Vertex Skinning

Vertex skinning (other names for this would be *enveloping* or, more scientifically, *Skeleton-Subspace Deformation* (SSD, see Lewis et al. 2000)) is a process of animating vertices of a 3d-mesh by controlling them via a skeleton. If the bones of a skeleton move, the vertices move with them.

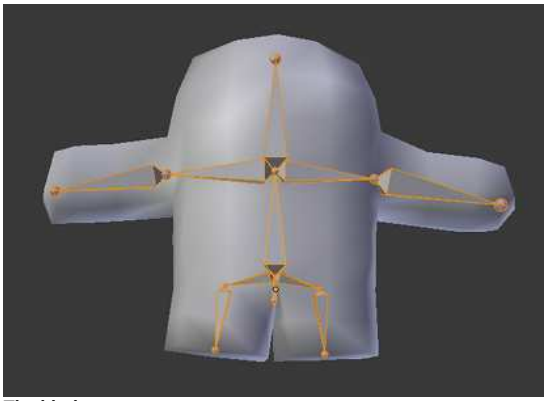
The simplest form of this would be to attach each vertex to one bone of the skeleton and transform it accordingly. But this would look unnatural for smooth vertex meshes that represent, for example, the skin of a game character (this is probably why it is called skinning). The solution to this is to use multiple bones to influence one vertex. For each bone, you define a *weight* value that tells you how much the bone influences the vertex.

In games, the number of influencing bones per vertex is usually restricted to 3 or 4 to save memory and speed up the calculations. The weight values should add up to 1 per vertex, otherwise the vertices do not follow the skeleton correctly. This allows you to omit one of the weight values to reduce the memory consumption (but this increases the number of computations a bit).

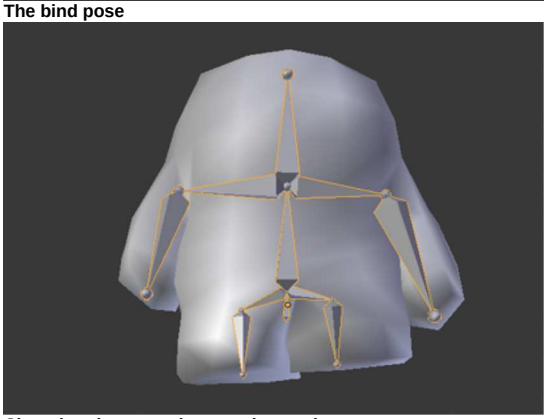
The vertices are defined in model space. To transform the vertices correctly, they must be transformed from this model space to the local coordinate space of the bone. Now the bones can be moved and rotated. The vertex should still be in the same relative position to the bone after this transformation, so transforming the vertex back to the model view using the newly transformed bone will set it to the correct position.

The matrix needed for the first part (the transformation to the local space of the joint) is called the *inverse bind matrix*. The *bind pose* of a bone is the position where the bone originally is. When creating an animation for a 3D-model, you usually use a simple pose (where the game character usually has his arms outstretched), to which you will add the skeleton. This is the bind pose. The *bind matrix* is a transformation from the root of the skeleton to the location of a joint the bind pose. That is why we need the *inverse* bind matrix that translates from this bind pose back to a coordinate space in which the joint is positioned in the zero-coordinate.

Then there is the second matrix that translates the vertex back to where the newly translated joint is. This matrix is created by going from the joint through all the parent joints and multiplying each translation matrix.



The bind pose



Changing the pose changes the mesh

The inverse bind pose matrix stays the same once the connection to the mesh is established, but the matrix for the current pose changes all the time and has to be recalculated each time one of the joints has been moved. Both of these matrices can be concatenated into one matrix. What you finally need for the skinning algorithm is the concatenated matrix that transforms the vertices from the bind position to the new position, an inverse-transpose of this matrix (without the translations, so a 3x3 matrix) needed to transform the surface normals, and a list of joint indices and corresponding weights for each vertex.

Here is an example of how to apply a skeleton pose directly to a mesh. You probably wouldn't want to copy the mesh data each time during an animation, one way of doing this is to save the changed vertex data in another, pre-created vertex array that is then used for rendering.

```
Mesh Mesh::applyPoseToMesh(SkeletonPose *pose)
{
    Mesh meshRet;

    // copy mesh values
    meshRet = *this;

    if(skinningData != 0)
    {
        // precalculate transformation matrices
        Matrix4 *boneMatrix = new Matrix4[skeleton.nrOfJoints];
        Matrix3 *boneMatrixIT = new Matrix3[skeleton.nrOfJoints];
        for(uint32 h = 0; h < skeleton.nrOfJoints; h++)
        {
            Joint &joint = skeleton.joints[h];

            boneMatrix[h] = pose->globalPose[h] * joint.invBindPose;
            boneMatrixIT[h] = Matrix3(boneMatrix[h]).inverse().transpose();
        }

        for(uint32 j = 0; j < vertexCount; j++)
        {
            Vertex &vert = vertices[j];

            SkinningData &data = skinningData[j];

            Vector4 vect(0.0f, 0.0f, 0.0f, 0.0f);

            meshRet.vertices[j].normal = Vector3(0.0f, 0.0f, 0.0f);
            for(uint32 h = 0; h < 4; h++)
            {
                uint8 jointIndex = data.jointIndex[h];

                vect += boneMatrix[jointIndex] * Vector4(vert.pos, 1.0f) * data.jointWeight[h];

                meshRet.vertices[j].normal += boneMatrixIT[jointIndex] * vert.normal * data.jointIndex[h];
            }

            meshRet.vertices[j].pos = Vector3(vect.x / vect.w, vect.y / vect.w, vect.z / vect.w);
            meshRet.vertices[j].normal.normalize();
        }

        delete boneMatrix;
        delete boneMatrixIT;
    }

    return meshRet;
}
```

Changing the mesh during an animation can be done either on the CPU or on the GPU. Both have its advantages and disadvantages. Animating on the GPU is usually faster (but not always) and doesn't need an extra copy of the mesh (there is something like a copy, but it is on the GPU). But for each joint, you need to send one 4x4 matrix (the skinning matrix) and one 3x3 matrix (the inverse transpose skinning matrix) as uniform variables to the vertex shader. Some hardware, especially when targeting mobile devices, do not support that many uniform variables, or use external memory for them, which is slow. One workaround for this is to use "batching": The mesh and the skeleton is divided into parts, that are drawn one after the other. As each part now needs less bones, it needs less memory. The difficulty here is to divide the skeleton and mesh into nice parts.

If you prefer to change the mesh on the CPU, use SIMD instructions (or something similar) to speed up the calculations. If the GPU or CPU version is faster seems to depend on how much you are already doing on the GPU and the CPU.

Here is an example (GLSL ES 2.0) of how to do skinning inside the vertex shader. Only 16 joints (see BoneMatrixArray) are supported here.

```
attribute vec3 inPosition;
attribute vec3 inNormal;
attribute vec2 inTexCoord;
attribute vec4 inTangent;

attribute mediump vec4 inBoneIndex;
attribute mediump vec4 inBoneWeights;

uniform mat4 modelViewProjection;

varying mediump vec2 v_texCoord;

uniform mediump int BoneCount;
uniform highp mat4 BoneMatrixArray[16];
uniform highp mat3 BoneMatrixArrayIT[16];

void main()
{
    //--------- skinning -----------

    // used to rotate the values (instead of getting them with the [] operator)
    mediump ivec4 boneIndex = ivec4(inBoneIndex);
    mediump vec4 boneWeights = inBoneWeights;

    highp vec4 position = vec4(0.0, 0.0, 0.0, 0.0);
    vec3 objectNormal = vec3(0.0, 0.0, 0.0);
    vec3 objectTangent = vec3(0.0, 0.0, 0.0);

    bool bSkinningApplied = false;
    for(lowp int i = 0; i < 4; i++)
    {
        if(boneIndex.x < BoneCount)
        {
            highp mat4 boneMatrix = BoneMatrixArray[boneIndex.x];
            mediump mat3 normalMatrix = BoneMatrixArrayIT[boneIndex.x];

            position += boneMatrix * vec4(inPosition, 1.0) * boneWeights.x;
            objectNormal += normalMatrix * inNormal * boneWeights.x;
            objectTangent += normalMatrix * inTangent.xyz * boneWeights.x;
        }

        // "rotate" the vectors
        boneIndex = boneIndex.yzwx;
        boneWeights = boneWeights.yzwx;
    }


    gl_Position = modelViewProjection * position;
    v_texCoord = inTexCoord;

    ... continue by doing the lighting calculations ...
}
```

References:

Lewis, J.P.; Cordner, Matt & Fong, Nickson (2000): Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation (http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0CDYQFjAB&url=http%3A%2F%2Fwww.researchgate.net%2Fpublication%2F2357143_Pose_Space_Deformation_A_Unified_Approach_to_Shape_Interpolation_and_Skeleton-Driven_Deformation%2Ffile%2F79e4150a60b3ea96a6.pdf&ei=zlBqU9_YL6K27QaP_YD4Dg&usg=AFQjCNF53ntQ92n92FGgvhSt--Wi-F2YxA&sig2=mh2DmUtwPAJdknSY1ocMpA&bvm=bv.66111022,d.ZGU)

back (index.php)