OpenGL Skeletal Animation With ASSIMP Tutorial part 3 — Skeleton & Animation classes
(part 3.1 - Skeleton class)

By that time we have written both **Mesh and Bone classes** where we keep every possible model data, but we still need some kind of a shell for the **Bone class** to actually start the skeletal animation from. I can't imagine any theory you need to understand the given class's code.

So lets dive into it!

```
#define MAX_BONES_AMOUNT 100

class Skeleton
{
   private:
      vector < Bone* > bones; //here we keep our bones
      vector < mat4 > bonesMatrices; //final transformation matrices that go to the shader

      float time; //little timer

      Animation* activeAnimation; //little Animation class to keep the custom animation data

      void renderBonesMatrices(ShaderLoader* shader); //update and send bones matrices to the shader

   public:
      Skeleton(vector < Bone* > &bones);

      void playAnimation(Animation* anim, bool reset = true); //play desired animation
      void stopAnimation(); //stop playing

      void update(ShaderLoader* shader); //here we update the animation and call renderBonesMatrices()

      ~Skeleton();
};
```

MAX_BONES_AMOUNT is a constant that defines a maximum amount of bones per model.

Skeleton:

-bones (we load all the bones in a model inside the **ModelLoader class** and then construct skeleton class, passing the bones vector to the constructor)

-bonesMatrices (in this vector we keep final transformation matrices of given bones: each bone's final matrix is a combination of all parents transformation matrices multiplied by it's offset matrix. Those matrices are sent to the shader)

-time (this is our custom per-model **frames** timer that we use to calculate bones transformation matrices from the time (passed frames) it shows. We can also adjust it to achieve framerate independent animation)

-activeAnimation (you will learn about this one in the next tutorial. But **Animation class** is unthinkably simple: we just come up with some random data about the animation. It's name, starting and ending frames indices, speed, priority and loop)

-renderBonesMatrices() (in this method we **get** and **pass** final transformation bones matrices to the given shader)

-constructor (just accepting the given bones to copy to the local storage)

-playAnimation() (here we accept an animation class instance to launch the skeleton animation starting from and ending with the frames indices kept inside the instance with the given speed that is also defined there)

-stopAnimation() (stop the animation if there is one playing)

-update() (here we call the renderBonesMatrices() function, update the timer and recalculate new bones transformation matrices relative to the time it shows)

And the realization:

```
Skeleton::Skeleton(vector < Bone* > &bones)
{
   this->bones = bones; //set bones

   time = 0; //time is 0

   activeAnimation = 0;
}
```

The constructor is as usually simple. There we copy bones vector to the local storage, set timer to 0 and set a nullpointer to activeAnimation pointer.

```
void Skeleton::playAnimation(Animation* anim, bool reset)
{
   if (activeAnimation) //if something is playing
   {
      if (anim->priority >= activeAnimation->priority) //if new animation has greater or equal priority
      {
         activeAnimation = anim; //change the animation
      }
   }
   else
   {
      activeAnimation = anim;
   }

   if (reset)
   {
      time = activeAnimation->startTime; //set timer to the beginning
   }
}
```

playAnimation() accepts **Animation class** instance and the boolean variable to indicate whether we want to reset the timer. Firstly, we check if there is an active, playing animation and if its priority is less or equal to the given instance we change the active animation to the new one. But if there is no active animation then we simply make the given instance an active one. Secondly, we check the reset variable and set the timer to the starting frame of the active animation if needed.

```
void Skeleton::stopAnimation()
{
   time = 0;
   activeAnimation = 0;
}
```

This function sets the timer to 0 and deactivates the activeAnimation by setting it's pointer to 0.

Before you read the next function code you need to have a look at the **vertex shader** code:

```
#version 330 core
```

```
#define BONES_AMOUNT 6
#define MAX_BONES_AMOUNT 100

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texture;
layout (location = 3) in float boneIDs[BONES_AMOUNT];
layout (location = 9) in float boneWeights[BONES_AMOUNT];

uniform mat4 localTransform;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform mat4 bones[MAX_BONES_AMOUNT];
uniform int meshWithBones;

out vec2 textureCoords;

void main()
{
   mat4 bonesTransform;

   if (meshWithBones == 1)
   {
      bonesTransform = mat4(0.0);

      for (int i = 0; i < BONES_AMOUNT; i++)
      {
         bonesTransform += bones[int(boneIDs[i])] * boneWeights[i];
      }
   }
   else
   {
      bonesTransform = mat4(1.0);
   }

   gl_Position = projection * view * model * localTransform * bonesTransform * vec4(position, 1.0);

   textureCoords = vec2(texture.x, -texture.y); //reverse textures
}
```

VertexShader:

-BONES_AMOUNT (a constant for the maximum amount of bones the vertex follows. We have the same constant defined in the **Mesh class** (mesh.hpp). So if you want to add the support of more bones per vertex you have to change this constant in both files and play with layout locations)

-MAX_BONES_AMOUNT (this constant if for a maximum amount of bones per model. The same constant is defined in **Skeleton class** (skeleton.hpp))

-layout locations (**0** for the position vector, **1** for the normal vector, **2** for UV coordinates, from **3** to **8** for bones ids, from **9** to **14** for bones weights)

-localTransform (you will know about this one in the **GameObject class** tutorial)

-model, view, projection (transformation matrices)

-bones (the actual bones matrices)

-meshWithBones (this variable is needed to identify whether this mesh has bones or not)

-textureCoords (reversed UV coordinates)

In the main() function we check if there are any bones in the mesh and calculate the combined transformation matrix (bonesTransform in the code) by looping through each bone index in the given vertex, taking the corresponding bone and multiplying it by it's weight. But if there aren't any bones in the mesh, we leave the combined matrix as a unit matrix. Then we calculate the final position of a vertex and reverse texture coordinates.

Now you are ready.

```
void Skeleton::renderBonesMatrices(ShaderLoader* shader)
{
   if (!bones.empty()) //if there are some bones in the mesh
   {
      glUniform1i(glGetUniformLocation(shader->ID, "meshWithBones"), 1); //say the shader that there are some bones in the mesh

      bonesMatrices.clear(); //clear the values

      for (int i = 0; i < MAX_BONES_AMOUNT; i++) //loop through 100 bones
      {
         if (i >= int(bones.size())) //if we have run out of bones
         {
            bonesMatrices.push_back(mat4(1.0)); //there is no transformation matrix, so we leave it as 1.0
         }
         else
         {
```

```
            mat4 res = bones[i]->getParentTransforms() * aiMatrix4x4ToGlm(bones[i]->node->mTransformation); //calculate transformation matrix for our bone

            bonesMatrices.push_back(res * bones[i]->offset); //calculate full transformation matrix
        }

        glUniformMatrix4fv(glGetUniformLocation(shader->ID, ("bones[" + to_string(i) + "]").c_str()), 1, GL_FALSE, value_ptr(bonesMatrices[i])); //send the matrix to the shader
      }
   }
   else //if there arent any bones in the mesh
   {
      glUniform1i(glGetUniformLocation(shader->ID, "meshWithBones"), 0); //say the shader that there are no bones in the mesh
   }
}
```

Firstly, we check if there are any bones in the current mesh and if so, tell the shader that the mesh isn't empty. Then we loop through all 100 (MAX_BONES_AMOUNT) bones and calculate the final bone transformation matrix (combining all parent transformation matrices, multiplying the combination by the current bone transformation matrix and multiplying the result by the given bone
offset matrix) and send this final matrix to she shader. If we have run out of bones in the model we leave the final transformation matrix as a unit matrix. And if the mesh isn't bone-rigged we tell the shader that the mesh is empty.

```
void Skeleton::update(ShaderLoader *shader)
{
   renderBonesMatrices(shader); //send bones matrices to the shader

   if (!activeAnimation) //if there is no animation
   {
      return; //do nothing
   }

   time += activeAnimation->speed; //update the timer

   if (time < activeAnimation->startTime) //if time is less than starting time of the animation
   {
      time = activeAnimation->startTime; //set time to the start
   }

   if (time > activeAnimation->endTime) //if the time is grater than the end time
   {
      if (activeAnimation->loop) //if animation is looped
      {
```

```
            time = activeAnimation->startTime; //set time to the start
        }
        else
        {
            stopAnimation(); //stop animation
        }
    }

    for (int i = 0; i < bones.size(); i++) //loop through each bone
    {
        bones[i]->updateKeyframeTransform(time); //calculate it`s tranformation matrix
    }
}
```

In the update() function we update every possible bone data. Firstly, we call renderBonesMatrices() function to send bones to the shader. Secondly, we update the timer. If the timer exceeded the active animation's last frame, we check whether the animation is looped and if so, set timer to the animation's starting frame. Thirdly, we loop through each bone in the mesh and update it's keyframe (node) matrix.

```
Skeleton::~Skeleton()
{
    for (int i = 0; i < bones.size(); i++)
    {
        delete bones[i];
    }

    delete activeAnimation;
}
```