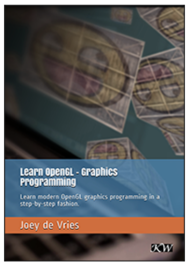
Learn OpenGL - Graphics Programming
Learn modern OpenGL graphics programming in a step-by-step fashion.
Joey de Vries

Mesh

With Assimp we can load many different models into the application, but once loaded they're all stored in Assimp's data structures. What we eventually want is to transform that data to a format that OpenGL understands so that we can render the objects. We learned from the previous chapter that a mesh represents a single drawable entity, so let's start by defining a mesh class of our own.

Let's review a bit of what we've learned so far to think about what a mesh should minimally have as its data. A mesh should at least need a set of vertices, where each vertex contains a position vector, a normal vector, and a texture coordinate vector. A mesh should also contain indices for indexed drawing, and material data in the form of textures (diffuse/specular maps).

Now that we set the minimal requirements for a mesh class we can define a vertex in OpenGL:

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

We store each of the required vertex attributes in a struct called `Vertex`. Next to a `Vertex` struct we also want to organize the texture data in a `Texture` struct:

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

We store the id of the texture and its type e.g. a diffuse or specular texture.

Knowing the actual representation of a vertex and a texture we can start defining the structure of the mesh class:

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

        Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Textur
        void Draw(Shader &shader);
    private:
        //  render data
        unsigned int VAO, VBO, EBO;

        void setupMesh();
};
```

As you can see, the class isn't too complicated. In the constructor we give the mesh all the necessary data, we initialize the buffers in the `setupMesh` function, and finally draw the mesh via the `Draw` function. Note that we give a shader to the `Draw` function; by passing the shader to the mesh we can set several uniforms before drawing (like linking samplers to texture units).

The function content of the constructor is pretty straightforward. We simply set the class's public variables with the constructor's corresponding argument variables. We also call the `setupMesh` function in the constructor:

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

    setupMesh();
}
```

Nothing special going on here. Let's delve right into the `setupMesh` function now.

Initialization

Thanks to the constructor we now have large lists of mesh data that we can use for rendering. We do need to setup the appropriate buffers and specify the vertex shader layout via vertex attribute pointers. By now you should have no trouble with these concepts, but we've spiced it up a bit this time with the introduction of vertex data in structs:

```
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],

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

    // vertex positions
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    // vertex normals
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offset
    // vertex texture coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offset

    glBindVertexArray(0);
}
```

The code is not much different from what you'd expect, but a few little tricks were used with the help of the `Vertex` struct.

Structs have a great property in C++ that their memory layout is sequential. That is, if we were to represent a struct as an array of data, it would only contain the struct's variables in sequential order which directly translates to a float (actually byte) array that we want for an array buffer. For example, if we have a filled `Vertex` struct, its memory layout would be equal to:

```
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];
```

Thanks to this useful property we can directly pass a pointer to a large list of `Vertex` structs as the buffer's data and they translate perfectly to what `glBufferData` expects as its argument:

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

Naturally the `sizeof` operator can also be used on the struct for the appropriate size in bytes. This should be 32 bytes (8 floats * 4 bytes each).

Another great use of structs is a preprocessor directive called `offsetof(s,m)` that takes as its first argument a struct and as its second argument a variable name of the struct. The macro returns the byte offset of that variable from the start of the struct. This is perfect for defining the offset parameter of the `glVertexAttribPointer` function:

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(V
```

The offset is now defined using the `offsetof` macro that, in this case, sets the byte offset of the normal vector equal to the byte offset of the normal attribute in the struct which is 3 floats and thus 12 bytes.

Using a struct like this doesn't only get us more readable code, but also allows us to easily extend the structure. If we want another vertex attribute we can simply add it to the struct and due to its flexible nature, the rendering code won't break.

Rendering

The last function we need to define for the `Mesh` class to be complete is its `Draw` function. Before rendering the mesh, we first want to bind the appropriate textures before calling `glDrawElements`. However, this is somewhat difficult since we don't know from the start how many (if any) textures the mesh has and what type they may have. So how do we set the texture units and samplers in the shaders?

To solve the issue we're going to assume a certain naming convention: each diffuse texture is named `texture_diffuseN`, and each specular texture should be named `texture_specularN` where `N` is any number ranging from `1` to the maximum number of texture samplers allowed. Let's say we have 3 diffuse textures and 2 specular textures for a particular mesh, their texture samplers should then be called:

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

By this convention we can define as many texture samplers as we want in the shaders (up to OpenGL's maximum) and if a mesh actually does contain (so many) textures, we know what their names are going to be. By this convention we can process any amount of textures on a single mesh and the shader developer is free to use as many of those as he wants by defining the proper samplers.

There are many solutions to problems like this and if you don't like this particular solution it is up to you to get creative and come up with your own approach.

The resulting drawing code then becomes:

```
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); // activate proper texture unit before
        // retrieve texture number (the N in diffuse_textureN)
        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.setInt(("material." + name + number).c_str(), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }
    glActiveTexture(GL_TEXTURE0);

    // draw mesh
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```

We first calculate the N-component per texture type and concatenate it to the texture's type string to get the appropriate uniform name. We then locate the appropriate sampler, give it the location value to correspond with the currently active texture unit, and bind the texture. This is also the reason we need the shader in the `Draw` function.

We also added `"material."` to the resulting uniform name because we usually store the textures in a material struct (this may differ per implementation).

Note that we increment the diffuse and specular counters the moment we convert them to `string`. In C++ the increment call: `variable++` returns the variable as is and **then** increments the variable while `++variable` **first** increments the variable and **then** returns it. In our case the value passed to `std::string` is the original counter value. After that the value is incremented for the next round.

You can find the full source code of the `Mesh` class here.

The `Mesh` class we just defined is an abstraction for many of the topics we've discussed in the early chapters. In the next chapter we'll create a model that acts as a container for several mesh objects and implements Assimp's loading interface.