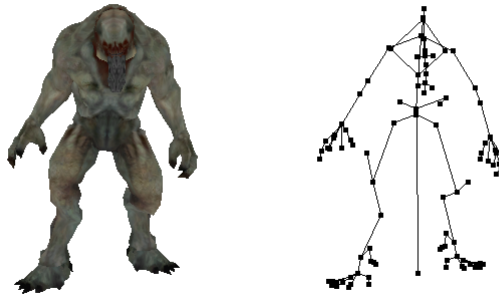# Tutorial 9: Skeletal Animation

Summary

In the last couple of tutorials, you've seen how to create a scene graph, and implemented a simple animating 'cube robot' using them. You're probably wondering how to get from simple animating characters like the cube robot, to the more advanced animating meshes seen in modern computer games. This tutorial will show you the theory behind *skeletal animation*, a way of realistically animating a mesh using a data structure similar to the scene graphs you've been using. You'll also see an implementation of skeletal animation, using a model from a commercial video game.

New Concepts

Skeletal Animation, skeletal joints, joint weights, quaternions

Skeletal Meshes

Modern video game characters area usually rendered by the movement of an underlying skeleton of *joints*, controlled by data loaded in from an animation file. This makes them much easier to control, and capable of much more interesting movement than the 'hard coded' animations we used for the CubeRobot.

Joints

The skeleton used to animate a skeletal mesh is made up of a number of joints. Each of these joints has a transformation consisting of a position and a rotation, and may have a parent joint its transform is in relation to. Sound familiar? Each joint is essentially the same thing as a scene node - if you look at the introduction image, you'll even spot a *transition* joint - just like our CubeRobot, some skeletal animation meshes will have a node at the local origin to aid in model placement.

The joints used in meshes don't necessarily directly replicate how 'real' joints are used in a human skeleton. In a game, a character's face might also be made up of joints - the sides of the mouth might be joints which can moved by an animation to make their lips move when talking. If you've ever seen a highly detailed game character who turned their eyes towards you, they did so due to having their eyeballs connected to 'joints'! Joints might also be used as 'attachment' points - for example a character holding a weapon in a game probably does so because the weapon mesh has been attached to a joint in the character's hand.

Skinning

If joints are just the same as scene nodes, then how exactly is skeletal animation better than the primitive animation we achieved with the CubeRobot? The problem with scene node animation is that every vertex is drawn in relation to only a single point of reference - the scene node itself. However, in skeletal animation, it is common to use a process known as *vertex skinning*, where each vertex position is built up from a number of points of references. This is useful for joints which rotate by a large amount. Take the elbow, for example. Image we wanted to build up a human model, complete with fully poseable arms. If we tried to simulate the elbow joint, we'd end up with something like this:
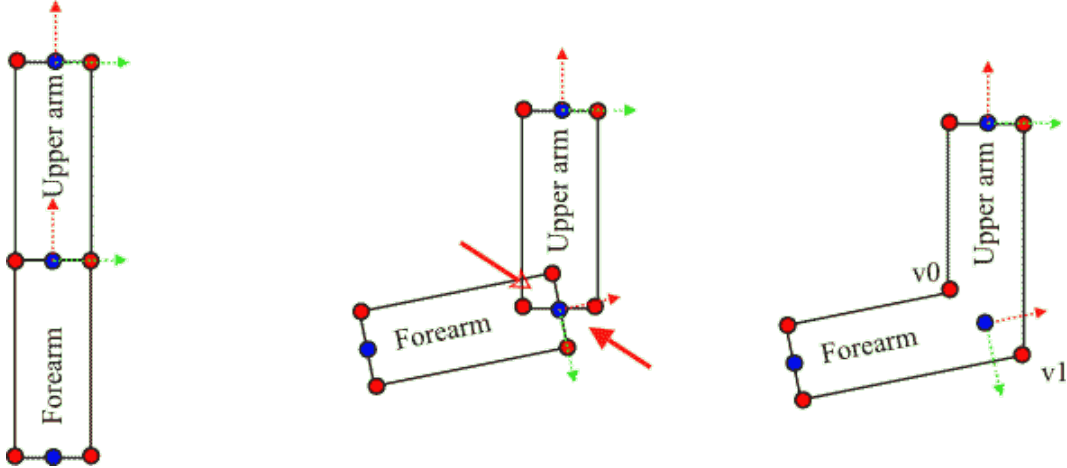
Example of vertex skinning. Middle image shows the overlaps and gaps that appear when rotating separate meshes, while right image shows how this can be avoided by using a single mesh and using multiple frames of reference for vertices (in this case v0 and v1)

If we animated this elbow joint, we'd end up with a 'hole' in the elbow - not very realistic! We could try to fill that hole by making the forearm joint area bigger, but we'd still end up with overlapping geometry, which if textured, would create an odd effect, like the clothing on the arm was actually disappearing inside the arm as the joint was rotated.

So instead, we have a single mesh for the entire arm (or even the entire model), which has each vertex drawn in relation to multiple frames of reference. There are two variations on using these points of references:

1) Each vertex has a position, and a number of joints the position is relative to.

2) Each vertex is relative to a number of 'anchoring' points, each of which has a position and a single joint it is relative to.
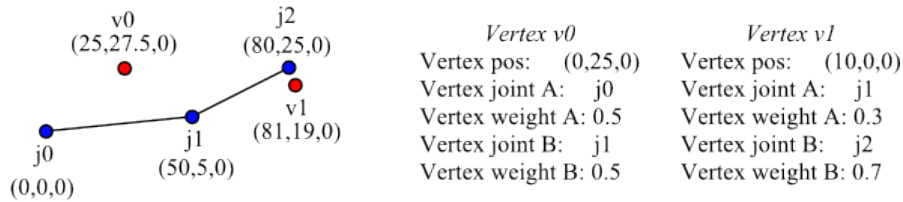
Weights

No matter which variation is used, the final position of the vertex is determined by its position in relation to a number of reference points. How much influence each reference point has on the final vertex position is determined by its *weighting*. Each vertex has a weight value for each reference point, which together add up to 1.0. These weight values are then used to scale the result of transforming an initial vertex position by a joint's transformation matrix.

$$position = \sum (t \cdot v) \cdot w$$

The final vertex position is formed by summing the results of multiplying each joint transform t *by either the vertex or weight position* v*, scaled by multiplying by joint weight value* w
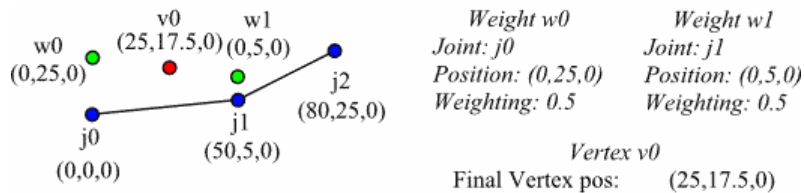
Here's some examples to show how a vertices' position is calculated using its reference points.

Firstly, here's an example using direct joint weightings. *v0* is drawn in reference to joints *j0* and *j1* equally, while *v1* is drawn in relation to *j1* and *j2*, with *j2* having more influence. In both cases the vertices local position is used, which in the case of *v1*, places it slightly further to the right than either of the joints it is weighted to.

|	
Vertex v0	*Vertex v1*
Vertex pos: (0,25,0)	Vertex pos: (10,0,0)
Vertex joint A: j0	Vertex joint A: j1
Vertex weight A: 0.5	Vertex weight A: 0.3
Vertex joint B: j1	Vertex joint B: j2
Vertex weight B: 0.5	Vertex weight B: 0.7

Direct vertex weightings

When using weighted anchor points, the final position is determined entirely by the weighting between the anchor weight positions. In this example, *v0* has two weight anchors - *w0* and *w1*, each with equal influence. This means that *v0* will end up exactly in the mid point between the joint-transformed positions of *w0* and *w1*.

Weight w0	*Weight w1*
Joint: j0	*Joint: j1*
Position: (0,25,0)	*Position: (0,5,0)*
Weighting: 0.5	*Weighting: 0.5*

Vertex v0
Final Vertex pos: (25,17.5,0)

Weighted anchor positions

Bind Pose

When a mesh is bound to its underlying skeleton (a process known as *rigging*) it is usually placed into a *bind pose*. For bipedal meshes, such as humans, or the HellKnight used in these examples, the bind pose is to have the mesh's legs slightly apart, and its arms out to either side. This is to make it easier to access individual vertices when modelling, and also to place the mesh in a fully 'stretched' state, making it easier to weight each vertex to the correct joints. When loading in a skeletal mesh, if no animation is applied to it, it will most likely be in this initial bind pose state.
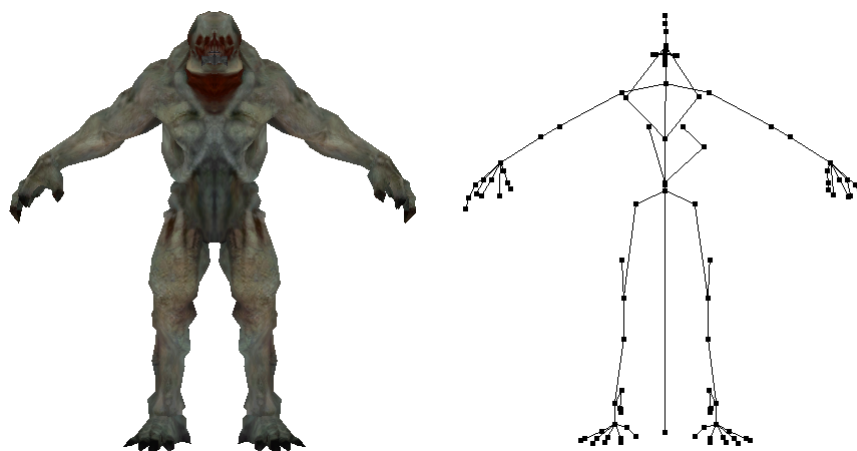
An example of a skeletal mesh and its underlying skeleton in its bind pose - note that the root node is transitional, and is between the character's feet to aid in placement

Animating Skeletal Meshes

Like the scene nodes you have been introduced to, each joint in a skeletal mesh may have a parent. The joint's position and orientation will always be in relation to that parent joint, so animating a skeletal mesh is simply a case of moving the skeleton joints to the desired orientation and position according to data loaded in from an animation file, and then reskinning the mesh in this new position. Like cartoons or movies, a skeletal mesh animation generally has a set number of frames per second. So if an animation has an internal frame rate of 24 frames a second, the mesh skeleton should be moved and the mesh reskinned every 24th of a second.

Using the framerate independence functionality you've already learned about, updating a mesh by a specific framerate should be easy but it's possible to go one step further, and create even smoother animations. If an example joint j has a transformation $t1$ in the first frame of animation, and a transformation of $t2$ in the second, it follows that at the time period between frames, the actual transformation of the joint will be somewhere between $t1$ and $t2$. Like the interpolation of vertex attributes, it's possible to interpolate smoothly between two frames of animation.

It's possible to go one step further than *that*, too. If one animation only animates the legs, say a running animation, it's possible to add another animation on top for the other joints - so there can be a separate 'running' animation and 'weapon reloading' animation that can be blended together, without having to create a 'running while reloading' animation for every weapon a character can hold.

Hardware Skinning

Joints are just transformation matrices, and weights are just floats and positions, so it is possible for skeletal mesh skinning to be performed on the graphics card, in a vertex shader. In fact doing so will probably be beneficial to the framerate of the application, as graphics hardware is faster at processing floating point calculations than the CPU, and the vertex data for the mesh to be skinned is probably in graphics memory anyway.

There are a variety of ways to store the skeletal transformation data on the graphics card - it could be sent as a large uniform array, or written as a texture and accessed via texture sampling. However the data stored and accessed from the graphics card, the vertices of the skeletal mesh will require more *attributes*. Depending on the variation of vertex skinning, these attributes may be a set of anchor positions and their weighting values and joint index, or maybe simply pairs of index values and weighting value. In either case, the vertex shader will require every joint's transformation, passed as a single large uniform array of matrices.

Due to how vertex attributes work, each vertex is required to have the same amount of data. But what if one vertex is influenced by one joint, and another by 4? Or 100? Generally, vertices to be transformed by hardware skinning are limited to being influenced by a low number of joints, commonly 4 or 8. Each vertex would then have 4/8 vertex skinning attributes, with weightings of 0.0 used for unused joint influence 'slots'. Similarly, there is a limit on the size of uniform data sent to the vertex shader, placing an upper limit on the number of joints that can be hardware skinned in a vertex shader via the uniforms you are used to using. For example, some hardware is limited to 256 vertex shader registers - that means only 256 vec4s can be sent to a vertex shader (remember, graphics hardware always works on 4 component vectors!). So that would place a hard limit of 64 joints sent to a vertex shader - 64 joint matrices · 4 vec4s per matrix = 256. Now, the 4th row of a joint matrix isn't *really* used - it'll always be (0,0,0,1). So we can cheat a little and reconstruct a joint's transformation matrix from 3 vec4s - now we can have 85 joints! However, there is a way of being able to send even more joints to a vertex shader as uniforms. A joints transformation matrix is made up of 1 vec4 for position, and a 3 by 3 matrix for rotation. There is a way of being able to 'compress' that rotation matrix into only 4 values, giving us enough space for 128 joints - *quaternions.*
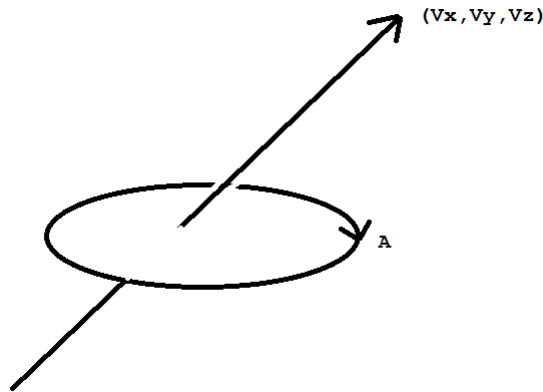
Quaternions

Any orientation in 3D space can be represented by an *axis*, and an *angle* of rotation around that axis - think of a vector pointing in a direction, with a twist. A quaternion consists of four coefficients (x, y, z, w) which are calculated as:

$$(x, y, z, w) = (\sin\ a\ /\ 2\ *\ V_x,\ \sin\ a\ /\ 2\ *\ V_y,\ \sin\ a\ /\ 2\ *\ V_z,\ \cos\ a\ /\ 2)$$

Where a is the angle of rotation around the axis (V_x, V_y, V_z).

Three dimensional orientation could also be represented by the use of three Euler angles (*pitch*, *yaw* and *roll*), which at first sight seems more efficient on memory, and may seem more intuitive to you. However, combining Euler angles is computationally expensive (trigonometry is not cheap!), and will also lead to *gimbal lock* (a situation where multiple axis' of a rotation point the same way, effectively removing a degree of freedom from further rotations). Working with quaternions resolves these disadvantages.

A quaternion can be thought of as an axis, and an angle of rotation around that axis

There are well-established algorithms for turning a quaternion into a rotation matrix, and vice versa the maths is pretty straightforward. Also, combining a series of rotations which are represented as quaternions is just a matter of multiplying the quaternions together (again the maths is straightforward and involves nothing more expensive that multiplications). Graphics libraries typically include optimised functions for working with quaternions.

So, we can keep the rotation as a quaternion, and reconstruct it in the vertex shader to a matrix, and set the joint's position as the translation matrix, giving us more joints at the cost of greater computation in reconstructing the matrix.

$$\begin{bmatrix} q_x \\ q_y \\ q_z \\ q_w \end{bmatrix} = \begin{bmatrix} 1 - 2 \cdot q_y^2 - 2 \cdot q_z^2 & 2 \cdot q_x \cdot q_y - 2 \cdot q_z \cdot q_w & 2 \cdot q_x \cdot q_z + 2 \cdot q_y \cdot q_w \\ 2 \cdot q_x \cdot q_y + 2 \cdot q_z \cdot q_w & 1 - 2 \cdot q_x^2 - 2 \cdot q_z^2 & 2 \cdot q_y \cdot q_z - 2 \cdot q_x \cdot q_w \\ 2 \cdot q_x \cdot q_z - 2 \cdot q_y \cdot q_w & 2 \cdot q_y \cdot q_z + 2 \cdot q_x \cdot q_w & 1 - 2 \cdot q_x^2 - 2 \cdot q_y^2 \end{bmatrix}$$

How to turn a quaternion q into a 3 by 3 rotation matrix

Example Program

To demonstrate skeletal animation, you're going to create an application that will load in and animate a mesh - a HellKnight from id Software's *Doom 3*. The *Doom 3* engine supports skeletal animation via the *MD5* format, a well documented, plain text file format. This program will load in the HellKnight's vertices from an *MD5Mesh* file, and an example animation from an *MD5Anim* file.

The example program for this tutorial is a little bit different, in that you aren't going to be writing the code for the skeletal animation - the code for loading in the *MD5Mesh* and *MD5Anim* file formats is just *too* tedious and error prone! Instead, we're going to add a few provided classes to the *nclgl* project, and take a look at how they implement the important parts of skeletal animation. The classes you'll be adding *are* fully commented though, so there's plenty to learn by looking through them!

There are 4 new classes involved in getting an animated MD5 mesh on screen - *MD5FileData* and *MD5Anim* load and store the structures described in an MD5 animated character's files, while *MD5Node* and *MD5Mesh* are subclasses of the *SceneNode* and *Mesh* classes you made earlier in the tutorial series.
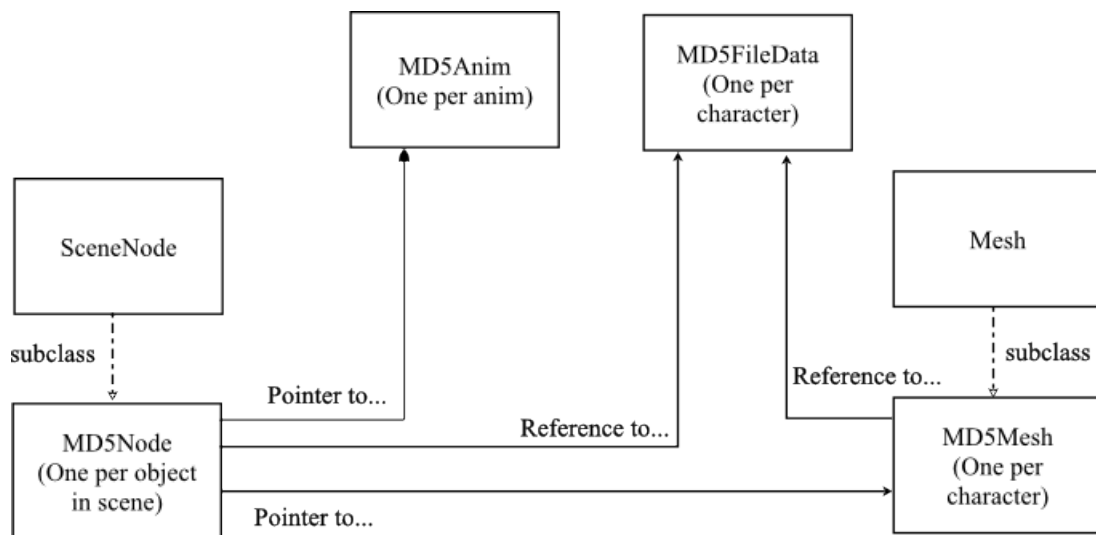
Class structure, and the relationships between each other and the classes you have already created

In the *./nclgl/* folder, you should find the class and header files for *MD5FileData*, *MD5Anim*, *MD5Node*, and *MD5Mesh* - add all the files to your *nclgl* static library project. The reason they area hidden away instead of being already in there is that the MD5 model format uses vertex indices extensively, and so the code quite simply would not compile until you programmed support for indices last tutorial!

MD5Node

Due to the space they take up, we don't really want multiple copies of the same MD5 mesh in memory - instead we're going to have only one, and then skin that one mesh into the correct pose for however many copies of that mesh we want in our game's world. In order to have multiple characters, all with their own animation and current skinned frame, we're going to derive a new subclass from the *SceneNode* class we created earlier. In it, we're going to have an instance of a special *skeleton* data structure, a reference to an MD5FileData, a **pointer** to the currently playing animation (if any), as well as an **unsigned int** to store the current animation frame, and a **float** to keep track of when to update the skeleton.

We also need some functions in our new node class - a *PlayAnim* function to set which animation the character should play, as well as overridden *Update* and *Draw* functions. That's all we need in order to control skeletal animation - most of the 'difficult' stuff will happen in our other classes.

```cpp
class MD5Node : public SceneNode {
public:
    MD5Node(const MD5FileData &ofType);
    ~MD5Node(void);

    virtual void    Update(float msec);
    virtual void    Draw(const OGLRenderer &r);

    void            PlayAnim(std::string name);    //Play a new anim!

protected:
    const MD5FileData&    sourceData;
    MD5Skeleton           currentSkeleton;
    MD5Anim*              currentAnim;

    float                 frameTime;
    unsigned int          currentAnimFrame; //Current frame of animation
};
```

MD5Node.h

The *Update* and *Draw* functions are very simple - *Update* will advance the current frame of animation according to how much time has passed since the last *Update*, and transform the skeleton accordingly. *Draw* will skin the mesh data to the correct pose for this *MD5Node*, and then draw it using the parent *Mesh* class *Draw* function.

```cpp
void  MD5Node::Update(float msec) {
    if(currentAnim && awake) {
        frameTime -= msec;

        while(frameTime < 0) {
            frameTime += 1000.0f / currentAnim->GetFrameRate();
            currentAnimFrame =
                currentAnimFrame++%(currentAnim->GetNumFrames());
        }

        currentAnim->TransformSkeleton(
                currentSkeleton,currentAnimFrame-1);
    }

    SceneNode::Update(msec);
}

void  MD5Node::Draw(const OGLRenderer &r) {
    MD5Mesh*m = (MD5Mesh*)mesh;
    m->SkinVertices(currentSkeleton);
    Mesh::Draw();
}
```

MD5Node.h

MD5Mesh

As described earlier, it is common to make the weighting amount and index information a vertex attribute. However, this data is quite specific information to skeletal animation, so there's not much point making it 'generic' vertex information inside our *Mesh* class - most meshes won't ever use it! So instead we subclass *Mesh*, and store a new OpenGL name for our vertex attribute data, and a pointer to the data itself. Our existing *BufferData* function won't touch this, so we make a new function, *BufferExtraData*, to handle the uploading of the vertex weighting into graphics memory. Finally, as we're going to be actually modifying the vertex data on the graphics card every time the mesh is reskinned into a new pose, which we'll handle with the *RebufferData* function.

```cpp
class MD5Mesh : public Mesh, public ChildMeshInterface   {
public:
   MD5Mesh(const MD5FileData&type);
   ~MD5Mesh(void);

   virtual void Draw();
   void        SkinVertices(const MD5Skeleton &skel);

protected:
   void   RebufferData();
   void   BufferExtraData();

   Vector2*      weights;
   GLuint        weightObject;

   const MD5FileData &  type;
};
```

MD5Mesh.h

Implementing MD5 skeletal mesh animation

The MD5 mesh format is a pretty straightforward implementation of skeletal animation, using the 'weights with anchor positions' variation. It has a skeleton made up of a list of joints, starting from the 'root' joint, stored as **structs**, like this:

```cpp
struct MD5Joint {
   int         parent;            //parent joint (0 for root)
   Vector3     position;          //position of this joint
   Quaternion  orientation;       //orientation of this joint
   Matrix4     transform;         //world transform for this joint
};

struct MD5Skeleton {
   int         numJoints;  //How many joints does this skeleton have?
   MD5Joint*   joints;     //Heap memory for numJoints MD5Joints
};
```

MD5FileData.h

It also has a list of weights, each of which has a position, weighting value, and a reference to the joint it is tied to. The vertices for an MD5 are stored as follows - note how they only have texture coordinate attributes, as they have their final position built up via a number of weighted anchors.

```cpp
12  struct MD5Weight {
13      int      weightIndex;        //index of weight from 0 - n
14      int      jointIndex;         //reference to joint
15      float    weightValue;        //weighting value between 0.0 and 1.0
16      Vector3  position;           //position relative to joint
17  };
18
19  struct MD5Vert {
20      int      vertIndex;          //index of vertex from 0 - n
21      Vector2  texCoords;          //uv map coordinates for vert
22      int      weightElements;     //number of weights that affect vert
23      int      weightIndex;        //index of the first weight
24  };
```
MD5FileData.h

MD5Anim Class

Each MD5 animation has a 'base frame' - a set of starting transforms for the joints in the mesh, similar to the 'bind pose' of the mesh itself. Each frame of the animation consists of a list of values representing the differences between the baseframe's transform and the transforms required for the current frame. This is a specific feature of the MD5 format, rather than of 'skeletal animation' in general, so other formats may not do this.

Creating the skeleton for an animation frame is really simple - it's just the same as the scene node transforms you are used to! The root node is transformed by the frame's transform, and then its children, and so on, in such a way as any node's parent node has already been transformed to that frame's position and orientation. So to create the world transform of any joint is simply a case of multiplying the joint's local transform by the world transform of its parent.

```
1   For each joint {
2       Generate local joint transform from frame transform
3
4       if joint is root {
5           Set joint transform
6       }
7       else{
8           Multiply local joint transform by parent joint transform
9           Set joint transform
10      }
11  }
```
MD5Anim TransformSkeleton Pseudocode

MD5Mesh Class

The *MD5Mesh* class handles the skinning of a mesh's vertices from a skeleton transformed by an animation frame. This is done through the *MD5Mesh* member function *SkinVertices*, which has a reference to some skeleton data as a parameter, created from an animation frame.

```
1  for each vertex {
2      vertex.texCoords = mesh.texCoords[vertex] //Direct Copy!
3      vertex.position  = Vector3(0,0,0)          //Made up of weightings...
4
5      for each vertex.weightElements   {
6        weight = mesh.weights[vertex.weightIndex + current weightElement]
7        joint  = skeleton.joints[weight.jointIndex]
8
9        vertex.position += (joint.transform * weight.position)
10                       *  weight.weightValue;
11     }
12  }
13  RebufferData()
```

MD5Mesh SkinVertices Pseudocode

So how is mesh data 'rebuffered'? Fortunately, this is quite simple. It's the same way as how data is buffered, but instead of calling the OpenGL API function **glBufferData**, we use **glBufferSub-Data**. This function copies a chunk of system memory to graphics memory, but unlike **glBufferData**, does not allocate any new memory, it just reuses the memory allocated for the VBO by a prior call to **glBufferData**, which *always* allocates new memory. Using **glBufferData**, you would effectively get a 'memory leak' within your graphics card memory, constantly allocating a new frame's worth of vertex attributes. As you can probably imagine, that quickly turns into rather a lot of data!

```
1  void MD5Mesh::RebufferData()  {
2      glBindBuffer(GL_ARRAY_BUFFER, bufferObject[VERTEX_BUFFER]);
3      glBufferSubData(GL_ARRAY_BUFFER, 0, numVertices*sizeof(Vector3),
4                  (void*)vertices);
5
6      if(textureCoords) {
7          glBindBuffer(GL_ARRAY_BUFFER, bufferObject[TEXTURE_BUFFER]);
8          glBufferSubData(GL_ARRAY_BUFFER, 0, numVertices*sizeof(Vector2),
9                  (void*)textureCoords);
10     }
11     if (colours)   {
12         glBindBuffer(GL_ARRAY_BUFFER, bufferObject[COLOUR_BUFFER]);
13         glBufferSubData(GL_ARRAY_BUFFER, 0, numVertices*sizeof(Vector4),
14                  (void*)colours);
15     }
16     if(indices) {
17       glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,bufferObject[INDEX_BUFFER]);
18         glBufferSubData(GL_ELEMENT_ARRAY_BUFFER, 0,
19               numVertices*sizeof(unsigned int), (void*)indices);
20     }
21  //A sneak preview of the lighting tutorials (definately not a mistake)
22     if(normals) {
23         glBindBuffer(GL_ARRAY_BUFFER, bufferObject[NORMAL_BUFFER]);
24         glBufferSubData(GL_ARRAY_BUFFER, 0, numVertices*sizeof(Vector3),
25                  (void*)normals);
26     }
27     if(tangents) {
28         glBindBuffer(GL_ARRAY_BUFFER, bufferObject[TANGENT_BUFFER]);
29         glBufferSubData(GL_ARRAY_BUFFER, 0, numVertices*sizeof(Vector3),
30                  (void*)tangents);
31     }
32  }
```

MD5Mesh.cpp

Renderer Class

To demonstrate how to use the MD5 skeletal animation classes, we're going to create a quick scene - just a camera pointing at an instance of the provided HellKnight mesh, as it runs through an animation. So in the *Renderer* class header, we need a pointer to an MD5*FileData* class, an *MD5Node* to control the HellKnight, and a *Camera* instance, and overloads for *RenderScene* and *UpdateScene*.

```cpp
1  #pragma once
2
3  #include "./NCLGL/OGLRenderer.h"
4  #include "./NCLGL/Camera.h"
5  #include "./NCLGL/MD5Mesh.h"
6
7  class Renderer : public OGLRenderer    {
8  public:
9      Renderer(Window &parent);
10     virtual ~Renderer(void);
11     virtual void RenderScene();
12     virtual void UpdateScene(float msec);
13 protected:
14     MD5FileData*    hellData;
15     MD5Node*        hellNode;
16     Camera*         camera;
17 };
```
Renderer.h

We don't need a new shader, the simple *TexturedVertex* and *TexturedFragment* programs from Tutorial 3 will do just fine. On line 7, we load in some data from an MD5 file into an *MD5FileData* class, then on line 8, we create a new *MD5Node* to control the file data. Each MD5 can have a number of animations, each loaded in from an MD5Anim file - the *AddAnim* function on line 14 will add the animations from such a file to a **map** within the *MD5FileData* instance. This animation can then be played by using the *PlayAnim* function of an MD5Node, passing in the name of the animation to play.

```cpp
1  #include "Renderer.h"
2  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
3      camera          = new Camera(0,90.0f,Vector3(-180,60,0));
4      currentShader   = new Shader(SHADERDIR"TexturedVertex.glsl",
5                                   SHADERDIR"TexturedFragment.glsl");
6
7      hellData    = new MD5FileData(MESHDIR"hellknight.md5mesh");
8      hellNode    = new MD5Node(*hellData);
9
10     if(!currentShader->LinkProgram()) {
11         return;
12     }
13
14     hellData->AddAnim(MESHDIR"idle2.md5anim");
15     hellNode->PlayAnim(MESHDIR"idle2.md5anim");
16
17     projMatrix = Matrix4::Perspective(1.0f,10000.0f,
18                  (float)width / (float)height, 45.0f);
19
20     glEnable(GL_DEPTH_TEST);
21     init = true;
22 }
```
Renderer.cpp

As ever, everything we create on the heap in the *Renderer* **constructor** must be **deleted** in the *Renderer* **destructor**.

```cpp
23 Renderer::~Renderer(void)  {
24     delete camera;
25     delete hellData;
26     delete hellNode;
27 }
```
<div align="center">Renderer.cpp</div>

In *UpdateScene*, along with the camera updating and view matrix construction, we must update our *MD5Mesh* instance, by calling *UpdateAnim*, passing the *msec* value to it as a parameter. This works out the next frame to generate a skeleton for, and skins its vertices with the resulting transformed skeleton.

```cpp
28  void Renderer::UpdateScene(float msec)   {
29     camera->UpdateCamera(msec);
30     viewMatrix     = camera->BuildViewMatrix();
31     hellNode->Update(msec);
32 }
```
<div align="center">Renderer.cpp</div>

Other than the call to *UpdateAnim*, we don't actually need any new code, so *RenderScene* should look very familiar to you. You can use the function *DrawSkeleton* if you want to see the skeleton joints and bones instead of the actual mesh.

```cpp
33 void Renderer::RenderScene()  {
34     glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
35
36     glUseProgram(currentShader->GetProgram());
37
38     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
39         "diffuseTex"), 0);
40
41     UpdateShaderMatrices();
42
43     hellNode->Draw(*this);
44
45     glUseProgram(0);
46     SwapBuffers();
47 }
```
<div align="center">Renderer.cpp</div>

Tutorial Summary

On running the program, you should see the HellKnight looping through an 'idle' animation, looking from side to side. No matter what framerate your application gets, the *UpdateAnim* function ensures that the animations proceed at the correct speed. Skeletal animation is the *de facto* standard way for complex characters to be animated in games, often with multiple blended animations. Skeletal animation with vertex skinning allows characters to move realistically, and with fewer graphical anomalies than a simple scene node transformations. Beyond the actual animating of characters, joints are useful for attaching in-game items to a mesh - joints will often have name strings, allowing them to be easily referenced by in-game scripting tools. There's not actually that much to skeletal animation and skinning, it's just a simple variation on the scene graph hierarchy you are now used to - correctly loading in the file formats is trickier than actually animating them (take a look at the file loading code, and then be very happy its been done for you!).

Further Work

1) Investigate the sourcecode for the *MD5Mesh*, *MD5FileData* and *MD5Anim* classes. Beyond the actual skinning and skeletal animation functions, they should provide an insight into how meshes and animations are stored in a triple-A title, and how to load them in using the STL.

2) Earlier it was mentioned that skeletal animation skinning could be performed on the graphics card. Investigate how to send a transformed skeleton's transformation matrices to a vertex shader. What about vertex attributes, how would you use those to send weights to the graphics card? Assume no vertex is influenced by more than 4 weights.

3) Investigate **Texture Buffer Objects**, and **Uniform Buffer Objects** - two methods of storing arbitrary data in graphics memory. Could they be used to perform vertex skinning? What might the advantages and disadvantages of these methods be?

4) Despite the various ways of storing data on the graphics card, some developers still keep the skeletal transform data as a simple texture in graphics memory. Can you think of any advantage to doing this? Hint: This method generally doesn't use nearest-neighbour sampling.