从0开始的OpenGL学习（十七）-加载模型

闪电的蓝熊猫 关注 IP属地: 江苏

0.299 2017.11.12 08:04:47 字数 5,418 阅读 15,027

本文主要解决一个问题：

> 如何在OpenGL中加载模型？

引言

学到现在，我们把盒子兄弟折磨得死去活来，虽说弄出了一些效果，但也总是感觉有点不给力，换个时髦的说法就是：用户体验不好。在实际的图形应用中，会有很多复杂并且有趣的模型，比我们的盒子强太多。但是，由于太复杂，我们不可能手动定义模型的顶点坐标、法线和纹理坐标等值。我们希望的是，直接把模型导入到应用中使用，把创建模型的工作交给专业的建模师去做。他们有很高端的工具，例如3DS Max、Maya等等。

这些3D建模工具十分强大，不仅可以创建模型，还能进行纹理贴图。导出的模型文件中，包含了我们需要的所有数据，包括顶点坐标、法线、纹理坐标等等。这样建模师就能专注做出酷炫的模型而不必操心图形程序员如何读取的问题。

我们的工作，自然就是读取这些模型文件，然后将其保存到应用之中备用。如果模型文件中的数据格式不能直接被OpenGL使用，我们也有责任将其转换成能被OpenGL识别的格式。一个严峻的问题是，模型的文件格式非常多，每个格式都有一套独特的保存数据的方法，我们需要一个个地去解析吗？理论上来说，是的。但是我们很幸运，已经有人做了这些事，我们只需要站在巨人的肩膀上就可以了。

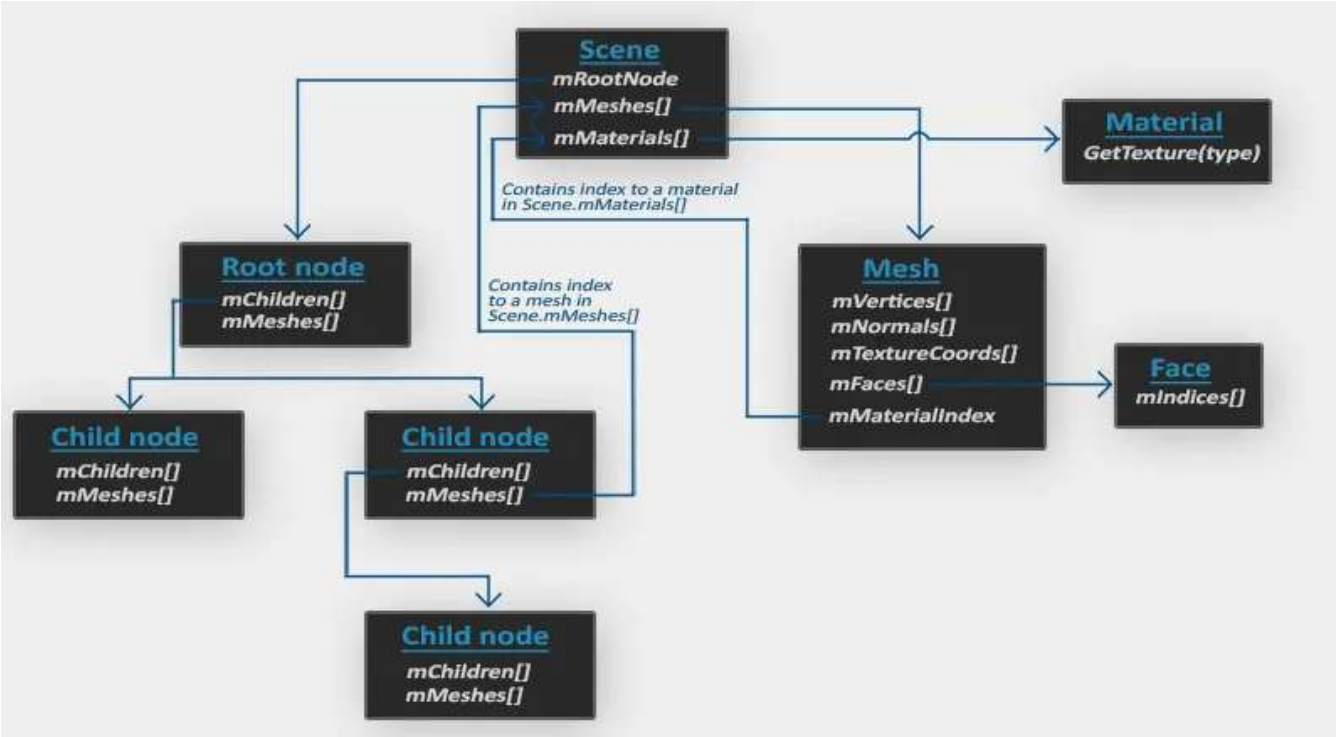
本文结构：

- 介绍Assimp库的基本内容
- 介绍用于OpenGL的数据结构
- 从Assimp读取模型，转换到自定义的数据结构使用

Assimp库

Assimp是一个比较流行的开源模型加载库。其作用是将市面上杂七杂八的各种格式的模型文件解析出来，保存成它自己的一套数据结构。当Assimp加载完模型之后，我们就能从其数据结构中提取数据了。因为它的数据结构是唯一的，我们就能轻松地从中提取数据，保存成可以在OpenGL中使用的结构。你可能要问了，为什么不直接使用Assimp的数据结构呢？原因嘛，笔者也不知道，可能是想更通用吧。

所有的模型被Assimp加载之后，都会保存到一个scene对象中。这个scene对象包含了一个根节点，根节点中包含了很多子节点，每个节点中都包含了网格数据。scene对象还包含了网格数据和材质数据。大致的结构是这样子：



Assimp结构

详细介绍一下这些基础但是非常重要的结构：

- Scene类：模型中的所有数据都会被保存在这个类的对象中，包括网格、材质、纹理、动画等等。它也包含了一个模型根节点的引用。
- Node类：成功加载模型后最少会有一个节点，这个节点就被称为跟节点。节点里包含了网格索引的数据，父节点信息和子节点信息，类似于树一样的结构。
- Mesh类：网格中包含了渲染所用的数据信息。包括顶点、法线、纹理坐标、面片（Face）、材质等信息。一个网格可能有多个面片。
- Face类：代表了物体的图元（三角形、正方形，点）。面片类中包含了图元的顶点索引，因为顶点和索引是分开的，我们很容易地就能使用索引缓存来渲染。
- Material类：scene对象中保存着所有的材质，每个网格都有一个材质索引，一个网格对象只有一种材质。

了解了Assimp的主要结构，我们大致可以整理出一个读取数据然后应用的流程：

1. 将使用Assimp读取成一个scene对象
2. 从根节点开始，检索每个节点的网格数据
3. 从网格数据中读取顶点数据、索引数据和材质属性等等
4. 将读取到的数据保存到自定义的数据结构中
5. 运用之前学到的知识，显示模型

编译Assimp

你可以访问Assimp的官方网站上去下载合适的版本，最新的版本是4.0.1，笔者选用的版本是3.3.1，没啥原因，就是看着3.3和我们的OpenGL核心模式的版本一样，有好感而已，哈哈。当然，如果你没法FQ，那么也可以下载笔者上传到网盘中的3.3.1版本源码（提取码：1au7）。

下载完成之后，使用CMake进行编译，如果你已经忘了，请参考之前的文章进行编译。笔者选择的IDE是VS2013，生成VS2013的解决方案之后，在Debug和Release模式下都编译一遍，得到的assimp-vc120-mt.lib和assimp-vc120-mt.dll就是我们所需要的东西了。这两个东西在lib和bin两个文件夹下面。

将这两个文件复制到合适的位置，确保我们的程序可以调用到。笔者将dll文件拷贝到了工程的运行目录下，将lib文件拷贝到了glfw3.lib所在的目录。笔者在这一步上没有遇到什么问题，如果你在编译的时候遇到问题，请随意在下面进行留言。

好，下一步，我们要来定义自己的数据结构了。

自定义数据结构

回想之前的例子，我们在绘制盒子的时候，使用的是一个顶点数组，顶点数组中包含了位置、法线、纹理坐标等信息。再看看Assimp的结构，我滴老天，这些都是啥，我们要怎样才能显示这些东西？别怕，我们不用实现所有的东西，只要能显示的部分就行了。那就开始找呗，scene类？不像，

里面的东西太多了。node类？有点像，不过里面还有子节点，这好像不需要。mesh类？好像是这货，但是里面的face要怎么实现？face类？里面有我们熟悉的顶点索引，但只有索引有什么用，具体数据呢？

找来找去，也只有mesh类看上去最顺眼，最合适，就它了。至于face，可以直接包含在类里面，也没多大的问题。整理一下思绪，mesh中肯定需要顶点位置、顶点法线、纹理坐标，要有材质（就是纹理），还要有索引。不着急把这些东西一股脑放进mesh类中，先来构思构思成员对象。

首先是顶点类，里面需要有位置、法线、纹理坐标信息，通过一个结构体的方式组合起来，使用的时候非常方便。定义如下：

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

除此之外，我们还需要一个纹理结构，包含纹理类型以及纹理ID，像这样：

```
struct Texture {
    unsigned int id;
    std::string type;
        std::string path;
};
```

有了这些基础的结构之后，我们就可以来定义网格类了。它不仅需要数据，还需要渲染的环境，我们会给他一个绘制函数，调用这个函数就会渲染这个网格对象。

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

```
public:
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> texture;

private:
    unsigned int VAO, VBO, EBO;     //渲染环境

    void setupMesh();
};
```

这个类并不复杂，里面仅仅包含了一些我们需要的东西而已。构造函数需要顶点、索引和纹理数据来进行初始化。渲染环境也在初始的时候就会生成好，随着网格生而生，自然也会随着网格亡而亡，网格的渲染只会使用这环境。
在绘制函数Draw里传递着色器是为了设置着色器中的材质纹理，其余的内容应该都是非常直观，容易理解的。

构造函数中，我们要把输入的参数全都复制给成员变量，然后调用setupMesh函数来初始化环境：

```
Mesh::Mesh(const std::vector<Vertex>& vertices, const std::vector<unsigned int>& indices, std::vector<Texture>& textures) {
    (this->vertices).insert((this->vertices).end(), vertices.begin(), vertices.end());
    (this->indices).insert((this->indices).end(), indices.begin(), indices.end());
    (this->textures).insert((this->textures).end(), textures.begin(), textures.end());

    setupMesh();
}
```

初始化

有了网格的数据之后，我们就要创建渲染环境了，包括VAO、VBO、EBO都要。创建环境的方法我们已经很熟悉了，无非就是先VAO、VBO、EBO的ID，然后分配内存，然后绑定到OpenGL的环境中，然后往里面塞数据，然后完事。根据这个思路，我们来组织代码：

```
void Mesh::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);

    //顶点位置
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);

    //顶点法线
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));

    //纹理坐标
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));

    //恢复默认的VAO环境
    glBindVertexArray(0);
}
```

代码非常熟悉，和之前我们写的代码几乎是没有区别。有一些不同的地方在这里做一下简单的解释（如果你熟悉C++，就可以跳过这段了。）：

首先是填充VBO时使用的参数变了，不再是我们熟悉的sizeof（数组），而是采用了数组长度*成员大小的方式，然后起始位置也成了取首元素的地址，这些的效果都是一样的。vector内部的元素也是彼此紧挨着存放，不存在空隙。

然后是设置顶点格式的时候，获取起始位置偏移也采用了offsetof宏，这个宏是用来获取某个成员的偏移的，比直接计算出偏移值写上去的可读性大了不知道多少。

使用这些结构设置不仅可以大大增强可读性，而且还能让我们在扩展功能的时候（例如往顶点结构中加个什么数据）不要再改其他的代码，大大提高效率，减少重复劳动的时间。不管我们写什么代码，这点都应该是要牢记的。

渲染

剩下的一个函数就是Draw函数了。在真正执行渲染操作（调用glDrawElements函数）前，我们需要把适当的纹理绑定到着色器中（这也是为什么要有一个着色器输入的原因）。

找出可显示的结构，找到mesh，然后深入mesh，找其内部需要的东西，顶点、纹理就是。mesh的数据成员，mesh的函数成员，然后实现函数成员。一个很困扰的问题就是我们不知道着色器中有多少纹理需要赋值，我们也不知道着色器中纹理变量的名字是啥。于是，我们只能做出一些假设，假设着色器中有3个漫反射纹理和2个镜面高光纹理，他们的名字有着如下的规律：

```
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
```

做出这样的假设后，我们在渲染函数中就可以构造纹理名，然后设置相应的纹理了。Draw就变成了这个样子：

```
void Mesh::Draw(Shader shader) {
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for (unsigned int i = 0; i < textures.size(); ++i) {
        glActiveTexture(GL_TEXTURE0 + i);
```

```
            std::string number;
            std::string name = textures[i].type;
            if (name == "texture_diffuse")
                number = std::to_string(diffuseNr++);
            else if (name == "texture_specular")
                number = std::to_string(specularNr++);

            shader.setFloat(("material." + name + number).c_str(), i);
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
        }

        //渲染
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

            glActiveTexture(GL_TEXTURE0);
    }
```

代码还是一如既往的直观。在设置纹理的时候，我们还采用了一些小技巧，就是通过纹理的索引构造纹理名进行设置。这样，我们的网格数据就算完成了。

> 解决设置纹理问题的方法有很多种，你不必一定要按照这种方法来设置，用你自己的方法解决问题是最好的！

模型

终于走到这一步了，现在，我们需要一个东西来把整个模型包含进去作为一个整体，这就是我们要创建的模型类。一个模型拥有多个网格组成，并配有相应的处理函数，如节点处理、网格处理、加载模型、加载纹理、渲染等。使用模型类需要先初始化，然后在绘制的地方调用渲染函数。

没什么再说的，快来看看我们的模型类：

```
class Model{
public:
    Model(char* path) {
        loadModel(path);
    }

    void Draw(Shader shader);

private:
    std::vector<Mesh> meshes;
    std::string directory;

    void loadModel(std::string path);
    void processNode(aiNode* node, const aiScene* scene);
    Mesh processMesh(aiMesh* mesh, const aiScene* scene);
    std::vector<Texture> loadMaterialtextures(aiMaterial* mat, aiTextureType type, std::string typeName);
};
```

代码很直观。我们先看来最简单的渲染函数Draw，这函数只需要遍历成员变量meshes，调用每个mesh的Draw函数就可以了。

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

导入3D模型到OpenGL

为了导入模型并转换成我们自定义的数据结构，我们需要包含Assimp的一些头文件。把"assimp-3.3.1\include"文件夹下的assimp文件下拷贝到我们统一的头文件目录下，然后包含进来：

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

我们第一个调用的函数必定是构造函数，而在这之后调用的，便是loadModel函数。在这函数中，我们会把模型用Assimp加载进来，保存成一个scene对象。这是Assimp的一个根对象。模型加载成scene对象之后，我们就可以从里面读取数据进行转换纹理。

加载模型的代码如下所示：

```
//导入成scene对象
Assimp::Importer importer;
const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
```

我们先定义一个Importer对象，然后调用其ReadFile函数。该函数需要一个文件路径和post-processing选项作为参数。除了简单地从文件中读取数据，Assimp还允许我们指定一些选项来进行额外的处理。aiProcess_Triangulate参数表示我们希望把模型中的基本图形变成三角形，aiProcess_FlipUVs参数表示我们希望翻转纹理y坐标（如果有必要的话），还记得我们介绍纹理章节中的操作吗？我们加载纹理之后，还调用了 `stbi_set_flip_vertically_on_load` 函数来反转y坐标。这里我们只需简单地设置一个选项就能做到，比手动调用函数方便多了。想了解更多的选项，请查看Assimp源码或者到这个网址查看，这里就不多啰嗦了。

加载一个模型成Scene对象十分简单，真正的挑战是如何把数据转换到自定义数据结构中。完整的loadModel函数如下所示：

```
//加载模型
void Model::loadModel(std::string path) {
    //导入成scene对象
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);

    //检测是否成功
```

```
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) {
        std::cout << "Assimp加载模型失败，错误信息: " << importer.GetErrorString() << std::endl;
        return;
    }

    directory = path.substr(0, path.find_last_of('/'));

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

加载完模型后，我们检查了加载的scene对象是否有效，检查了scene对象中的数据是否完整，还检查了scene对象中的根节点是否有效。如果这些有一个是无效值，那就说明我们加载失败了，剩下的工作无法继续进行，报出错误信息返回。

如果加载成功，我们就要保存当前的目录以备调用。然后，调用processNode来处理节点数据，每一个节点都可能包含子节点，所以这个函数会在其内部进行递归调用。

Assimp的节点类中包含一些网格索引信息，检索和处理每个网格是我们的函数要做的主要事情。processNode函数内容如下所示：

```
//处理节点
void Model::processNode(aiNode* node, const aiScene* scene) {
    //处理节点的所有网格信息
    for (int i = 0; i < node->mNumMeshes; ++i) {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }

    //对所有的子节点做相同处理
    for (int i = 0; i < node->mNumChildren; ++i) {
        processNode(node->mChildren[i], scene);
    }
}
```

处理网格

将一个aiMesh对象转换成我们的mesh对象并不难，我们所要做的就是获取aiMesh中网格的所有属性，然后将其保存到mesh对象中。processMesh函数的大致结构就变成如下的样子：

```
//处理网格
Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene) {
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;

    for (int i = 0; i < mesh->mNumVertices; ++i) {
        Vertex vertex;
        //处理顶点，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);
}
```

收拾一个网格基本上有三个步骤要做：检索所有的顶点数据、检索所有的网格索引、检索相关的材质数据。将处理完成的数据保存到3个向量中，传递给Mesh的构造函数，生成一个新的Mesh对象返回。

步骤一、检索顶点数据

顶点数据在aiMesh对象中被保存成三个不同的部分，分别保存在mVertices（位置），mNormals（法线），mTextureCoords（纹理坐标）。遍历这三个数组，将数组的元素提取出来放到一个vector对象中，然后保存到Vertex对象中，我们的任务就完成了：

```
//处理顶点，法线和纹理坐标
glm::vec3 vector;
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;

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

if (mesh->mTextureCoords[0]) {  //看看是否有纹理信息
    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);
```

值得一提的是，Assimp允许每个顶点有最多8个纹理坐标，但是我们不准备都是用，我们就是用第一个就好。实现的时候，我们也检查了是否存在纹理，如果不存在，就给一个无效值。

检索索引

Assimp中，每个网格都有一组面片组成，每个面片都是一个图元，在我们这就是三角形。每个面片中都有索引信息，指明了要用到哪些顶点来绘制，按照什么顺序来绘制。所以，我们要遍历所有的面片，将所有的索引信息都保存到mesh对象的索引数组中：

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

处理完成之后，我们就可以通过glDrawElements函数来绘制网格了。但是，为了显示的效果更好，我们还需要来添加网格的材质信息才行。

相关材质

和顶点的数据一样，材质的数据都被保存在scene对象中，mesh对象中只有材质的索引，我们可以判断mesh的材质索引字段是否有效，如果有效再把数据保存到Model对象中去:

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

我们首先从scene的mMaterials数组中获取了aiMaterial对象。然后，调用自定义的loadMaterialtextures函数加载所有的漫反射和镜面高光纹理。该函数会返回一个Texture结构的数组供我们保存到模型成员变量textures中。

loadMaterialtextures函数遍历指定纹理类型的所有纹理的文件路径，然后加载并生成纹理，保存到Texture结构中。其内容如下所示：

```
std::vector<Texture> Model::loadMaterialtextures(aiMaterial* mat, aiTextureType type, std::string typeName) {
    std::vector<Texture> textures;
    for (int i = 0; i < mat->GetTextureCount(type); ++i) {
        aiString str;
        mat->GetTexture(type, i, &str);
        Texture texture;
        texture.id = TextureFromFile(str.C_Str(), directory);
        texture.type = typeName;
        texture.path = str.C_Str();
        textures.push_back(texture);
    }
    return textures;
}
```

我们首先通过材质类本身的GetTextureCount函数获取了指定纹理类型的纹理数。然后，通过GetTexture函数取得纹理的路径，存放到临时变量str中。接着，调用辅助函数TextureFromFile加载纹理并返回纹理ID。TextureFromFIle函数的实现和我们之前加载纹理的实现相同，如果还是没有头绪，可以查看之后的完整源代码。

> 请注意：我们假设纹理文件的路径和模型文件的路径是相同的。所以我们将模型路径保存起来备用，如果路径不同，则需要换一种使用的方式。

等等，这就够了吗？？？

加载纹理可不是那么廉价的操作，上面的代码会对检索到的所有纹理加载生成一遍，即使这个纹理已经被加载了很多次。这很快会成为模型加载的性能瓶颈！聪明的你可能已经想到，纹理是不是可以复用？没错，纹理是可以复用的。所以，接下来我们的工作，就是加载纹理前先搜索已经加载的纹理，如果没加载过就加载，加载过了直接保存。

我们在Model类中添加一个私有成员变量 `textures_loaded` ：

```
std::vector<Texture> textures_loaded;
```

对loadMaterialtextures函数做一下修改，每次读取材质中的纹理图片路径，都从textures_loaded中找一遍，如果找到了，就直接拷贝，没找到的话再加载读取：

```
std::vector<Texture> Model::loadMaterialtextures(aiMaterial* mat, aiTextureType type, std::string typeName) {
    std::vector<Texture> textures;
    for (int i = 0; i < mat->GetTextureCount(type); ++i) {
        aiString str;
        mat->GetTexture(type, i, &str);

        bool skip = false;
        for (int j = 0; j < textures_loaded.size(); ++j) {
            if (std::strcmp(textures_loaded[j].path.c_str(), str.C_Str()) == 0) {
                textures.push_back(textures_loaded[j]);
                skip = true;
                break;
            }
        }

        if (!skip) {
            Texture texture;
            texture.id = TextureFromFile(str.C_Str(), directory);
            texture.type = typeName;
            texture.path = str.C_Str();
            textures.push_back(texture);
            textures_loaded.push_back(texture);
        }
    }

    return textures;
}
```

好了，现在我们拥有了一个可以加载模型的类，现在就去使用吧。完整的源代码请查看这里。

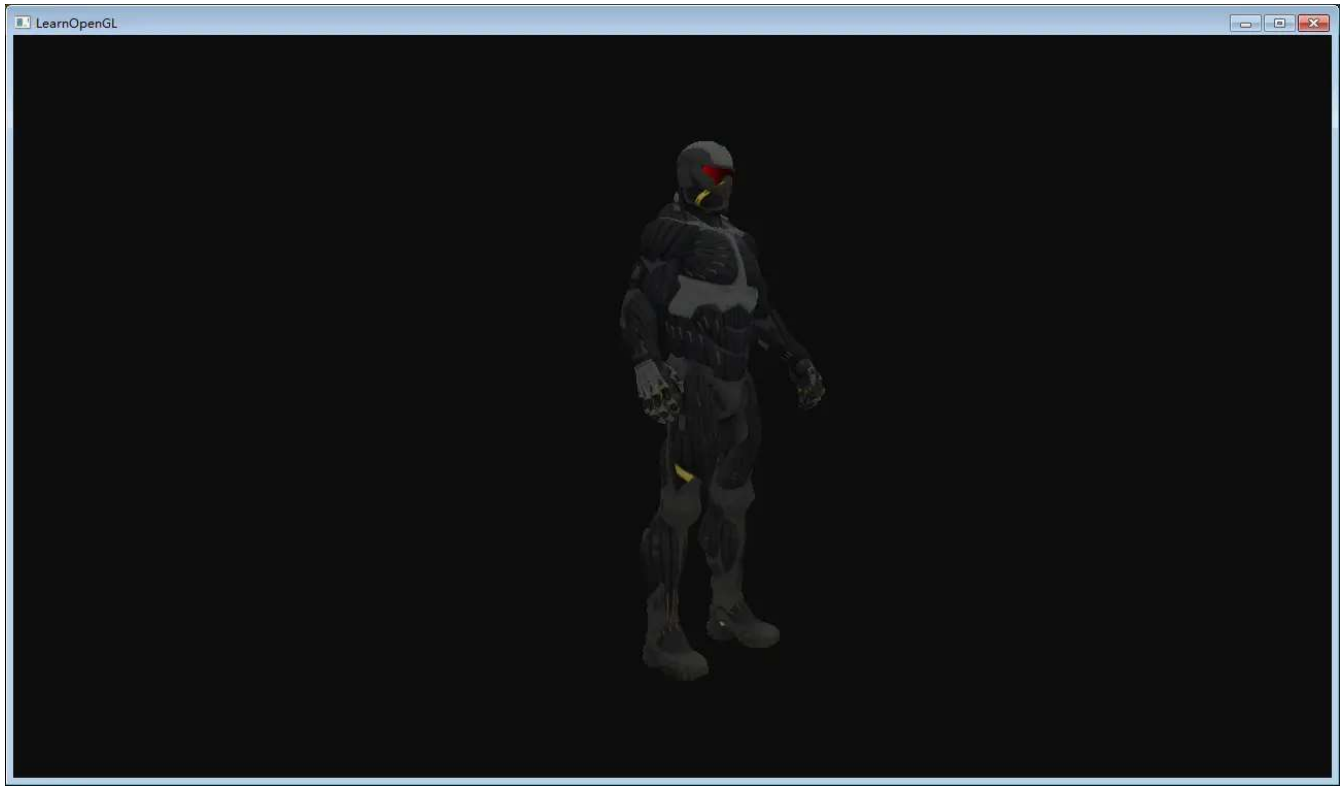
酷炫模型登场

终于，我们可以加载一个真正的模型而不是由某个不知名的艺术家弄出来的盒子兄弟（你得承认，我们盒子兄弟也是很酷的！）。去网上弄了一个模型来，这是孤岛危机中的战斗服，酷的一比！

模型示意图

在笔者提供的模型下载链接中，你下载到的是经过一些小小改动的模型包。材质文件和模型文件在同一级的目录下面方便程序读取。

有了模型资源之后，我们就可以加载了。加载的方法很简单，只要一行代码，'Model model(模型路径



显示效果

显示效果

如果有问题，请参考完整的源码

总结

在本章中，我们没有学什么OpenGL的新知识，但完成了一件早就想做却迟迟没做的事：模型加载。我们用了一个开源的模型加载库Assimp，创建了自己的顶点、纹理、网格和模型数据结构，并且将加载成Assimp结构的数据转换成自定义的数据结构，经过测试，我们的数据结构完全可用，太棒了！

下一篇
目录
上一篇

参考资料

www.learnopengl.com（非常好的网站，建议学习）