OpenGL Skeletal Animation With ASSIMP Tutorial part 2 — Mesh & Bone classes (part 2.2 - Bone class)

In the previous tutorial we wrote **Mesh class** where we keep only graphical information of the model**.** In this one we will figure out how to write **Bone class** to store bones information.

As it was mentioned earlier, you can imagine bones as sticks inside the model, where the position of the stick is dependent on time. When constructing the Model, artist adds bones into it and defines their position in particular moments. Those moments are called **keyframes**. **Keyframe** usually keeps bone's scaling, translation **vectors** and rotation **quaternion** in order to construct it's transformation matrix. So, as the time goes on, we update bone's transformation matrix from the corresponding keyframe data, then we take the vertices it pulls and multiply them by the given matrix considering bone's weight. This process moves the vertices, which, as a result, move the model. It is called **skeletal animation**.

There is one more thing you need to know about **skeletal animation**. Imagine you needed to animate human body in all details: arms movement, fingers flexure... Of course if an arm moves fingers also move, but not vice versa, if fingers move arms stand still. This fact gives us a hint how to organize our bones structure – in a **tree** (in this example an arm is a parent for the fingers). From that fact another important fact flows. In order to properly make the skeletal animation we have to construct the transformation matrices properly. If we combine the transformation matrices of all parents of a given bone with it's keyframe data we will get the proper given bone's transformation matrix.

Here is the **Bone class** declaration:

```
class Bone
{
  public:
```

```
	string name; //the name of the bone
	int id; //bone's id in the model

	aiNode* node; //the node this bone is attached to, we keep transformation matrix here
	aiNodeAnim* nodeAnim; //this bone`s animation node, keyframe data is here

	Bone* parentBone; //parent bone
	mat4 offset; //offset matrix

	//keyframe data. This data is calculated in a particular period of time
	vec3 pos;
	quat rot;
	vec3 scal;


	Bone(int id, string name, mat4 &offset);

	mat4 getParentTransforms(); //calculates the whole transformation matrix starting from the root bone

	vec3 calcInterpolatedPosition(float time); //calculates interpolated position between two keyframes
	quat calcInterpolatedRotation(float time); //calculates interpolated rotation between two keyframes

	int findPosition(float time); //finds the index of the keyframe with the value less than time
	int findRotation(float time); //same as above

	void updateKeyframeTransform(float time); //updates this bone transformation matrix

	~Bone();
};
```

Bone:

-name (we need it to find the bone's parent bone. This is made in the **ModelLoader class** in the future tutorial)

-id (just to know the index/number of the the bone in the model)

-node (I will explain node's belonging to the bone in the **ModelLoader class** in the future tutorial. For now you only need to know that we will use it just to keep bone's transformation matrix)

-nodeAnim (each bone has it's aiNodeAnim in ASSIMP. There all bone's keyframe data is kept)

-parentBone (each bone has a parent bone it is attached to, except the root bone)

-offset (this matrix transforms from the local model space into the bone space, needed to make the animation correctly)

-pos, rot, scal (in those ones we keep time-corresponding keyframe data)

-constructor (it accepts: bone's id, bone's name, it's offset matrix. This data is loaded inside the **ModelLoader class** and then passed to the constructor)

-getParentTransforms() (this function returns mat4 – the combination of all parents' transformation matrices)

Before the next functions usage explanation, there is one more thing about the keyframes you have to understand. Keyframes keep bone's transformation data to construct the transformation matrix from it. For each keyframe particular moment of time is associated, that means that we only have limited amount of them and constructing transformation matrices from the closest-to-our-in-program-time one will give an intermittent and rough animation. There are two ways to achieve the smooth one: ask an artist to define more keyframes and still take the closest or to calculate the **interpolated** value between the **two** closest keyframes. We will implement the second method because of it's versatility.

-calcInterpolatedPosition() (this function takes time and returns vec3 - interpolated value between the closest-to-the-time **position** keyframes)

-calcInterpolatedRotation() (this function takes time and returns quaternion – interpolated value between the closest-to-the-time **rotation** keyframes)

-findPosition(), findRotation() (takes time and returns the index of the closest less-than-time keyframe)

-updateKeyframeTransform() (this function updates the bone's transformation matrix from the given time)

And here is the step-by-step realization:

```
Bone::Bone(int id, string name, mat4 &offset)
{
   this->id = id; //set id
   this->name = name; //set name
   this->offset = offset; //set offset matrix
```

```
    //null pointers
    parentBone = 0;
    node = 0;
    nodeAnim = 0;
}
```

In constructor we just copy the loaded in the **ModelLoader class** data to the local storage.

```
mat4 Bone::getParentTransforms()
{
    Bone* b = parentBone; //this bone`s parent

    vector < mat4 > matrices; //vector to remember matrices

    while (b) //loop through all parents to push_back all transformation matrices
    {
        mat4 tmp = aiMatrix4x4ToGlm(b->node->mTransformation); //get the transformation matrix
        matrices.push_back(tmp); //remember it

        b = b->parentBone; //next parent
    }

    mat4 res; //all parent transformation matrix

    for (int i = matrices.size() - 1; i >= 0; i--) //loop backward! The last element of the array is the root bone
    {
        res *= matrices[i]; //here we calculate the combined transformation matrix
    }

    return res;
}
```

In the getParentTransforms() function we calculate the combined transformation matrix of all parents of the given bone. We loop through all parents of the bone (the parents are already defined in the **ModelLoader class**) then we remember the transformation matrix taken from the corresponding aiNode (this data is not static, in the updateKeyframeTransform() function we calculate new transformation matrix and store it into the same aiNode) and then jump to the parent's parent… Because we have looped from the given bone down to the root bone, we have to combine matrices in a reverse order – from the root bone up to the given bone's parent.

```
vec3 Bone::calcInterpolatedPosition(float time)
```

```
{
   vec3 out; //result

   if (nodeAnim->mNumPositionKeys == 1) //if amount of keyframes == 1 we dont have to interpolate
anything
   {
      aiVector3D help = nodeAnim->mPositionKeys[0].mValue; //get its value/position vector

      out = vec3(help.x, help.y, help.z);

      return out;
   }

   //if there is more than one keyframe
   int positionIndex = findPosition(time); //find index of the keyframe
   int nextPositionIndex = positionIndex + 1; //the next index

   float deltaTime = (float)(nodeAnim->mPositionKeys[nextPositionIndex].mTime - nodeAnim-
>mPositionKeys[positionIndex].mTime); //calculate time difference between keyframes
   float factor = (time - (float)nodeAnim->mPositionKeys[positionIndex].mTime) / deltaTime; //calculate
how far we are between keyframes in the interval of [0;1]

   aiVector3D begin = nodeAnim->mPositionKeys[positionIndex].mValue; //get the first keyframe value
   aiVector3D end = nodeAnim->mPositionKeys[nextPositionIndex].mValue; //get the second one

   vec3 p1(begin.x, begin.y, begin.z);
   vec3 p2(end.x, end.y, end.z);

   out = mix(p1, p2, factor); //using glm function to mix the values dependent on factor

   return out;
}
```

The name of the above function says everything about it's purpose. Here we calculate and return interpolated position vector. Firstly, we check whether there is only one keyframe and if so, return it's position value. If there are more than one keyframes, we get the index of the closest one with the less-than-given-time value and the keyframe index after it. Then we calculate time difference between two found keyframes and convert the given time into the interval of [0;1]. So, the **factor** (variable in the code) is a value between 0 and 1 that says how far we are from the left keyframe. And the last step is to interpolate the value between two keyframes with the factor as a parameter. We are using GLM function to do this.

```
quat Bone::calcInterpolatedRotation(float time)
{
   quat out; //result
```

```
    if (nodeAnim->mNumRotationKeys == 1) //if amount of keyframes == 1 we dont have to interpolate
anything
    {
        aiQuaternion help = nodeAnim->mRotationKeys[0].mValue; //get it`s value/rotation quaternion

        out = quat(help.w, help.x, help.y, help.z);
        return out;
    }

    //if there is more than one keyframe
    int rotationIndex = findRotation(time); //find index of the keyframe
    int nextRotationIndex = rotationIndex + 1; //the next index

    float deltaTime = (float)(nodeAnim->mRotationKeys[nextRotationIndex].mTime - nodeAnim-
>mRotationKeys[rotationIndex].mTime); //calculate time difference between keyframes
    float factor = (time - (float)nodeAnim->mRotationKeys[rotationIndex].mTime) / deltaTime;  //calculate
how far we are between keyframes in the interval of [0;1]

    aiQuaternion begin = nodeAnim->mRotationKeys[rotationIndex].mValue;
    aiQuaternion end = nodeAnim->mRotationKeys[nextRotationIndex].mValue;

    quat r1(begin.w, begin.x, begin.y, begin.z);
    quat r2(end.w, end.x, end.y, end.z);

    //up to this point everything is the same as for position

    out = slerp(r1, r2, factor); //here we use another glm function to mix the quternion values dependent on
factor

    return out;
}
```

This function is absolutely identical to the calcInterpolatedPosition() function, except two things. The first one, is that we are using rotation keyframes instead of position ones and the second is in different GLM function to interpolate rotation values.

Position data is kept in 3d vector but rotation data is kept in quaternion and to calculate the interpolated rotation value another GLM function is needed.

```
int Bone::findPosition(float time)
{
    for (int i = 0; i < nodeAnim->mNumPositionKeys - 1; i++) //loop through each position keyframe
    {
        if (time < nodeAnim->mPositionKeys[i + 1].mTime) //if the given time is less than in a keyframe
        {
            return i;
        }
    }

    return -1;
}

int Bone::findRotation(float time)
{
    for (int i = 0; i < nodeAnim->mNumRotationKeys - 1; i++) //loop through each rotation keyframe
    {
        if (time < nodeAnim->mRotationKeys[i + 1].mTime) //if the given time is less then in a keyframe
        {
            return i;
        }
    }

    return -1;
}
```

findPosition() and findRotation() functions are really simple, they return the index of the closest keyframe with the less-than-given-time value. We are looping through each keyframe this bone (aiNodeAnim) has and if we find one with greater time value than the given time, we return the index of the previous keyframe.

```
void Bone::updateKeyframeTransform(float time)
{
    if (!nodeAnim) //if there are no animations for this bone
    {
        return;
    }

    scal = vec3(1.0); //we dont support scaling animations
    pos = calcInterpolatedPosition(time); //calculate interpolated position
    rot = calcInterpolatedRotation(time); //calculate interpolated rotation

    mat4 res(1.0);

    //here we construct the transformation matrix in STR order
    res *= scale(scal);
    res *= translate(pos);
    res *= mat4_cast(rot);

    node->mTransformation = glmToAiMatrix4x4(res); //mTransformation is a public variable so we can
store there our new transformation matrix
}
```

And the last function in the **Bone class** – updateKeyframeTransform().

Firstly, we check whether there are any animations available for this bone. Then we calculate interpolated position and rotation (you can add support for scaling aswell, but keep in mind that most of the models do not scale while animated). The next step is to construct the transformation matrix. We are using some GLM functions to con
vert vectors and quaternion to matrices and then combine them into the final transformation matrix. And the last step is to update node's matrix.

If you remember, we are using node's transformation matrix inside the getParentTransform() function to correctly calculate the given bone's transformation matrix. So to close up the circle we have to basically update the node's matrix with the newly calculated one.

```
Bone::~Bone(){}
```