OpenGL Skeletal Animation With ASSIMP Tutorial part 2 — Mesh & Bone classes (part 2.1 - Mesh class)

Before the actual implementation of **skeletal animation**, firstly we have to understand how to keep the **Meshes** and the **Bones** data. I will split this tutorial into two parts: **Mesh class** and **Bone class**.

Mesh is the set of vertices and textures. More exactly mesh is a set of triangles (polygons) that are constructed from those vertices, chosen in a particular order. The set of meshes gives us a **Model**. Usually models are made in a special programs like Blender, Maya, 3Ds Max.

You can imagine **Bones** as sticks inside the model, where the position of the stick is dependent on time. Each bone has an array of **vertices** which it pulls or each vertex has the bones to follow. Moreover, each bone that the vertex follows has it's strength/weight. It basically means that the bone with less weight will be followed less than the bone with higher weight.

On the picture you can see the bones movement. A little curvature of the bones inside the caterpillar's body also curves the mesh.

Lets start with the implementation of **vertex** and **texture structures**:

```
#define BONES_AMOUNT 6

struct Vertex
{
   vec3 position;
   vec3 normal;
   vec2 texCoords; //UV coordinates

   float boneIDs[BONES_AMOUNT] = {0.0f}; //bones that are followed by this vertex
   float weights[BONES_AMOUNT] = {0.0f}; //strength/weight of the above bones
};

struct Texture
```

```
{
   unsigned int id; //opengl id
   string type; //type diffuse/specular
   string path; //path to the texture
};
```

BONES_AMOUNT is a constant that defines a maximum amount of bones that are used by a vertex.

Vertex:

-position (in local space)

-normal (needed for lights purposes, but we don't have lights in our program)

-texCoords (to map the texture)

-boneIDs array (we will define the ID of each bone in the **ModelLoader class** in the future tutorial and attach the Vertex to the bone via it's ID)

-weight array (the strength/weight of the bones. Of course if the bone is followed by more than one vertex it will most likely have different weights in each of them)

Texture:

-id (generated in OpenGL)

-type (needed for lights purposes, but we don't have lights in our program. We are only using diffuse textures to make the model visible)

-path (path to the texture/image)

And here is the **Mesh class**:

```
class Mesh
{
   private:
      GLuint VAO;
      GLuint VBO, EBO;

      vector < Vertex > vertices; //vector of vertices this mesh has
      vector < GLuint > indices; //vector of indices for EBO
      vector < Texture > textures; //textures this mesh has

      void setupMesh(); //here we generate VAO, VBO, EBO
```

```
  public:
    Mesh (vector < Vertex > &v, vector < unsigned int > &i, vector < Texture > &t);

    void draw(ShaderLoader *shader); //here we render the mesh to the given shader

    ~Mesh();
};
```

Mesh:

-VAO, VBO, EBO (we will generate the vertex array and the object buffers for each mesh)

-vectors (keep all loaded data from the model file)

-setupMesh() (here we generate OpenGL buffers for the loaded data)

-constructor (we pass loaded mesh data from the **ModelLoader class** here)

-draw() (here we render the mesh to the shader. Pass shader to draw into, check loadshader.hpp file)

And the realization:

```
Mesh::Mesh (vector < Vertex > &v, vector < unsigned int > &i, vector < Texture > &t)
{
  vertices = v; //copy vertex data
  indices = i; //copy index data
  textures = t; //copy textures data

  setupMesh(); //call setup
}
```

The constructor is simple, just copying the loaded from model file data to the local storage.

```
void Mesh::setupMesh()
{
  glGenVertexArrays(1, &VAO); //generate VAO
  glGenBuffers(1, &VBO); //generate VBO
  glGenBuffers(1, &EBO); //generate EBO

  glBindVertexArray(VAO); //bind VAO

  glBindBuffer(GL_ARRAY_BUFFER, VBO); //bind VBO
  glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.size(), &vertices[0],
GL_STATIC_DRAW); //insert vertex data into VBO
```

```
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO); //bind EBO
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * indices.size(), &indices[0],
GL_STATIC_DRAW); //insert index data into EBO

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    glEnableVertexAttribArray(0); //0 layout for position

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, normal));
    glEnableVertexAttribArray(1); //1 layout for normal

    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
texCoords));
    glEnableVertexAttribArray(2); //2 layout for UV

    for (int i = 0; i < BONES_AMOUNT; i++)
    {
        glVertexAttribPointer(3 + i, 1, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(offsetof(Vertex,
boneIDs) + sizeof(float) * i));
        glEnableVertexAttribArray(3 + i); //from 3 to 8 (BONES_AMOUNT) layouts for bones ids. Array in
shader uses N layouts, equal to the size, instead of single layout location the vec uses
    }

    for (int i = 0; i < BONES_AMOUNT; i++)
    {
        glVertexAttribPointer(9 + i, 1, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(offsetof(Vertex,
weights) + sizeof(float) * i));
        glEnableVertexAttribArray(9 + i); //from 9 to 14 for the weights
    }

    glBindVertexArray(0); //unbind
}
```

The setupMesh function needs a bit of explanation. Firstly, we generate VAO, VBO and EBO. Secondly, we fill those buffers with data. VBO gets vertices, EBO – indices. Then we configure the passing-into-shader data and match it with shader's layout locations. Note that we are using offsetof() function, it is only possible to do like that if we keep vertex data in a structure. Next step is to match the bones data with shader's inputs. Because inputs in the shader for the bones data are arrays, each their element will use it's own layout location, so we have to come up with some little tricky loops to solve that problem.

```
void Mesh::draw(ShaderLoader *shader)
{
    unsigned int diffuseNR = 1; //amount of diffuse textures
    unsigned int specularNR = 1; //amount of specular textures
```

```
    for (int i = 0; i < textures.size(); i++) //loop through textures
    {
        glActiveTexture(GL_TEXTURE0 + i); //set the texture active

        string number; //this one is needed if we use more than one texture of the same type

        if (textures[i].type == "texture_diffuse") //if it is a diffuse texture
        {
            number = to_string(diffuseNR); //this is the diffuseNR`s texture
            diffuseNR++; //amount of diffuse + 1
        }
        else if (textures[i].type == "texture_specular")
        {
            number = to_string(specularNR); //this is the specularNR`s texture
            specularNR++; //amount of specular + 1
        }

        glUniform1i(glGetUniformLocation(shader->ID, ("material." + textures[i].type + number).c_str()),
i); //send the texture to the shader (example: material.texture_diffuse1)
        glBindTexture(GL_TEXTURE_2D, textures[i].id); //bind this texture
    }

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

    glBindVertexArray(0); //unbind VAO
    glBindTexture(GL_TEXTURE_2D, 0); //unbind textures
}
```

The draw function may be confusing at the first look but there is nothing difficult there. However, there are some tricks that need explanation. The first trick is in passing textures into the shader.

There is a structure in the shader:

```
struct Material
{
    sampler2D texture_diffuse1;
    sampler2D texture_specular1;
    float shininess;
};

uniform Material material;
```

There we keep textures information of the current rendering **Mesh**. Now our program supports only two types of textures: diffuse and specular and only in a single instance (you can add the support of as many you want). So when

passing material data we only need to define the type of the texture and it's number.

The second trick is in tracking the textures. We loop through each texture in the mesh and remember it's parameters (type and ordinal number). Then we update the textures counter variable and throw the texture into the shader with given parameters.

And the last step is to draw the mesh.

```
Mesh::~Mesh(){}
```