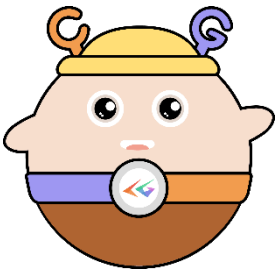
華中科技大學



计算机图形学课程

实验：模型导入

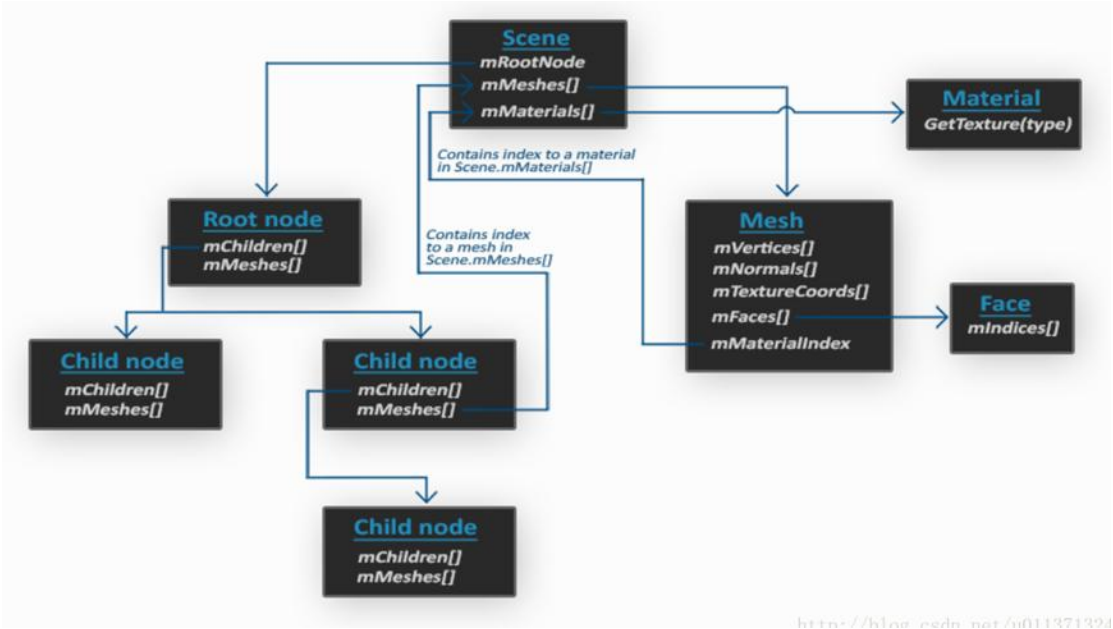
目录

1.模型加载库 Assimp

1.1Assimp

1.1.1Assimp 简介

Assimp 是一个非常流行的模型导入库，它支持多种格式的模型文件，如 obj、3ds、c4e 等。Assimp 加载所有模型和场景数据到一个 Scene 类型的对象中，同时为场景节点、模型节点生成具有对应关系的数据结构。数据结构图如下：



- Scene 对象：包括模型所有的场景数据，如 Material 质和 Mesh。同样，场景的根节点引用也包含在这个 Scene 对象中。
- Root node：可能也会包含很多子节点和一个指向保存模型点云数据 mMeshes[]的索引集合。根节点上的 mMeshes[]里保存了实际了 Mesh 对象，而每个子节点上的 mMesshes[]都只是指向根节点中的 mMeshes[]的一个引用。
- Mesh：本身包含渲染所需的所有相关数据，比如顶点位置、法线向量、纹理坐标、面片及物体的材质。

1.1.2 构建 Assimp

Assimp 可以从它官方网站的下载页上获取。我们选择自己需要的版本进行下载。这里我们建议你自己进行编译，如果你忘记如何使用 CMake 自己编译一个库的话，可以复习创建窗口小节。

【当然我们也为你准备了编译好的库，你可以直接使用它】

下载源码包之后，将其解压并打开。和绘制窗口小节一样，我们需要：

- 编译生成的库
- include 文件夹

库生成完毕之后，我们需要让 IDE 知道库和头文件的位置。有两种方法：

1. 找到 IDE 或者编译器的/lib 和/include 文件夹，添加 Assimp 的 include 文件夹里的文件到 IDE 的/include 文件夹里去。
2. 推荐的方式是建立一个新的目录包含所有的第三方库文件和头文件，并且在你的 IDE 或编译器中指定这些文件夹。我个人会使用一个单独的文件夹，里面包含 Libs 和 Include 文件夹，在这里存放 OpenGL 工程用到的所有第三方库和头文件。这样我的所有第三方库都在同一个位置（并且可以共享至多台电脑）。然而这要求你每次新建一个工程时都需要告诉 IDE/编译器在哪能找到这些目录。

完成上面步骤后，我们就可以使用 Assimp 来导入漂亮的 3D 模型了！

2.Mesh && Model 类的创建

2. 1Mesh

2.1.1 数据结构

在构建 Mesh 和 Model 之前我们需要用到之前的 Assimp 库的相关头文件，因此在文件头加上相关头文件：

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

网格(Mesh)代表的是单个的可绘制实体，一个网格应该至少需要一系列的顶点，每个顶点包含一个位置向量、一个法向量和一个纹理坐标向量。一个网格还应该包含用于索引绘制的索引以及纹理形式的材质数据（漫反射/镜面光贴图）。

因此我们这样定义网格的数据结构：

```
struct Vertex {
        glm::vec3 Position;
        glm::vec3 Normal;
        glm::vec2 TexCoords;
};
```

我们将所有需要的向量储存到一个叫做 Vertex 的结构体中，我们可以用它来索引每个顶点属性。除了 Vertex 结构体之外，我们还需要将纹理数据整理到一个 MeshTexture 结构体中。

```
struct MeshTexture {
        unsigned int id;
        string type;
        string path;
};
```

该数据结构中存储了纹理的 id 和它的类型，比如是漫反射贴图或者是镜面光贴图，以及其路径。这在后面解析模型的纹理有重要作用。在我们实现顶点和纹理的猴，我们可以开始定义网格类的结构，如下所示：

```
class Mesh {
    public:
        /*网格数据*/
        vector<Vertex> vertices;
        vector<unsigned int> indices;
        vector<MeshTexture> textures;
        /*初始化函数*/
 Mesh(vector<Vertex>vertices,vector<unsigned int>indices,vector<Texture>
textures);
/*绘制函数*/
        void Draw(Shader shader);
    private:
        /*渲染数据*/
        unsigned int VAO, VBO, EBO;
                /*初始化网格数据*/
        void setupMesh();
};
```

setupMesh 函数中初始化缓冲，并最终使用 Draw 函数来绘制网格

2.1.2setupMesh()函数

首先我们来了解一下构造器函数 Mesh()，它有如下形式：

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

       setupMesh();
   }
```

前面部分非常好理解，关键是 setupMesh 函数，

```
    void setupMesh()
        {
            glGenVertexArrays(1, &VAO);
            glGenBuffers(1, &VBO);
            glGenBuffers(1, &EBO);
            glBindVertexArray(VAO);
            glBindBuffer(GL_ARRAY_BUFFER, VBO);
            glBufferData(GL_ARRAY_BUFFER, vertices.size() *
sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);
            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
            glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(unsigned int), &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);
        }
```

这里运用到了前面关于 VAO, VBO 和 EBO 的相关知识，这里不再重复说明。通过 setupMesh()函数我们便将绘制模型所需要的顶点传到了对应的缓冲区中，便于进行下一步的绘制工作。

2.1.3Draw()函数

我们是使用 Draw()函数来绘制每个网格，在真正渲染这个网格之前，我们需要在调用 glDrawElements 函数之前先绑定相应的纹理。然而，这实际上有些困难，我们一

开始并不知道这个网格（如果有的话）有多少纹理、纹理是什么类型的。所以我们下一步定义纹理单元和采样器。

为了解决这个问题，我们需要设定一个命名标准：每个漫反射纹理被命名为 texture_diffuseN，每个镜面光纹理应该被命名为 texture_specularN，其中 N 的范围是 1 到纹理采样器最大允许的数字。

有了这些规定，我们写出最终的渲染函数：

```
void Draw(Shader shader)
{
unsigned int diffuseNr = 1;
unsigned int specularNr = 1;
unsigned int normalNr = 1;
unsigned int heightNr = 1;
for (unsigned int i = 0; i < textures.size(); i++)
{
glActiveTexture(GL_TEXTURE0 + i);
string number;
string name = textures[i].type;
if (name == "texture_diffuse")
number = std::to_string(diffuseNr++);
else if (name == "texture_specular")
number = std::to_string(specularNr++);
glBindTexture(GL_TEXTURE_2D, textures[i].id);
}
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
glActiveTexture(GL_TEXTURE0);
}
```

这样，我们便完成了 Mesh 类，接下来我们来讲如何写一个适合的 Model 类进行模型的绘制。

2.2Model

2.2.1 数据结构

模型是多个网格的集合，因此这一小节我们写一个 Model 类，通过使用 Assimp 和 Mesh 类完成对 3D 模型的绘制。同样地，我们将分别讲解 Model 类中主要的函数方法。

2.2.2loadModel ()函数

该函数用于加载模型，在加载模型模型之后，我们会检查场景和其根节点不为 null，并且检查了它的一个标记(Flag)，来查看返回的数据是不是不完整的。如果遇到了任何错误，我们都会通过导入器的 GetErrorString 函数来报告错误并返回。我们也获取了文件路径的目录路径。

```
    void loadModel(string const &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)
                {
                        cout << "ERROR::ASSIMP:: " <<
importer.GetErrorString() << endl;
                        return;
                }
                directory = path.substr(0, path.find_last_of('/'));
                processNode(scene->mRootNode, scene);
        }
```

若无错误发生，我们想要处理场景中的所有节点，将根节点传入了递归的 processNode 函数。因为每个节点（可能）包含有多个子节点，这样可以遍历到各个节点。

2.2.3processNode()和 processMesh()函数

processNode()函数检查每个节点的网格索引，并索引场景的 mMeshes 数组来获取对应的网格。返回的网格将会传递到 processMesh 函数中，它会返回一个 Mesh 对象，我们可以将它存储在 meshes 列表/vector。所有网格都被处理之后，我们会遍历节点的所有子节点，并对它们调用相同的 processMesh 函数。当一个节点不再有任何子节点之后，这个函数将会停止执行。

processMesh()则将 aiMesh 对象转化为我们自己的网格对象，处理网格的过程主要有三部分：获取所有的顶点数据，获取它们的网格索引，并获取相关的材质数据。处理后的数据将会储存在三个 vector 当中，我们会利用它们构建一个 Mesh 对象，并返回它到函数的调用者那里。两个函数的步骤如下：

```
void processNode(aiNode *node, const aiScene *scene)
{
    // 处理节点所有的网格）
    for(unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }
    // 接下来对它的子节点重复这一过程
    for(unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}
```

```
Mesh 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;
            // 处理顶点位置、法线和纹理坐标
            ...
            vertices.push_back(vertex);
        }
        // 处理索引
        ...
        // 处理材质
        if(mesh->mMaterialIndex >= 0)
        {
            ...
        }

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

具体的 model 代码写法较为复杂，这里就不在全部展示出来，你可以查询工程文件下的头文件去查看代码的写法。在完成这些步骤后，我们就可以进行最后一步，把模型文件导入到我们的场景中去。

3.绘制模型

在做好前面的准备工作后，绘制模型就变得相当简单了。我们只需要加载模型，然后调用其 Draw()方法即可。效果如下：