网格

原文	Mesh (http://learnopengl.com/#!Model-Loading/Mesh)
作者	JoeyDeVries
翻译	Meow J
校对	暂未校对

通过使用Assimp，我们可以加载不同的模型到程序中，但是载入后它们都被储存为Assimp的数据结构。我们最终仍要将这些数据转换为OpenGL能够理解的格式，这样才能渲染这个物体。我们从上一节中学到，网格(Mesh)代表的是单个的可绘制实体，我们现在先来定义一个我们自己的网格类。

首先我们来回顾一下我们目前学到的知识，想想一个网格最少需要什么数据。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）。

既然我们有了一个网格类的最低需求，我们可以在OpenGL中定义一个顶点了：

```
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`Texture`结构体中。

```
struct Texture {
    unsigned int id;
    string type;
};
```

我们储存了纹理的id以及它的类型，比如是漫反射贴图或者是镜面光贴图。

知道了顶点和纹理的实现，我们可以开始定义网格类的结构了：

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

你可以看到这个类并不复杂。在构造器中，我们将所有必须的数据赋予了网格，我们在`setupMesh`函数中初始化缓冲，并最终使用`Draw`函数来绘制网格。注意我们将一个着色器传入了`Draw`函数中，将着色器传入网格类中可以让我们在绘制之前设置一些uniform（像是链接采样器到纹理单元）。

构造器的内容非常易于理解。我们只需要使用构造器的参数设置类的公有变量就可以了。我们在构造器中还调用了`setupMesh`函数：

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

    setupMesh();
}
```

这里没什么可说的。我们接下来讨论`setupMesh`函数。

初始化

由于有了构造器，我们现在有一大列的网格数据用于渲染。在此之前我们还必须配置正确的缓冲，并通过顶点属性指针定义顶点着色器的布局。现在你应该对这些概念都很熟悉了，但我们这次会稍微有一点变动，使用结构体中的顶点数据：

```
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);

    // 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(V
ertex, Normal));
    // 顶点纹理坐标
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(V
ertex, TexCoords));

    glBindVertexArray(0);
}
```

代码应该和你所想得没什么不同，但有了`Vertex`结构体的帮助，我们使用了一些小技巧。

C++结构体有一个很棒的特性，它们的内存布局是连续的(Sequential)。也就是说，如果我们将结构体作为一个数据数组使用，那么它将会以顺序排列结构体的变量，这将会直接转换为我们在数组缓冲中所需要的float（实际上是字节）数组。比如说，如果我们有一个填充后的`Vertex`结构体，那么它的内存布局将会等于：

```
Vertex vertex;
vertex.Position  = glm::vec3(0.2f, 0.4f, 0.6f);
vertex.Normal    = glm::vec3(0.0f, 1.0f, 0.0f);
vertex.TexCoords = glm::vec2(1.0f, 0.0f);
// = [0.2f, 0.4f, 0.6f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f];
```

由于有了这个有用的特性，我们能够直接传入一大列的Vertex结构体的指针作为缓冲的数据，它们将会完美地转换为glBufferData所能用的参数：

```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STAT
IC_DRAW);
```

自然sizeof运算也可以用在结构体上来计算它的字节大小。这个应该是32字节的（8个float * 每个4字节）。

结构体的另外一个很好的用途是它的预处理指令offsetof(s, m)，它的第一个参数是一个结构体，第二个参数是这个结构体中变量的名字。这个宏会返回那个变量距结构体头部的字节偏移量(Byte Offset)。这正好可以用在定义glVertexAttribPointer函数中的偏移参数：

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Verte
x, Normal));
```

偏移量现在是使用offsetof来定义了，在这里它会将法向量的字节偏移量设置为结构体中法向量的偏移量，也就是3个float，即12字节。注意，我们同样将步长参数设置为了Vertex结构体的大小。

使用这样的一个结构体不仅能够提供可读性更高的代码，也允许我们很容易地拓展这个结构。如果我们希望添加另一个顶点属性，我们只需要将它添加到结构体中就可以了。由于它的灵活性，渲染的代码不会被破坏。

渲染

我们需要为Mesh类定义最后一个函数，它的Draw函数。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。所以我们该如何在着色器中设置纹理单元和采样器呢？

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。比如说我们对某一个网格有3个漫反射纹理，2个镜面光纹理，它们的纹理采样器应该之后会被调用：

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

根据这个标准，我们可以在着色器中定义任意需要数量的纹理采样器，如果一个网格真的包含了（这么多）纹理，我们也能知道它们的名字是什么。根据这个标准，我们也能在一个网格中处理任意数量的纹理，开发者也可以自由选择需要使用的数量，他只需要定义正确的采样器就可以了（虽然定义少的话会有点浪费绑定和uniform调用）。

> 像这样的问题有很多种不同的解决方案。如果你不喜欢这个解决方案，你可以自己想一个你自己的解决办法。

最终的渲染代码是这样的：

```
void Draw(Shader shader)
{
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i); // 在绑定之前激活相应的纹理单元
        // 获取纹理序号（diffuse_textureN 中的 N）
        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++);

        shader.setFloat(("material." + name + number).c_str(), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }
    glActiveTexture(GL_TEXTURE0);

    // 绘制网格
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```

我们首先计算了每个纹理类型的N-分量，并将其拼接到纹理类型字符串上，来获取对应的uniform名称。接下来我们查找对应的采样器，将它的位置值设置为当前激活的纹理单元，并绑定纹理。这也是我们在`Draw`函数中需要着色器的原因。我们也将`"material."`添加到了最终的uniform名称中，因为我们希望将纹理储存在一个材质结构体中（这在每个实现中可能都不同）。

> 注意我们在将漫反射计数器和镜面光计数器插入`stringstream`时，对它们进行了递增。在C++中，这个递增操作：`variable++`将会返回变量本身，**之后**再递增，而`++variable`则是**先**递增，再返回值。在我们的例子中是首先将原本的计数器值插入`stringstream`，之后再递增它，供下一次循环使用。

你可以在这里 (https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/mesh.h) 找到`Mesh`类的完整源代码

我们刚定义的`Mesh`类是我们之前讨论的很多话题的抽象结果。在下一节 (../03 Model/)中，我们将创建一个模型，作为多个网格对象的容器，并真正地实现Assimp的加载接口。
