OpenGL Skeletal Animation With ASSIMP Tutorial part 4 — ModelLoader class

Finally, we reached the point where we have entirely implemented the skeletal animation part. In this tutorial we will write **ModelLoader class** to import animated models from the files kept on the hard drive to our in-program data structure (mesh and skeleton classes) then to render the model to the screen and observe it's beautiful animation.

ASSIMP is a very popular model importing library that can import a lot of different model file formats. ASSIMP organizes imported model's information in its own data structure which is formed in a such way that importing different formats doesn't affect its setup.

Here is a diagram of the part of ASSIMP data structure that is used in this tutorial series:

red arrows show the object reference; green arrows show equality or an index reference

-ASSIMP imports every possible model data into the **aiScene** object. From there we have an access to all other objects in the structure. **aiScene** keeps an array of **aiMeshes**, a reference to the **root aiNode**, an array of **aiAnimations** and an array of **aiMaterials.**

-aiMesh keeps model's graphical information including bones: **mVetices[]** keep vertices 3D space coordinates, **mNormals[]** keep vertices normals, **mFaces[]** keep indices (a proper order) of vertices to take to construct triangles from, **mTextureCoords[]** keep vertices UV coordinates, **mBones[]** keep meshe's bones information, **mMaterialIndex** is an index of a material (set of tectures) in the mMaterials[] array in the aiScene object.

-**aiBone** keeps mesh's bone data, excluding keyframe data. Here are kept: **mOffsetMatrix –** bone's offset matrix, **mWeights[]** where indices of the vertices (mVertices[] in aiMesh) with their weight coefficient are kept (the ones that the bone pulls), **mName –** the name of the bone (there is an aiNode with the same name in order to identify bone's parent and an

aiNodeAnim with the same name to find bone's keyframe data).

-**aiNode** is something like a bridge in the ASSIMP data structure. From here we construct bones and meshes hierarchy and identify their keyframe data. **aiNode** keeps an array of aiNodes **– mChildren[]**, **mParent –** a reference to the parent node, **mMeshes[]** array – indices of the meshes in the mMeshes[] array in the aiScene object, **mName –** the name of the node (there also are an aiBone and an aiNodeAnim with same names in order to track bones parents and associate keyframes to the bones), **mTransformation matrix –** transformation matrix of the node (bone).

-**aiAnimation** is bones animation object, it's main task is to keep every bone keyframe data: **mName –** the name of the animation, an array of **aiNodeAnims** (in fact those keep bones keyframe data).

-**aiNodeAnim** is an animation node of the particular bone, we need this node to find and assign keyframes to the bones. It keeps: **mNodeName –** the name of the node (each aiNodeAnim is associated with a particular bone via their identical names), **mPositionKeys[]**, **mRotationKeys[]**, **mScalingKeys[] -** arrays of keyframes (each keyframe in arrays has it's associated time and value – 3D vector or quaternion).

Now I will present you the **ModelLoader class** and we will talk how to actually parse the ASSIMP data structure:

```
class ModelLoader
{
   private:
      vector < Bone* > bones; //here we load bones data
      vector < Mesh* > meshes; //here we load meshes data
      vector < Texture > textures_loaded; //here we keep the loaded textures to optimize resources
      string directory; //this is the directory where the model is kept
      Skeleton *skeleton; //our skeleton which is constructed from the bones

      Importer import; //assimp`s importer
      const aiScene* scene; //here assimp stores all the loaded data

      vector < aiNode* > nodes; //we need this vector to store important information into bones in future.
Actually in aiNode the transformation matrix of the particular bone is kept
      vector < aiNodeAnim* > nodesAnim; //here the transformation animation data of the particular bone
is kept
```

```
        void processNode(aiNode *node); //this method is needed to fill the nodes vector above
        void processNodeAnim(); //this method is needed to fill the nodesAnim vector above
        void processBone(); //here we extract bones data from assimp into our format
        void processMeshes(aiNode *node); //extract meshes data
        Mesh* processMesh(aiMesh *mesh); //extract the single mesh data

        vector < Texture > loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName); //
extract textures data
        unsigned int textureFromFile(const char* path); //load textures from file

        Bone* findBone(string name); //find the bone via it`s name
        int findBoneId(string name); //find bone`s index in bones vector via it`s name
        aiNode* findAiNode(string name); //find aiNode via it`s name
        aiNodeAnim* findAiNodeAnim(string name); //find aiNodeAnim via it`s Name

    public:
        ModelLoader();

        void loadModel(string path); //call this to load the model

        void getModelData(Skeleton *&skeleton, vector < Mesh* > &meshes); //this method is used to get the
loaded data and then to store it into the gameobject

        ~ModelLoader();
};
```

ModelLoader:

-bones vector (we keep loaded bones data here. We will loop through mMeshes[] array in the aiScene object and construct the bones from the mBones[] vector kept in the corresponding aiMesh)

-meshes vector (we keep loaded meshes data here. We recursively loop through nodes and take mesh's information from the associated aiMesh in the aiScene array with the index in the mMeshes[] array in the aiNode)

-textures_loaded (here we keep loaded from files textures. If the model needs the already loaded texture (when parsing materials), we take it from this vector instead of loading the texture again)

-directory (the path to the model's directory)

-skeleton (constructed skeleton from the loaded bones)

-import (ASSIMP's class to import models from files into it's data structure)

-scene (ASSIMP's class where it keeps everything. Have a look at the diagram above)

-nodes vector (we recursively fill this vector with aiNodes in order to

look for the nodes via their name faster)

-nodesAnim vector (loop through the aiAnimation and fill this vector with aiNodeAnims also to look for the nodes faster)

-processNode() (here we recursively fill the nodes vector with aiNodes)

-processNodeAnim() (here we loop through the aiAnimation and fill this vector with aiNodeAnims)

-processBone() (in this method we construct our bones from the data in the aiBone and define their parent bones)

-processMeshes() (here we construct our meshes vector by calling processMesh() function for each aiMesh and remembering the output)

-processMesh() (in this function we construct a single mesh from the given aiMesh data. We also add bones indices with their weights into the vertices they need to follow)

-loadMeterialTextures() (in this method we extract textures from materials the mesh uses)

-texturefromFile() (this function loads a texture from the file and returns its OpenGL id)

-findBone() (finds a bone via the given name, uses bones vector)

-findBoneId() (finds a bone's id via the given name, uses bones vector)

-findAiNode() (finds an aiNode via the given node, uses nodes vector)

-findAiNodeAnim() (finds an aiNodeAnim via the given name, uses nodesAnim vector)

-loadModel() (in this method to import a model via ASSIMP into its data structure and then call all other in-class functions to extract information into our data structure)

-getModelData() (we call this method from the **GameObject class** to take the meshes vector and skeleton class data. Those are constructed while parsing an ASSIMP data structure)

Of course, realization:

```
ModelLoader::ModelLoader(){}
```

An empty constructor...

```
void ModelLoader::loadModel(string path)
{
   scene = import.ReadFile(path, aiProcess_Triangulate); //assimp loads the file

   if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) //if
something gone wrong
   {
      cout << "ERROR::ASSIMP::" << import.GetErrorString() << endl; //error
      return;
   }

   directory = path.substr(0, path.find_last_of('/')); //get only the directory from the whole path to the file

   processNode(scene->mRootNode); //fill the nodes vector
   processNodeAnim(); //fill the nodesAnim vector
   processBone(); //get bones
   processMeshes(scene->mRootNode); //get meshes

   skeleton = new Skeleton(bones);
}
```

In the loadModel() function we import the model via ASSIMP and then parse it's data structure. Firstly, we call an ASSIMP importer to import the model into it's data structure and then check the data correctness. Secondly, we extract the directory from the model's path and call some functions to load up our data structure from the ASSIMP's information. Then we construct the skeleton object from the loaded bones data.

```
void ModelLoader::getModelData(Skeleton *&skeleton, vector < Mesh* > &meshes)
{
   skeleton = this->skeleton;
   meshes = this->meshes;
}
```

In the getModelData() method we just assign the pointers of the **GameObject class** objects with already loaded into the ModelLoader objects data.

```
void ModelLoader::processNode(aiNode *node)
{
   nodes.push_back(node);

   for (int j = 0; j < node->mNumChildren; j++) //recursive loop through each child to fill the vector
   {
      processNode(node->mChildren[j]);
   }
```

```
}
```

This method is needed to store all the nodes into the vector in order to look for them via the names faster. Here we loop recursively through the children of all nodes starting from the root node and push_back them into the nodes vector.

```
void ModelLoader::processNodeAnim()
{
   if (scene->mNumAnimations == 0) //if there are no animations
   {
      return;
   }

   //cout << scene->mNumAnimations << endl;

   for (int i = 0; i < scene->mAnimations[0]->mNumChannels; i++) //loop through the animation to get
each bone animation data
   {
      nodesAnim.push_back(scene->mAnimations[0]->mChannels[i]);
   }
}
```

This function stores keyframes of the first animation into the vector in order to assign them to the bones faster. Here we check if there are any animations available for the mesh and then loop through the first one to extract aiNodeAnims (keyframes for the bones). Note that our program only supports single animated models, thats why we loop only through the first element in the mAnimations array. If you want to support multi animated models, you need to consider each element of the mAnimations array.

```
void ModelLoader::processBone()
{
   for (int i = 0; i < scene->mNumMeshes; i++) //loop through each mesh of the model
   {
      for (int j = 0; j < scene->mMeshes[i]->mNumBones; j++) //loop through each bone which is
connected to that mesh
      {
         string boneName = scene->mMeshes[i]->mBones[j]->mName.data; //get bone`s name
         mat4 boneOffset = aiMatrix4x4ToGlm(scene->mMeshes[i]->mBones[j]->mOffsetMatrix); //get
bone`s offset matrix. This is the matrix which transforms from the mesh space to the bone space

         Bone* bone = new Bone(bones.size(), boneName, boneOffset); //make new bone from the data we
got (index, name, offset)

         bone->node = findAiNode(boneName); //each bone has its node with same name
```

```
            bone->nodeAnim = findAiNodeAnim(boneName); //same with the animation node

            if (!bone->nodeAnim) //if there is no animations for the bode. This often happens with root bones
            {
                cout << "ERROR::NO ANIMATIONS FOUND FOR " << boneName << endl;
            }

            bones.push_back(bone); //push back the bone
        }
    }

    for (int i = 0; i < bones.size(); i++) //here we are looking for the parent bone for our bones
    {
        string parentName = bones[i]->node->mParent->mName.data; //get the bone parent name. For each
aiNode assimp keeps the parent pointer, as the bone has the same name as it`s aiNode we can do like that

        Bone* parentBone = findBone(parentName); //find the parent bone by it`s name

        bones[i]->parentBone = parentBone; //set the parent bone for the bone

        if (!parentBone) //if there is no parent bone
        {
            cout << "NO PARENT BONE FOR " << bones[i]->name << endl;
        }
    }

    //done
}
```

In the method above we are loading bones information into the bones vector. We loop through mMeshes[] array in the aiScene and then through the mBones[] array in the aiMesh in the nested loop. Then we get the name of the current bone and it's offset matrix and construct the **bone** object from this data. After that we define the aiNode and aiNodeAnim for the given bone via it's name (we use in-class findAiNode() and findAiNodeAnim() functions. Corresponding nodes have same names as the bone's name). Then we check whether there are any animations available for the bone and if no, return an error and just before the loop ending, we push_back the bone to the bones vector. Then in the next loop we are looking for the parents for our bones. We loop through the bones vector and get the bone's parent name (we get the bone's parent name from the parent node name, which child is the given bone's node. Because the names of the bone and the node are identical, we just use the node's name instead). Then we find the bone via its name (using findBone() function) and set the bone's parent pointer to the

found bone. In the last step we check the parent's existence.

```
void ModelLoader::processMeshes(aiNode *node)
{
   //we start from the root node
   for (int i = 0; i < node->mNumMeshes; i++) //loop through each mesh this node has
   {
      aiMesh *mesh = scene->mMeshes[node->mMeshes[i]]; //get the mesh from it`s index
      meshes.push_back(processMesh(mesh)); //here we call another function to extract the data and then
push_back the complete mesh
   }

   for (int j = 0; j < node->mNumChildren; j++) //recursive loop through the each node
   {
      processMeshes(node->mChildren[j]);
   }
}
```

In this function we loop recursively through the mNodes[] and through the corresponding mMeshes[] defined via the mMeshes[] indices kept inside the aiNode and call the processMesh() function for each aiMesh to load the mesh data into the our format. Then we push_back the output mesh into the **meshes** vector.

Note that the next function is really huge and in order to achieve code explanation readability I will split it into the parts.

```
Mesh* ModelLoader::processMesh(aiMesh *mesh)
{
   vector < Vertex > vertices; //meshes vertices (position, normal, texture coords, bone ids, bone weights)
   vector < unsigned int > indices; //vertex indices for EBO
   vector < Texture > textures; //textures (id, type, path)

   for (int i = 0; i < mesh->mNumVertices; i++) //loop through each vertex in the mesh
   {
      Vertex vertex;
      vec3 helpVec3;

      helpVec3.x = mesh->mVertices[i].x; //positions
      helpVec3.y = mesh->mVertices[i].y;
      helpVec3.z = mesh->mVertices[i].z;

      vertex.position = helpVec3; //set position


      helpVec3.x = mesh->mNormals[i].x; //normals
      helpVec3.y = mesh->mNormals[i].y;
```

```
        helpVec3.z = mesh->mNormals[i].z;

        vertex.normal = helpVec3; //set normal


        if (mesh->mTextureCoords[0]) //if there is a texture
        {
            vec2 helpVec2;

            helpVec2.x = mesh->mTextureCoords[0][i].x; //textures coords
            helpVec2.y = mesh->mTextureCoords[0][i].y;

            vertex.texCoords = helpVec2; //set texture coords
        }
        else
        {
            vertex.texCoords = vec2(0.0f, 0.0f);
        }

        vertices.push_back(vertex);
    }
```

The first part is for loading graphical (vertex) information of the mesh. In the function's beginning we define **vertices**, **indices** and **textures** vectors where we will store the mesh's information. Then we loop through mVertices[], mNormals[] and mTexturecoords[] arrays kept in the given aiMesh and extract vertices positions, normals vectors and UV coordinates from those arrays. We construct the **vertex** from the given data and push_back it into the vertices vector.

```
    //we have loaded the vertex data, its time for the bone data

    for (int i = 0; i < mesh->mNumBones; i++) //loop through each bone that mesh has
    {
        aiBone* bone = mesh->mBones[i];

        for (int j = 0; j < bone->mNumWeights; j++) //loop through the each bone`s vertex it carries in that
mesh
        {
            aiVertexWeight vertexWeight = bone->mWeights[j]; //get the id and the weight of the carried vertex

            int startVertexID = vertexWeight.mVertexId; //get the carried vertex id

            for (int k = 0; k < BONES_AMOUNT; k++) //BONES_AMOUNT is a constant that is for the
maximum amount of the bones per vertex
            {
                if (vertices[startVertexID].weights[k] == 0.0) //if we have an empty space for one more bone in
the vertex
```

```
            {
                vertices[startVertexID].boneIDs[k] = findBoneId(bone->mName.data); //set bone index to the
vertex it carries

                vertices[startVertexID].weights[k] = vertexWeight.mWeight; //set bone weight/strength to the
vertex it carries

                break; //only one place for the single bone
            }

            if (k == BONES_AMOUNT - 1) //if we have more that enough bones
            {
                //cout << "ERROR::LOADING MORE THAN " << BONES_AMOUNT << " BONES\n"; //
this could take a lot of time
                break;
            }
        }
    }
}
```

In the second part we load bones information into the vertices (the indices of the bones which they need to follow; those bones weights). Firstly, we loop through the mBones[] array in the given aiMesh and through the mWeights[] array kept in the corresponding aiBone in the nested loop. Secondly, we extract vertices ids and weights from the data kept in the mWeights[] array. Thirdly, we look for an empty spot for the given bone id in the found vertex and if there is one, store there corresponding data there. If there are no empty spots (this means that the model we are loading has more than BONES_AMOUNT bones for the vertex to follow) we output an error.

```
//we have loaded everything for the vertices including bones, loading indices

for (int i = 0; i < mesh->mNumFaces; i++) //loop through the mesh`s planes/faces
{
    aiFace face = mesh->mFaces[i];

    //if face.mNumIndices == 3 we are loading triangles
    for (int j = 0; j < face.mNumIndices; j++)
    {
        indices.push_back(face.mIndices[j]);
    }
}

//we have loaded indices, loading materials

aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex]; //get the material by it`s index
```

```
    vector < Texture > diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE,
"texture_diffuse"); //calling another function to load diffuse textures
    textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end()); //instead of the loop we can
insert vector like that

    vector < Texture > specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR,
"texture_specular"); //calling another function to load specular textures. We don't have lights in this app,
but if you want to add some...
    textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());

    return new Mesh(vertices, indices, textures);
}
```

In the last part we load vertices' indices and textures. Firstly, we loop through the mFaces[] array in the given aiMesh and take indices from there (indices are the correct order of the vertices to construct triangles from). Secondly, we take the material index, kept in the aiMesh, and define the corresponding material object taking it from the aiScene. Then we load and insert diffuse and specular textures (by calling loadMetrialTextures() function) into the local textures vectors. In the last two steps we copy textures information from the local vectors into the main textures vector and return the newly constructed **Mesh** pointer.

```
vector < Texture > ModelLoader::loadMaterialTextures(aiMaterial *mat, aiTextureType type, string
typeName)
{
    vector < Texture > textures; //vector for the particular texture type

    for (int i = 0; i < mat->GetTextureCount(type); i++) //loop through each typed texture
    {
        aiString helpStr;

        mat->GetTexture(type, i, &helpStr); //get texture`s name

        bool skip = false;

        for (int j = 0; j < textures_loaded.size(); j++) //loop through already loaded textures
        {
            if (strcmp(textures_loaded[j].path.data(), helpStr.C_Str()) == 0) //if we have already loaded it
            {
                textures.push_back(textures_loaded[j]); //use the loaded one instead of loading the new one

                skip = true;
                break;
            }
        }
```

```
        if (!skip) //if the texture is new
        {
            Texture texture;

            texture.id = textureFromFile(helpStr.C_Str()); //calling another function to load texture from file

            texture.type = typeName; //set the texture`s type

            cout << "Texture type: " << typeName << endl;

            texture.path = helpStr.C_Str(); //set the texture`s path

            textures.push_back(texture); //take the texture
            textures_loaded.push_back(texture); //remember the texture
        }
    }

    return textures;
}
```

In the method above we load textures of a given type into the **textures** vector. Firstly, we loop through the textures of a given type in the current material and get their name. Secondly, we check whether we have already loaded the texture with the same name and if yes, just push_back it into the local textures vector again, but if no, call the textureFromFile() function to load it from the file, define it's type and it's path and then to insert the loaded texture into the **textures** and **textures_loaded** vectors. And in the last step we return the newly constructed textures vector.

```
unsigned int ModelLoader::textureFromFile(const char *path)
{
    string filename = string(path); //convert to string
    filename = directory + '/' + filename; //concatenate strings to get the whole path

    unsigned int textureID;
    glGenTextures(1, &textureID); //gen texture, opengl function
    glBindTexture(GL_TEXTURE_2D, textureID); //bind the texture

    int W, H; //width and height
    unsigned char* image = SOIL_load_image(filename.c_str(), &W, &H, 0, SOIL_LOAD_RGBA); //using
SOIL to load the RGBA image

    if (image) //if the image is fine
    {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, W, H, 0, GL_RGBA, GL_UNSIGNED_BYTE,
image); //fill the texture with image data
        glGenerateMipmap(GL_TEXTURE_2D); //generate mipmaps
```

```
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        SOIL_free_image_data(image); //free the memory
    }
    else
    {
        cout << "Fail to load texture at path: " << path << endl;
        SOIL_free_image_data(image);
    }

    return textureID;
}
```

The function above loads texture from the file into the OpenGL and returns it's id. We are using **SOIL library** to carry the actual loading. Firstly, we concatenate the directory and texture name strings to get the full path to the texture. Secondly, we generate OpenGL textures and call SOIL function to load the texture into the char* array. Thirdly, we check the loaded data correctness and if everything is fine, send the texture into the OpenGL, generate mipmaps and define some texture's parameters. But if the data is corrupted, we output an error. In the end we clear up the SOIL memory and return the texture's id.

```
Bone* ModelLoader::findBone(string name)
{
    for (int i = 0; i < bones.size(); i++) //just loop through the array of bones
    {
        if (bones[i]->name == name) //if name matches the name
        {
            return bones[i]; //return bone
        }
    }

    return 0;
}
```

In this method we are looking for the bone via it's name. We loop through the bones array and check the names coincidence (the given name and the bone's name). If names match, we return the current bone. If we didn't find any bones, we return 0.

```
int ModelLoader::findBoneId(string name)
{
   for (int i = 0; i < bones.size(); i++) //just loop through the array of bones
   {
      if (bones[i]->name == name) //if name mathes the name
      {
         return bones[i]->id; //return id
      }
   }

   return -1;
}
```

In this method we are looking for the bone's id via it's name. We loop through the bones array and look for the bone's name to match with the given name. If we found one, return the bone's id. If we didn't find any, return -1.

```
aiNode* ModelLoader::findAiNode(string name)
{
   for (int i = 0; i < nodes.size(); i++) //loop through the array of nodes
   {
      if (nodes[i]->mName.data == name) //if node name mathes the name
      {
         return nodes[i]; //return node
      }
   }

   return 0;
}
```

In this method we are looking for the aiNode with the same name as the given one. Just looping through the nodes vector, checking the name and returning the node if there was a match. If didn't find any, we return 0.

```
aiNodeAnim* ModelLoader::findAiNodeAnim(string name)
{
   for (int i = 0; i < nodesAnim.size(); i++) //loop through the array of animation nodes
   {
      if (nodesAnim[i]->mNodeName.data == name) //if animation node name matches the name
      {
         return nodesAnim[i]; //return animation node
      }
   }

   return 0;
}
```

In this method we are looking for the aiNodeAnim. We loop through the nodesAnim vector, check the name difference and return the node. If we didn't find any nodes, we return 0.

```
ModelLoader::~ModelLoader(){}
```