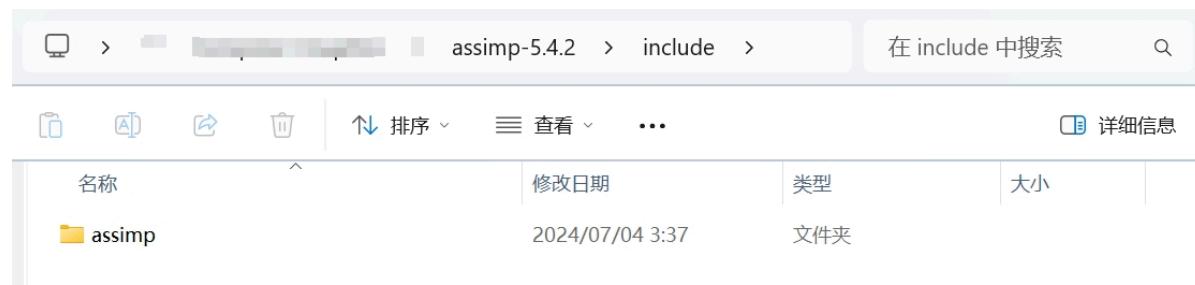
在 Debug 中搜索			
名称	修改日期	类型	大小
assimp-vc143-mtd.dll	2024/12/15 23:00	应用程序扩展	19,337 KB
assimp-vc143-mtd.pdb	2024/12/15 23:00	VisualStudio.pdb.e...	89,044 KB
unit.exe	2024/12/15 23:01	应用程序	6,324 KB
unit.pdb	2024/12/15 23:01	VisualStudio.pdb.e...	24,732 KB

然后把以上两个文件分别放入我们需要运行的项目中，如图：

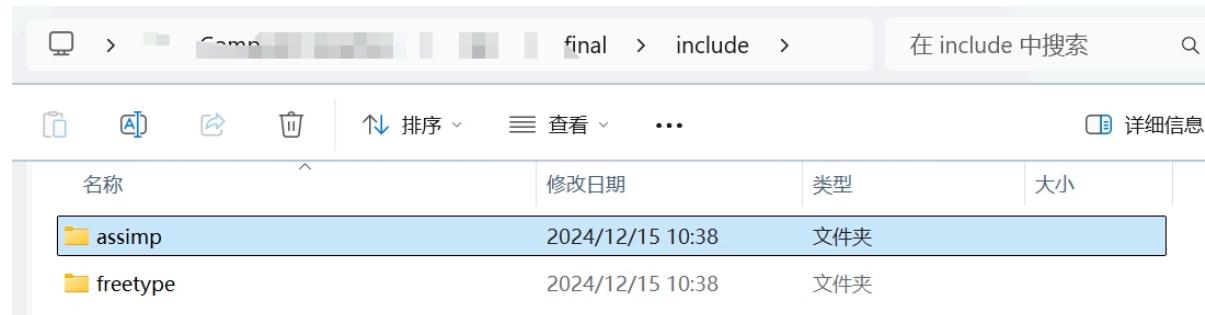
名称	修改日期	类型	大小
.vs	2024/12/15 10:40	文件夹	
.vscode	2024/12/15 10:38	文件夹	
final	2024/12/15 10:40	文件夹	
include	2024/12/15 10:38	文件夹	
lib	2024/12/15 10:38	文件夹	
README	2024/12/15 10:38	文件夹	
resources	2024/12/15 10:38	文件夹	
shader	2024/12/15 10:38	文件夹	
x64	2024/12/15 10:40	文件夹	
.gitattributes	2024/12/15 10:38	Git Attributes 源文件	1 KB
.gitignore	2024/12/15 10:38	Git Ignore 源文件	8 KB
assimp-vc143-mtd.dll	2024/12/15 10:38	应用程序扩展	19,337 KB
final.sln	2024/12/15 10:38	Visual Studio Soluti...	2 KB
final.vcxproj	2024/12/15 10:38	VC++ Project	8 KB

在 lib 中搜索			
名称	修改日期	类型	大小
assimp-vc143-mtd.lib	2024/12/15 10:38	Object File Library	402 KB
glfw3.lib	2024/12/15 10:38	Object File Library	1,426 KB

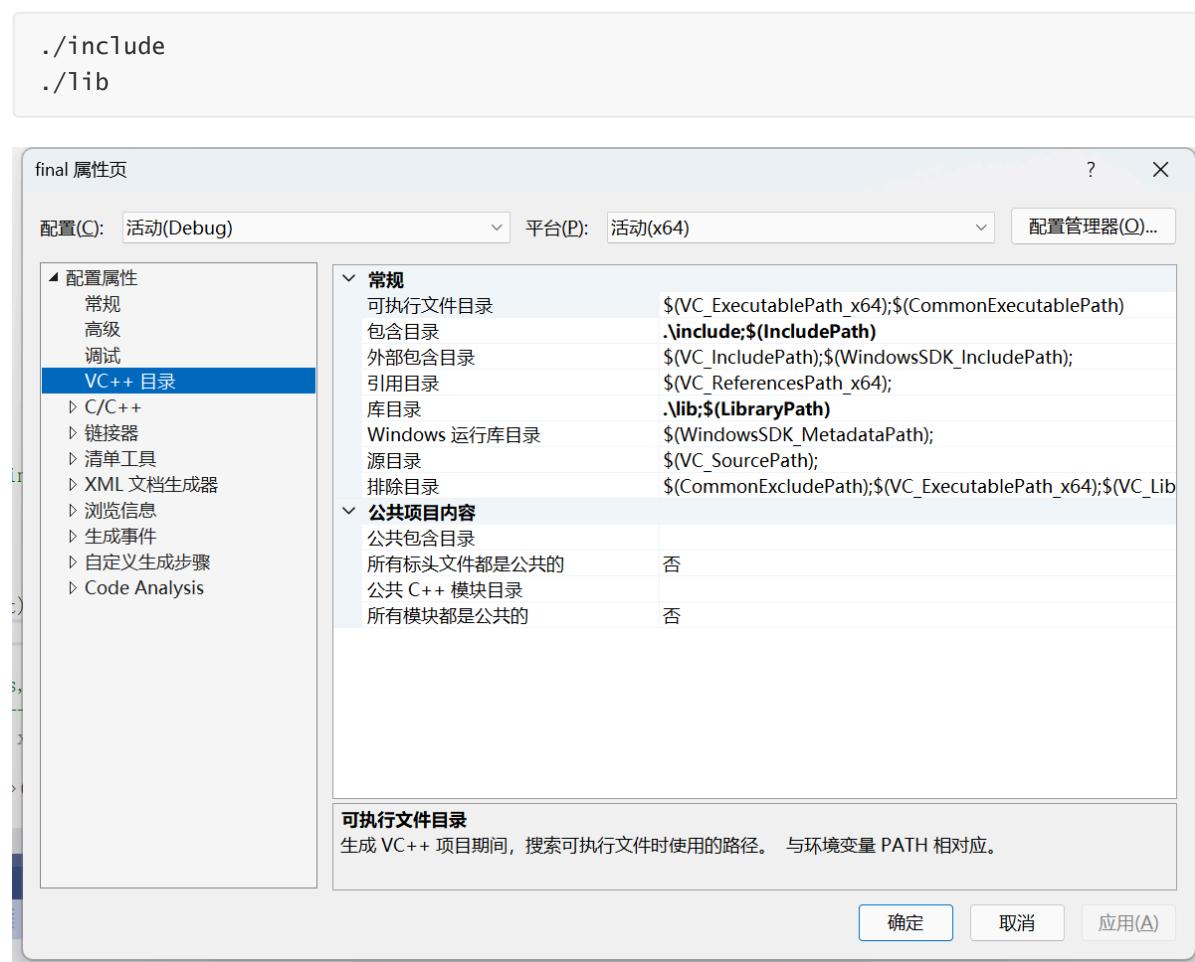
并且把 assimp 的包含文件复制到我们项目的包含文件路径中，assimp 源码的包含文件路径在 /assimp-5.4.2/include：



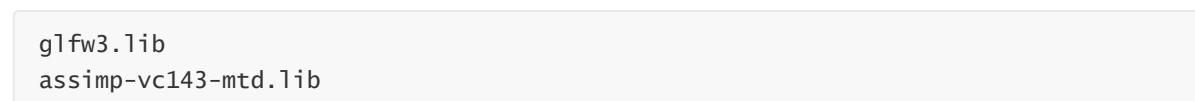
复制上图的 assimp 文件夹到到我们需要运行的项目的 include 中，如 /final/include：

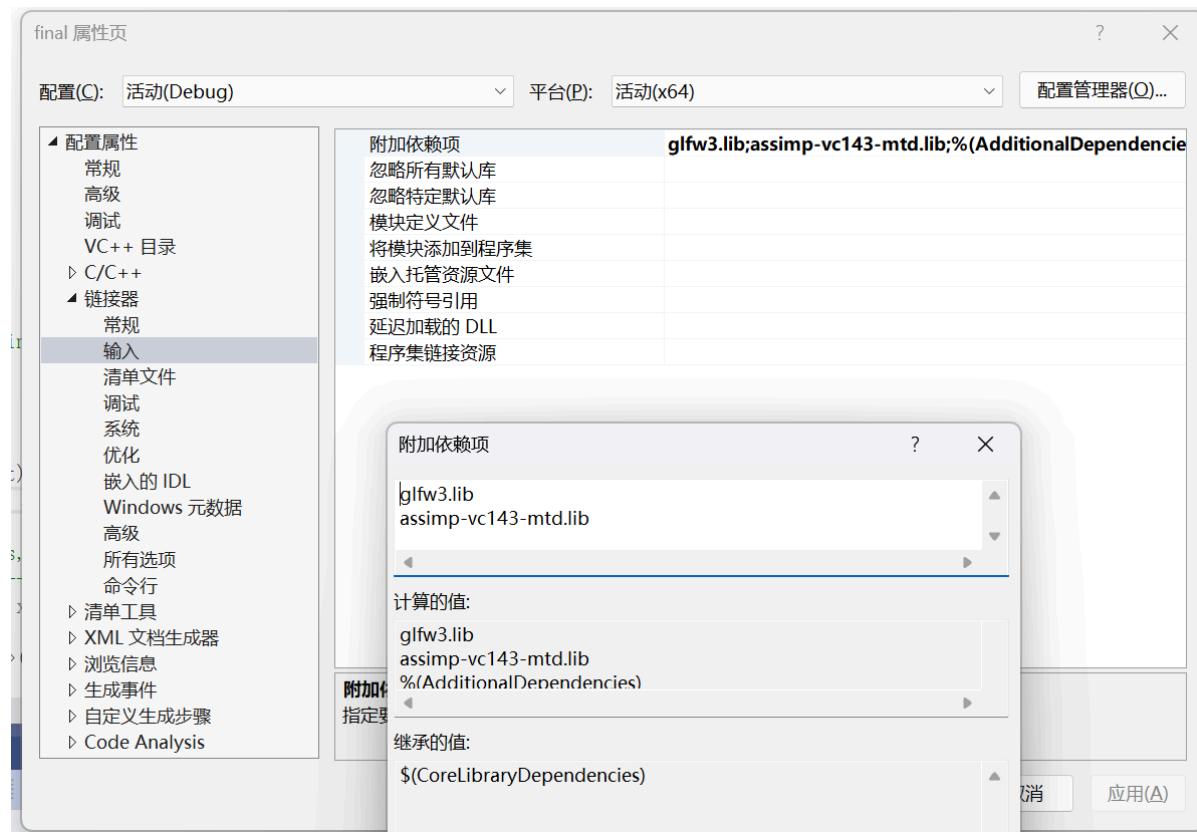


最后在解决方案资源管理器右键-属性添加对应的包含文件路径和库文件路径：



在链接器-输入中添加：





`glfw3.lib` 在 Assignment0 中已经导入，我们不再赘述，此处直接演示。完成上述内容后，便可以调用 `assimp` 库的文件了。

## assimp加载模型文件

这一环节的实现参考了LearnOpenGL CN中“[assimp](#)”的章节。

我们需要做的第一件事是将一个物体加载到**Scene**对象中，遍历节点，获取对应的**Mesh**对象（需要递归搜索每个节点的子节点），并处理每个**Mesh**对象来获取顶点数据、索引以及它的材质属性。最终的结果是一系列的网格数据，我们会将它们包含在一个**Model**对象中。

### 网格类的实现

`Mesh` 类是用于表示和渲染3D模型的基本组件。它封装了模型的顶点数据、索引数据和纹理，并提供了绘制方法。本项目 `Mesh` 的实现参考了learnOpenGL教程中“[网格](#)”章节的实现。

首先我们对顶点与纹理进行抽象，使用 `Vertex` 类与 `Texture` 类来对其进行管理。`Vertex` 类包括位置、法线、纹理坐标、切线、副切线以及骨骼动画相关的属性。`Texture` 类包括纹理的 ID、类型和路径。

```

struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
    glm::vec3 Tangent;
    glm::vec3 Bitangent;
    int m_BoneIDs[MAX_BONE_INFLUENCE];
    float m_Weights[MAX_BONE_INFLUENCE];
};

struct Texture {

```

```
    unsigned int id;
    string type;
    string path;
};
```

之后我们将 `vertex` 与 `Texture` 封装到 `Mesh` 类中，其定义如下：

```
class Mesh {
public:
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    unsigned int VAO;

    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;
        setupMesh();
    }

    void Draw(Shader &shader);
private:
    unsigned int VBO, EBO;
    void setupMesh();
};
```

`Mesh` 类包含了网格的顶点数据、索引数据和纹理数据。

`setupMesh` 方法用于初始化 VBO、EBO 和 VAO，并将顶点数据和索引数据加载到 GPU 中。

`Draw` 方法用于绘制网格：首先激活和绑定纹理，然后绑定 VAO 并调用绘制命令，将顶点与纹理最终绘制到屏幕上。

```
void Draw(Shader &shader)
{
    // ...

    for(unsigned int i = 0; i < textures.size(); i++)
    {
        // ...

        glUniform1i(glGetUniformLocation(shader.ID, (name + number).c_str()), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }

    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()),
    GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

    glActiveTexture(GL_TEXTURE0);
}
```

## 模型类的实现

`Model` 类是对“模型”概念的抽象，主要是对网格与纹理进行了进一步的封装。本项目中 `Model` 类的实现参考了 learnOpenGL 教程中“[模型](#)”的章节。

```
class Model
{
public:
    // model data
    vector<Texture> textures_loaded;      // stores all the textures loaded so far,
    optimization to make sure textures aren't loaded more than once.
    vector<Mesh> meshes;
    string directory;

    // ...
}
```

接下来简要介绍几个关键的函数，包括 `loadModel`，`processNode` 等。

`loadModel` 函数是 `Model` 类的核心函数之一，用于加载模型。通过 Assimp 库读取模型文件，并在读取文件后，调用 `processNode` 函数对模型节点进行解析。

```
void loadModel(string const &path)
{
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_CalcTangentSpace);

    // ...

    processNode(scene->mRootNode, scene);
}
```

`processNode` 函数通过先序遍历的方法，递归的遍历模型的所有节点，解析其中的网格，并将其加入到 `Model` 实例的成员 `meshes` 中。

```
void processNode(aiNode *node, const aiScene *scene)
{
    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);
    }
}
```

上述这两个函数会在一个 `Model` 实例被构造时调用，相当于为模型对象进行了初始化。

最后在需要的时候，我们通过调用 `Draw` 函数，进行模型的绘制。`Draw` 函数会遍历 `meshes` 数组，依次调用其中每个网格对象的 `Draw` 函数，这些网格对象是在 `processNode` 函数中被解析得到的。

```
class Model
{
public:
    // ...

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

到此，我们实现了模型绘制的功能。

### 加载模型实例

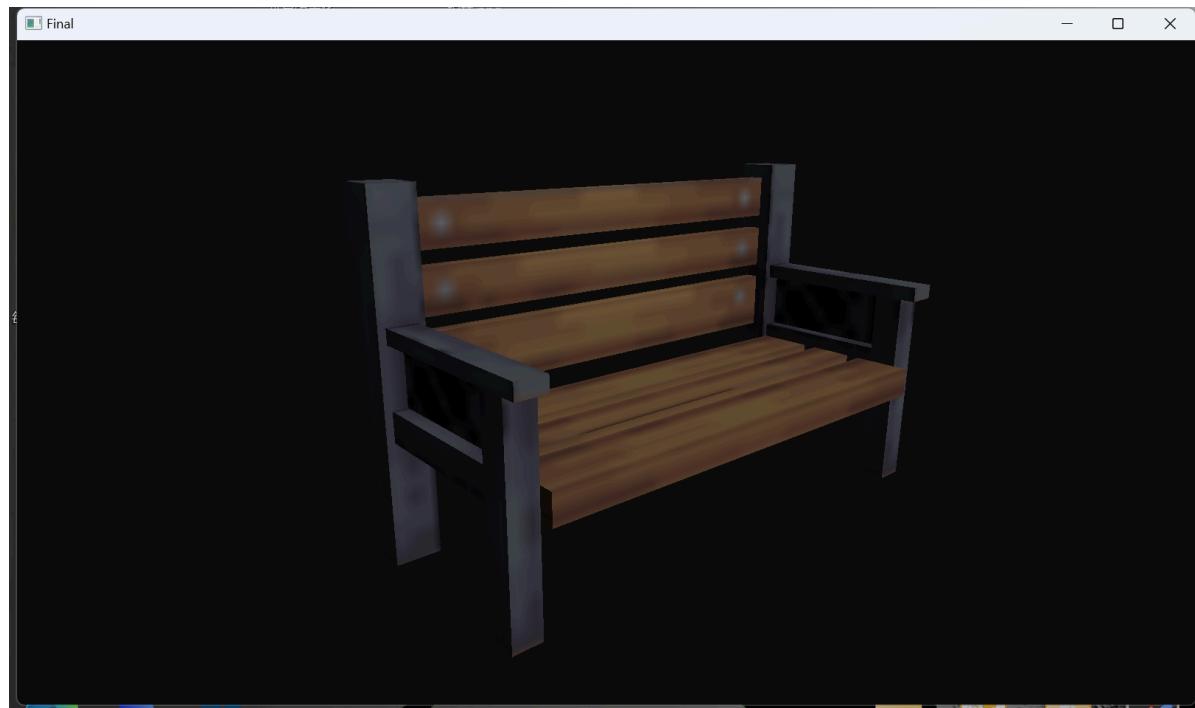
总的来说，在屏幕上绘制一个模型的大致流程如下：

首先初始化一个 Model 实例，然后在渲染循环中的某处，调用 Draw 函数，将模型绘制到屏幕上。

```
Model model("path/to/model");

Render Loop:
// ...
shader = useShader();
modelMatrix = calculateModelMatrix(...);
shader.setModelMatrix(modelMatrix);
model.Draw();
```

如图所示，成功加载长椅模型。

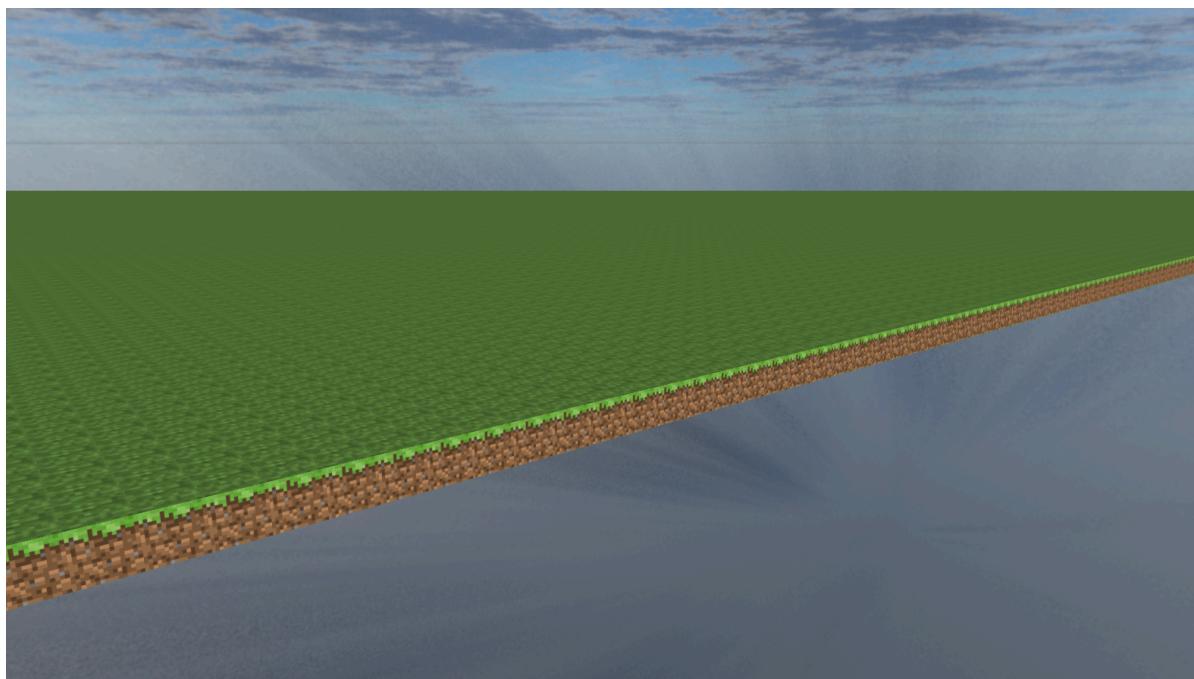
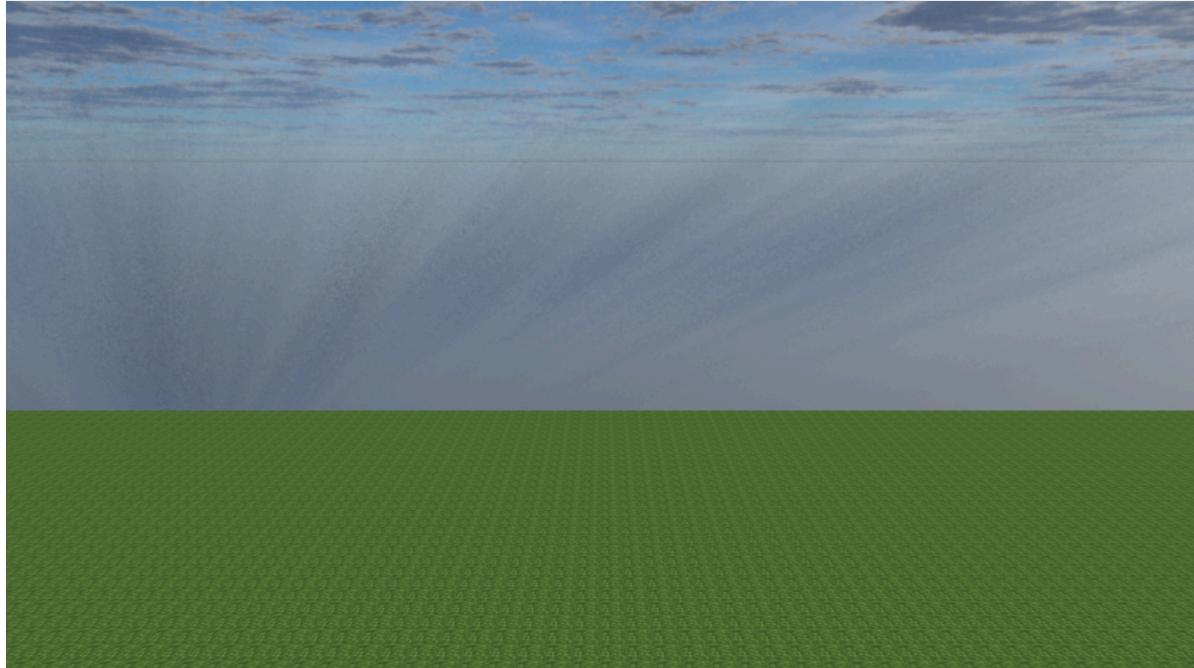


## 实例化渲染地面

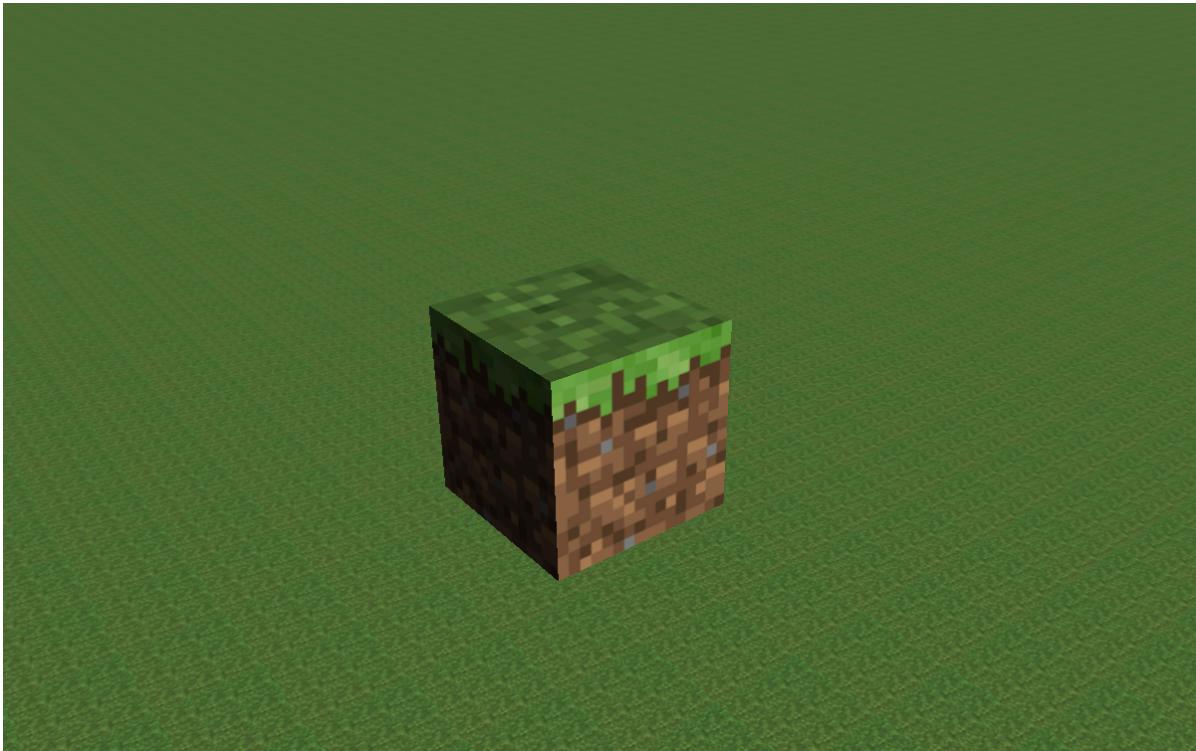
---

这一环节的实现参考了LearnOpenGL CN中“[实例化](#)”的章节。

本项目中，地面是由500x500个简单的方块拼接而成的。下面是渲染好的地面：



组成地面的一个基本方块示意如下，这是一个游戏 `Minecraft` 中草方块制的一个方块模型。



我们通过上述模型，以实例化的方式，绘制出一个“地面”。

我们首先需要通过某种方法计算出，“地面”上每一个方块模型的坐标，这些坐标被存储在数组 `groundModelMatrices` 中（同时也对方块做一些旋转与缩放）。

```
unsigned int size = 500;
glm::mat4* groundModelMatrices;
groundModelMatrices = new glm::mat4[size * size];
for (auto i = 0; i < size * size; i++) {
    glm::mat4 model = glm::mat4(1.0f);
    // Calculate world position of the cube according to i
    int row = i / size;
    int col = i % size;
    model = glm::translate(model, glm::vec3(static_cast<float>(col) -
static_cast<float>(size)/2.0f, -1.0f, static_cast<float>(row) -
static_cast<float>(size) / 2.0f));
    model = glm::scale(model, glm::vec3(0.5f, 0.5f, 0.5f));
    model = glm::rotate(model, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f,
0.0f));
    groundModelMatrices[i] = model;
}
```

之后我们对实例数组进行一些配置：

- 创建一个缓冲区 `buffer`，并将其绑定到 `GL_ARRAY_BUFFER`。
- 使用 `glBufferData` 将 `groundModelMatrices` 数据上传到 GPU。
- 遍历 `grass_cube` 的每个网格，配置顶点属性，使其能够使用实例化绘制技术。
- 使用 `glVertexAttribPointer` 和 `glVertexAttribDivisor` 设置顶点属性的分隔符，确保每个实例使用不同的模型矩阵。

```
unsigned int buffer;
 glGenBuffers(1, &buffer);
 glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

```

glBufferData(GL_ARRAY_BUFFER, size * size * sizeof(glm::mat4),
&groundModelMatrices[0], GL_STATIC_DRAW);
for (unsigned int i = 0; i < grass_cube.meshes.size(); i++)
{
    unsigned int VAO = grass_cube.meshes[i].VAO;
    glBindVertexArray(VAO);

    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)
(sizeof(glm::vec4)));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(2
* sizeof(glm::vec4)));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(3
* sizeof(glm::vec4)));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);
    glBindVertexArray(0);
}

```

最后在渲染循环中，与 `Mesh` 类的 `Draw` 函数类似，我们遍历实例数组的所有网格，绑定纹理，进行绘制。

不同的是，我们在实例化绘制时调用 `glDrawElementsInstanced`，**一次性绘制所有实例**。

```

groundShader.use();

// ...

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, grass_cube.textures_loaded[0].id);
for (unsigned int i = 0; i < grass_cube.meshes.size(); i++)
{
    auto textures = grass_cube.meshes[i].textures;

    // . . .

    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i);

        // ...

        glUniform1i(glGetUniformLocation(groundShader.ID, (name +
number).c_str()), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }
    glBindVertexArray(grass_cube.meshes[i].VAO);
}

```

```
    glDrawElementsInstanced(GL_TRIANGLES, static_cast<unsigned int>(grass_cube.meshes[i].indices.size()), GL_UNSIGNED_INT, 0, size * size);
    glBindVertexArray(0);
    glActiveTexture(GL_TEXTURE0); // reset texture
}
```

到此，我们成功的绘制了地面，实现了通过大量的“草方块”拼接为一个“地面”，并同时保证了程序的流畅性。

## assimp加载动画模型

这一环节的实现参考了LearnOpenGL CN中“[骨骼动画](#)”的章节。

在上一部分的基础上，我们成功加载了模型，而想要让模型动起来，就需要加载骨骼动画。它主要实现 在 `code/include/learnopengl/` 路径下的 `animation.h` 和 `animator.h` 文件。

以下是实现的方法的简要介绍：

`Animation` 类：

- `FindBone`：根据名称查找骨骼，返回相应的指针。
- `GetTicksPerSecond`：获取每秒的滴答数。
- `GetDuration`：获取动画的持续时间。
- `GetRootNode`：获取根节点数据。
- `GetBoneIDMap`：获取骨骼ID映射。
- `ReadMissingBones`：读取动画中涉及的骨骼，并更新模型的骨骼信息映射。
- `ReadHeirarchyData`：递归读取节点层次结构数据，将每个节点的名称和变换存储到 `AssimpNodeData` 中。

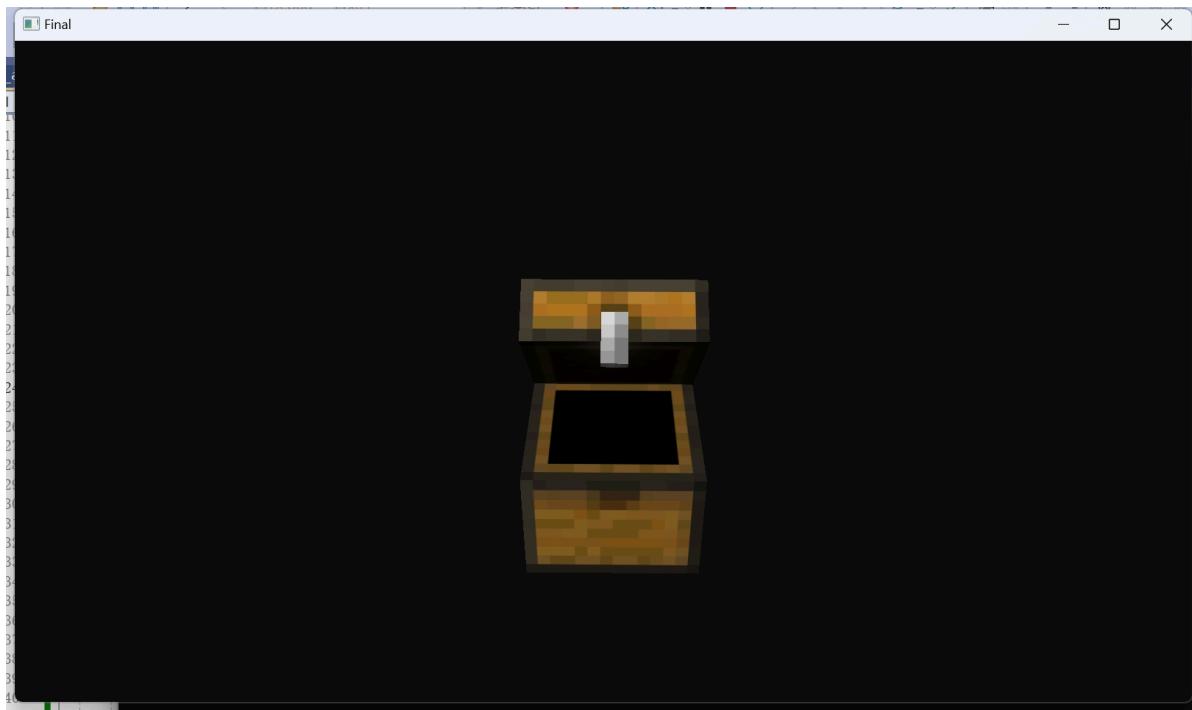
`Animator` 类：

- `UpdateAnimation`：更新当前动画时间，计算骨骼变换。
- `PlayAnimation`：切换到新的动画，并重置当前时间。
- `CalculateBoneTransform`：递归计算每个骨骼的变换，并将其应用到最终骨骼矩阵中。
- `GetFinalBoneMatrices`：返回最终的骨骼矩阵，以便在渲染时使用。

然后我们调用这两个类来加载骨骼动画：

```
Model bModel("resources/minecraft_chest/scene.gltf");
Animation danceAnimation("resources/minecraft_chest/scene.gltf", &bModel);
Animator animator(&danceAnimation);
bModel.CalculateCenter();
```

与静态模型的加载类似，只需额外添加 `Animation` 和 `Animator` 的实例来加载骨骼动画。运行后，可以看到一个箱子不断地打开和关闭，成功加载了动画。



## 着色器详解

所有的着色器都位于 `/code/shader` 目录下，注意 `.vs` 是顶点着色器，`.fs` 是片段着色器。实际上，前面加载和渲染的模型已经使用到了这一部分着色器的内容，并且均完成了 blinn-phong 光照。

`vertex.vs` 是我们最基础的顶点着色器，它进行顶点最基本的几何变换，这是我们着色器渲染的第一步。它通过一个最基本的顶点变换链实现了顶点从模型空间到裁剪空间的变换：

```
void main()
{
    TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

类似的，`fragment.fs` 是我们最基础的片段着色器，它用于从纹理中采样颜色并输出到帧缓冲区，实现最基础的纹理映射功能。它的逻辑同样简单，通过从 `Texcoords` 中使用 `texture_diffuse1` 进行采样即可。

```
void main()
{
    FragColor = texture(texture_diffuse1, TexCoords);
}
```

`ani_shader.vs` 这一顶点着色器通过骨骼影响权重计算顶点的位置和法线，用于动画模型的变形计算，支持最大骨骼数和最大骨骼影响权重，其主要的工作在于通过循环计算骨骼顶点变换并获取法线位置。

代码中的关键点在于在第二个 `if` 语句中，它判断骨骼索引是否越界，如果已经越界则直接使用原始的法线。这一方法的作用是防止无效骨骼影响代码的运行。然后，我们通过使用骨骼变换矩阵与顶点原始位置进行相乘，计算出骨骼对于顶点的变形影响，在最后进行累计加权，根据骨骼位置调整法线位置：

```

for(int i = 0; i < MAX_BONE_INFLUENCE; i++)
{
    if(boneIds[i] == -1)
        continue;
    if(boneIds[i] >= MAX_BONES)
    {
        totalPosition = vec4(aPos, 1.0f);
        totalNormal = aNormal; // 如果没有骨骼影响，使用原法线
        break;
    }
    vec4 localPosition = finalBonesMatrices[boneIds[i]] * vec4(aPos, 1.0f);
    totalPosition += localPosition * weights[i];
    totalNormal += mat3(finalBonesMatrices[boneIds[i]]) * aNormal * weights[i]; // 加权法线
}

```

`ani_shader.fs` 这一片段着色器用于处理了平行光的光照效果并输出最终的片元颜色。有三个 `uniform` 全局变量和经由 `vs_OUT` 结构体从顶点着色器传来的 `vec3 FragPos`, `vec3 Normal` 和 `vec2 TexCoords` 变量, 他们分别用于用于计算观察方向、计算光照和从纹理中采样颜色。通过使用 `Blinn-phong` 光照模型, 我们计算环境光, 漫反射光和镜面反射光, 并在组合后输出最后的颜色, 注意在这里我们将透明度的A分量设置为1, 即完全不透明:

```

vec3 color = texture(floorTexture, fs_in.TexCoords).rgb;
// Ambient
vec3 ambient = 0.05 * color;

// Diffuse
vec3 normalizedLightDirection = normalize(-lightDirection);
vec3 normal = normalize(fs_in.Normal);
float diff = max(dot(normalizedLightDirection, normal), 0.0);
vec3 diffuse = diff * color;

// Specular
vec3 viewDir = normalize(viewPos - fs_in.FragPos);
vec3 reflectDir = reflect(-normalizedLightDirection, normal);
float spec = 0.0;
vec3 halfwayDir = normalize(normalizedLightDirection + viewDir);
spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0); // shininess factor

vec3 specular = vec3(0.3) * spec;

// x100N0E«
FragColor = vec4(ambient + diffuse + specular, 1.0);

```

`blinn-phong.vs` 用于处理顶点的基本变换, 并将计算结果输出给后面的片段着色器。其主要功能包括计算世界空间片段位置、世界空间法向量, 传递纹理坐标和计算裁剪空间位置:

```

vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
vs_out.Normal = normalize(mat3(transpose(inverse(model))) * aNormal);
vs_out.TexCoords = aTexCoords;
gl_Position = projection * view * model * vec4(aPos, 1.0);

```

`blinn-phong.fs` 片段着色器中的操作与 `ani_shader.fs` 中的操作类似，不多赘述。

注意，我们加载静态模型使用的是 `blinn-phong` 着色器，而动画模型使用的是 `ani_shader` 着色器，该着色器同时加载了骨骼动画和 `blinn-phong` 光照。天空盒使用的是另外的着色器，将在下一部分讲解。

## 实现天空盒

---

这一环节的实现参考了LearnOpenGL CN中“[立方体贴图](#)”的章节。

在计算机图形学中，天空盒（Skybox）是一种常见的技术，用于在三维场景中模拟远处的环境背景。其实现主要依赖于立方体贴图（Cubemap），通过将六个图像分别贴到一个立方体的六个面上，模拟天空或环境的全景效果。

### 加载天空盒

在 `loadCubemap` 函数中实现天空盒的加载：

1. 创建纹理对象，将生成的纹理对象绑定到 `GL_TEXTURE_CUBE_MAP` 目标，以便后续操作都应用于这个立方体贴图。
2. 加载六个面纹理：循环遍历每个面，使用 `stbi_load` 函数加载纹理数据，再使用 `glTexImage2D` 函数将加载的图像数据传递给OpenGL，用于为当前绑定的立方体贴图的某一面生成纹理。
3. 设置纹理参数：设置缩小和放大过滤方式为 `GL_LINEAR`，表示使用线性插值来平滑纹理；设置立方体贴图的纹理坐标超出范围时的环绕方式为 `GL_CLAMP_TO_EDGE`，在两个面之间采样时永远返回它们的边界值，以防止出现不连续的边缘。（纹理的R坐标对应的是纹理的第三个维度，和位置的z一样）

```

unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char* data = stbi_load(faces[i].c_str(), &width, &height,
        &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width,
height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i]
<< std::endl;
        }
    }
}

```

```

        stbi_image_free(data);
    }
}

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

return textureID;
}

```

其中，`faces`是一个包含立方体贴图六个面纹理路径的`std::vector`。

```

std::vector<std::string> faces
{
    "resources/textures/clouds/clouds1_east.bmp", // 东
    "resources/textures/clouds/clouds1_west.bmp", // 西
    "resources/textures/clouds/clouds1_down.bmp", // 下
    "resources/textures/clouds/clouds1_up.bmp", // 上
    "resources/textures/clouds/clouds1_north.bmp", // 北
    "resources/textures/clouds/clouds1_south.bmp", // 南
};

unsigned int cubemapTexture = loadCubemap(faces);

```

## 天空盒着色器

构造新的顶点着色器`skybox.vs`用于天空盒。将顶点坐标翻转Y轴方向，确保（大部分的）立方体的纹理方向正确。

注意，将输出位置的z分量设置为它的w分量，这样的话，当透视线除法执行之后，z分量会变为`w / w = 1.0`，最终的标准化设备坐标将永远会有一个等于1.0的z值（最大的深度值）。由此，其它所有的东西都将在天空盒前面，只有在没有可见物体的地方，天空盒才能通过深度测试并被渲染。

```

#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = vec3(aPos.x, -aPos.y, aPos.z);
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}

```

片段着色器`skybox.fs`接受顶点着色器输出的位置向量作为纹理坐标，以采样`samplerCube`：

```

#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}

```

## 渲染天空盒

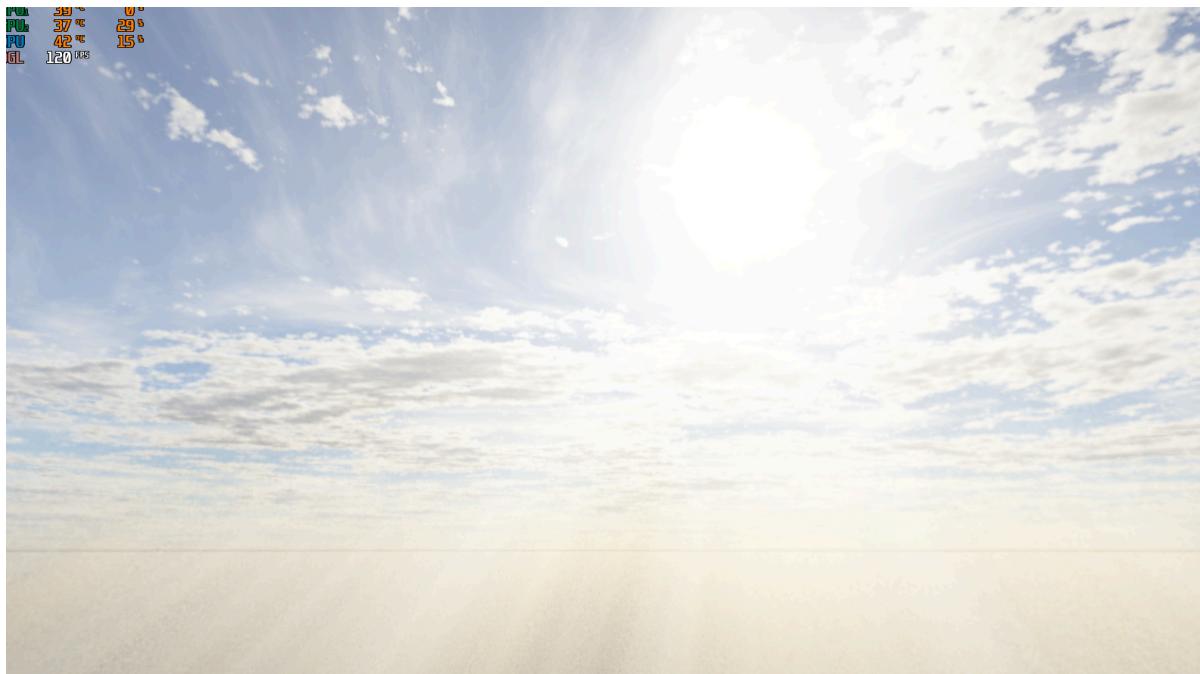
接下来，只需要在渲染循环中绑定立方体贴图纹理，skybox采样器就会自动填充上天空盒立方体贴图。

绘制天空盒时，移除观察矩阵中的位移部分，让移动不会影响天空盒的位置向量，但保留旋转变换，让玩家仍然能够环顾场景；还要将深度函数从默认的GL\_LESS改为GL\_LEQUAL，保证天空盒在值小于或等于深度缓冲而不是小于时通过深度测试，使得深度缓冲能够填充上天空盒的1.0值。结束后记得恢复默认的深度测试函数，避免影响其他场景对象的渲染。

```

{
    glDepthFunc(GL_LEQUAL); // change depth function so depth test passes when
values are equal to depth buffer's content
    skyboxShader.use();
    glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
    glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix())); // remove
translation from the view matrix
    skyboxShader.setMat4("view", view);
    skyboxShader.setMat4("projection", projection);
    // skybox cube
    glBindVertexArray(skyboxVAO);
    glEnable(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthFunc(GL_LESS); // set depth function back to default
}

```



## 摄像机的实现

---

这一环节的实现参考了LearnOpenGL CN中“[摄像机](#)”的章节。

在 `main.cpp` 中，我们使用函数 `processInput` 来实现键盘输入对摄像机位置的控制。它通过 WSAD 控制前后左右方向，Ctrl 键和空格键控制向下和向上运动。它使用 `glfwGetKey` 方法来识别按下了哪个键，并将这些键转换为定义在 `camera.h` 中的一个枚举类，随后将方向数据传给我们的 `camera.h` 中定义的 `ProcessKeyboard` 函数。

```

void processInput(GLFWwindow* window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime * 10.0f);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime * 10.0f);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime * 10.0f);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime * 10.0f);
    if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS)
        camera.ProcessKeyboard(UP, deltaTime * 10.0f);
    if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS || glfwGetKey(window, GLFW_KEY_RIGHT_CONTROL) == GLFW_PRESS)
        camera.ProcessKeyboard(DOWN, deltaTime * 10.0f);
}

```

`Camera_Movement` 枚举类中规定了上下左右和前后六个方向：

```

enum Camera_Movement {
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT,
    UP,
    DOWN
};

```

`ProcessKeyboard` 函数中，我们根据输入的方向进行操作，将摄像机的位置定义为运动速度乘以按键按下的时间来实现运动。与此同时，我们还对于摄像机的高度做了最低的限制，确保它不会穿过地面：

```

void ProcessKeyboard(Camera_Movement direction, float deltaTime)
{
    float velocity = MovementSpeed * deltaTime;
    if (direction == FORWARD)
        Position += Front * velocity;
    if (direction == BACKWARD)
        Position -= Front * velocity;
    if (direction == LEFT)
        Position -= Right * velocity;
    if (direction == RIGHT)
        Position += Right * velocity;
    if (direction == UP)
        Position += Up * velocity;
    if (direction == DOWN)
        Position -= Up * velocity;

    // 限制摄像机高度不能低于地板
    const float minHeight = 0.0f;
    if (Position.y < minHeight)
        Position.y = minHeight;
}

```

我们在 `mouse_callback` 函数中实现通过移动鼠标来移动视角的功能。它提供一个接口，将鼠标移动的数据（通过 `glfwSetCursorPosCallback` 方法读取）接入后传输给预定义的 `ProcessMouseMovement` 函数：

```

void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
    float xpos = static_cast<float>(xposIn);
    float ypos = static_cast<float>(yposIn);

    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}

```

`ProcessMouseMovement` 函数中，我们根据传入的 X 轴和 Y 轴的变换与我们预定义的偏移量相乘后，与偏航角 `Yaw`（控制左右方向）和俯仰角 `Pitch`（控制上下方向）相加。我们还在代码中加入了俯仰角的角度限制，以防止发生上下颠倒的情况。最后我们调用 `updateCameraVectors` 函数，将新的欧拉角数据保存在相机向量中：

```

void ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch = true)
{
    xoffset *= MouseSensitivity;
    yoffset *= MouseSensitivity;

    Yaw += xoffset;
    Pitch += yoffset;

    // make sure that when pitch is out of bounds, screen doesn't get flipped
    if (constrainPitch)
    {
        if (Pitch > 89.0f)
            Pitch = 89.0f;
        if (Pitch < -89.0f)
            Pitch = -89.0f;
    }

    // update Front, Right and Up Vectors using the updated Euler angles
    updateCameraVectors();
}

```

接下来，`scroll_callback`方法提供了缩放的功能，这是通过读取鼠标滚轮的输入（通过`glfwSetScrollCallback`方法得到）来实现的。

```

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(static_cast<float>(yoffset));
}

```

核心的实现方法在`ProcessMouseScroll`函数中，这是通过控制`zoom`的值（即摄像机的视角）来实现的。当我们向前滚动鼠标时，`yoffset`为负值，`zoom`值将变大，导致视角变窄，物体被放大：

![[Pasted image 20241217180315.png]]

最后只需要将摄像机数据传给着色器就大功告成：

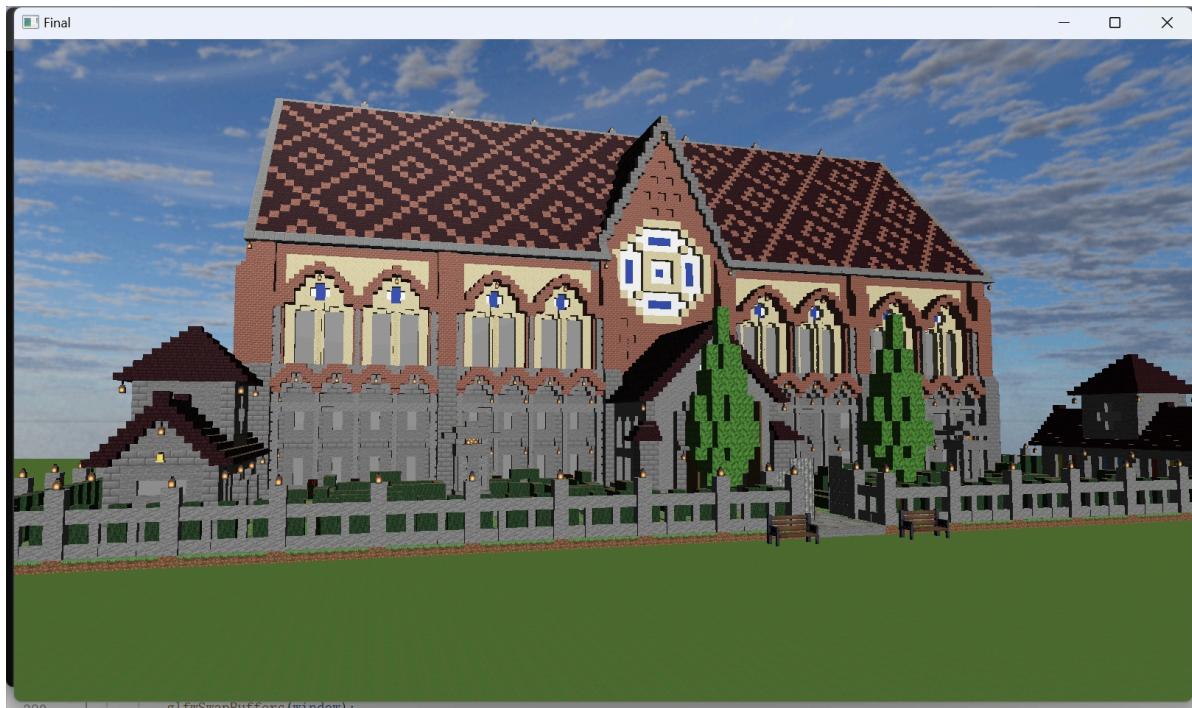
```

ourShader.setVec3("viewPos", camera.Position);

```

## 最终实现效果

成功加载了所有的所需要的静态模型：城堡、长椅等，以及动画模型：宝箱怪，实例化的草地和阳光明媚的天空盒。最终运行的结果如下：



## 小组分工及占比

姓名	主要贡献	总贡献占比
陈柏林	assimp库的编译和导入、静态模型和骨骼动画的实现	25%
黄雍明	完成blinn-phong着色器，实现可交互摄像机	25%
李天扬	寻找静态模型，构建多个Minecraft方块模型，完成实例化渲染地面	25%
吕星龙	寻找动画模型，实现天空盒	25%

## 相关参考网站

1. [LearnOpenGL CN](#) : 提供了 OpenGL 基本理论及其实现。
2. [编译安装Assimp on CSDN](#): 提供了assimp编译以及链接到 OpenGL 项目的教程。
3. [Sketchfab](#): 包含大量免费可下载的静态模型和动画模型。
4. [OpenGameArt](#): 包含大量免费可下载的天空盒。