MESH

What we want to eventually do, after loading the models we want into the application (which we'll do in the next chapter), is transform the data stored in Assimp's data structures to a format that OpenGL understands so we can render the objects.

We'll begin by defining our own mesh class.

What is the minimal data a mesh should have?

- A set of vertices, where each vertex contains a position vector, a normal vector, and a texture coordinate vector.
- Indices for indexed drawing.
- Material data in the form of textures (diffuse/specular maps).

Create a Mesh class header file that contains a constructor and functions to store mesh data. Then, implement the constructor.

- First, create a struct for storing vertex data and a struct for storing texture data, as they are defined above. Then, think of what functions you might need to initialize these variables, and what other variables might be required.
- Mesh Class Initial Setup

Initialization

Implement the setupMesh function.

- Set up the appropriate buffers and specify the vertex data layout (vertex attributes).
- Setting Up Mesh Buffers and Attributes

Rendering

Implement the Draw function.

- Iterate through each texture and bind it to its own texture unit. Set up the material struct in the fragment shader like this: New Material Struct
- Drawing the Mesh

Mesh Class Initial Setup

Thursday, April 14, 2022 2:42 PM

In the constructor, we give the mesh all the necessary data.
The `vao`, `vbo`, and `ebo` variables are initialized in the setupMesh function.
The `Draw` function draws the mesh.

- **NOTE:** We passed a shader to the `Draw` function so that we can set uniforms before drawing (e.g. linking samplers to texture units).

Mesh.h

```
#pragma once

#include <string>
#include <vector>

#include <glm/glm.hpp>
#include <assimp/Importer.hpp>

#include "Shader.h"

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

struct Texture {
   unsigned int id;
   std::string type; // diffuse or specular
};

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

   Mesh(std::vector<Vertex> verts, std::vector<unsigned int> inds, std::vector<Texture> texs);

   void Draw(Shader& shader);

private:
   // render data
   unsigned int vao, vbo, ebo;

   void setupMesh();
};
```

In the constructor, we simply set the class's public variables with the constructor's corresponding argument variables and then call the `setupMesh` function.

Mesh.cpp

```
Mesh::Mesh(std::vector<Vertex> verts, std::vector<unsigned int> inds, std::vector<Texture> texs)
   : vertices(verts), indices(inds), textures(texs) {
   setupMesh();
}
```

Setting Up Mesh Buffers and Attributes

Thursday, April 14, 2022 3:20 PM

Essentially, we need to generate the buffers, bind them (in the appropriate order to setup the VAO), initialize each buffer object's data, and describe the layout of the vertex data.

```
void Mesh::setupMesh() {
   glGenVertexArrays(1, &vao);
   glGenBuffers(1, &vbo);
   glGenBuffers(1, &ebo);

   glBindVertexArray(vao);

   glBindBuffer(GL_ARRAY_BUFFER, vbo);
   // You must pass a reference to the first index of the array, and not a reference to the array itself. Otherwise, you will get a
   // segmentation fault.
   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);

   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
   glEnableVertexAttribArray(0);
   glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
   glEnableVertexAttribArray(1);
   glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
   glEnableVertexAttribArray(2);

   glBindVertexArray(0);
   glBindBuffer(GL_ARRAY_BUFFER, 0);
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

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

New Material Struct

Thursday, April 14, 2022 4:14 PM

```
struct Material {
   sampler2D texture_diffuse1;
   sampler2D texture_diffuse2;
   sampler2D texture_diffuse3;
   sampler2D texture_specular1;
   sampler2D texture_specular2;

   float shininess;
};
```

NOTE: If you remember from a previous chapter, OpenGL has a (typical) maximum of 16 different texture units per texture object, so if you wanted to use 16 diffuse/specular maps per object, you very well could.

Drawing the Mesh

Thursday, April 14, 2022 3:32 PM

This implementation stores the `name` from the textures member variable and stores the `number` based on the `diffuseNum` or `specularNum` value, respectively, which is incremented whenever its respective texture name is found in `textures`.
The `uniform` of the `sampler2D` for the diffuse/specular texture is then located based on the values above, and set based on the current iteration (which represents the texture unit).
At the end of the loop, we bind the texture to the active texture unit and continue processing the rest of the textures.
Finally, we reset the active texture unit back to the 0th texture unit (VERY IMPORTANT) and draw the mesh.

```
void Mesh::Draw(Shader& shader) {
   int diffuseNum = 0;
   int specularNum = 0;

   // Binds all the textures to their own texture units and sets the respective uniforms in the fragment shader
   for (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(++diffuseNum);
       else if (name == "texture_specular")
          number = std::to_string(++specularNum);

       glUniform1i(glGetUniformLocation(shader.id, ("material." + name + number).c_str()), i);

       glBindTexture(GL_TEXTURE_2D, textures[i].id);
   }

   // Draws the mesh
   glBindVertexArray(vao);
   glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
   glBindVertexArray(vao);

   // Resets the active texture unit
   glActiveTexture(GL_TEXTURE0);
}
```