Fast Skinning

March 21st 2005
J.M.P. van Waveren

Abstract

Two approaches to matrix palette skinning are presented and optimized. Furthermore the Intel Streaming SIMD Extensions are used to exploit parallelism and get the most out of every clock cycle. The optimized routines are well over two times faster than the implementation in C on a Pentium 4.

1. Introduction

Presenting realistic organic models has become increasingly important in many applications among which computer games. Such an organic model is typically represented by a polygonal mesh often referred to as a 'skin' which continuously changes shape with an underlying structure often referred to as a 'skeleton'. The process of animating the mesh is referred to as 'skinning'.

On today's hardware skinning can be performed both on a general purpose CPU and a more specialized GPU as available on many graphics cards. The speed of these GPUs increases rapidly and maximum parallelism is exploited which makes skinning on the GPU favorable in many cases. However, for some applications it may be beneficial to perform skinning on the CPU.

Performing skinning on the CPU improves compatibility across a wide range of systems. Older systems may have graphics cards (like the GeForce2 and GeForce4MX) that do not support the necessary features to perform skinning on the GPU.

Many applications also need the post-transformed data for shadow volumes and collision detection. Current graphics cards do not allow this data to be retrieved after it has been processed. Skinning on the CPU however, allows easy access to the animated mesh while still allowing the GPU to do the lighting and clipping calculations. The calculation of shadow volumes may be offloaded completely to the GPU. However, this is not particularly efficient on today's hardware [10].

When skinning on the GPU the number of joints from the skeleton that influence each vertex must be static for each batch of triangles [9]. This can lead to many wasted transformations on the GPU because the maximum number of joint influences will be used for all vertices in a single batch.

Some rendering techniques require a mesh to be drawn multiple times often to temporary buffers or surfaces. For instance shadow volume and shadow map rendering requires a mesh to be processed for every light source interacting with the mesh. When a mesh needs to be rendered multiple times any vertex transformations have to be recalculated on the GPU for every rendering pass because intermediate results cannot be saved. This may cause the performance to be limited by vertex program execution on the GPU. When this happens performance may improve when vertices are transformed on the CPU where any transformations only need to be performed once.

On current hardware the number of vertices that is processed in a single batch has to be maximized for optimal performance [11,12]. The batch sizes should be at least several thousands of triangles. When skinning on the GPU this may be hard to achieve because the number of joints in the skeleton used for one triangle mesh is restricted by vertex program constant space. The program constants are typically used to store the transformation matrices for the joints of a skeleton.

The above screenshots show a model from the computer game DOOM III with a skeleton with 128 joints. The original hardware vertex program specification has a fixed limit of 96 vertex program constants where each constant is a vector with four floating point components. Even when 3x4 matrices or quaternions are used this imposes a significant limit on the maximum number of joints that can be used for a single mesh. This means meshes that use more joints than can be stored in the vertex program constants like the one in the model from DOOM III above have to be split into smaller meshes which reduces efficiency and increases the number of duplicate vertices [9].

1.1 Previous Work

Skeletal animation systems are not new [2, 3] and are used in many applications. On today's hardware skeletal animation of meshes can be performed on both the CPU and the GPU. As the number of triangles used to present realistic models increases skinning can become a time consuming process on both the CPU and GPU. Optimizing the skinning process is essential for many applications and hardware features like specialized instruction sets are often exploited for this purpose [8].

1.2 Layout

Section 2 shows some properties of and the data used for matrix palette skinning. Section 3 describes one approach to matrix palette skinning. Another approach is presented in section 4. In section 5 the SSE3 instruction set is used to further improve the performance. The results of the optimizations are presented in section 6 and several conclusions are drawn in section 7.

2. Matrix Palette Skinning

Matrix palette skinning animates a mesh where the vertices are transformed using a palette of the matrices that describe the joint transformations of a skeleton. To calculate the position and other properties of each vertex a selection of the joint matrices are weighted and used to transform a set of base vectors.

A vertex of the mesh is described in code as follows.

```
struct Vec4 {
    float   x, y, z, w;
};

struct Vertex {
    Vec4    position;
    Vec4    normal;
    Vec4    tangent;
};
```

The vertex uses 4D vectors for the position, normal and tangent while 3D vectors may be sufficient. However, using 4D vectors improves memory alignment and the last component of the 4D vectors could also be used for other purposes. The 'tangent.w' could for instance be used to store a texture polarity sign. The second tangent can then be derived using a cross product between the normal and the tangent where the vector is flipped based on the texture polarity.

```
bitangent.x = tangent.w * ( normal.y * tangent.z - normal.z * tangent.y );
bitangent.y = tangent.w * ( normal.z * tangent.x - normal.x * tangent.z );
bitangent.z = tangent.w * ( normal.x * tangent.y - normal.y * tangent.x );
```

For some of the vertex properties 3D vectors could be used and interleaved with new properties, like 4 byte colors, to maintain alignment. However, simple 4D vectors are used here to get good alignment with minimal complexity.

A joint transformation is described with a 3x4 matrix as follows.

```
 Struct JointMat {
    float   mat[3*4];
};
```

Such a 3x4 matrix consists of a 3x3 orthonormal rotation matrix and a 3D translation vector. The first three elements of each row are from the 3x3 rotation matrix and the last element of each row is a translation along one of the coordinate axes.

A single influence or joint weight from the palette is described in code as follows.

```
struct JointWeight {
    float   weight;
    int     jointMatOffset;
    int     nextVertexOffset;
};
```

The 'weight' is the scale factor for the influence. Usually the weights for all influences for a single vertex add up to one.

The 'jointMatOffset' is the offset in bytes to the joint matrix associated with the influence. This byte offset can be added directly to the base pointer of an array with all the joint matrices to get the correct matrix for an influence.

The 'nextVertexOffset' is the offset in bytes to the first weight for the next vertex and is used to tell when the last joint influence for a vertex has been processed. The last joint influence is found when this offset equals the size of a 'JointWeight' object. The 'nextVertexOffset' also allows easy capping of the number of influences to one influence per vertex which could be used for a very simple approach to LOD. This does require that the joint influences are sorted with decreasing weight. Only the first joint influence would be used with an assumed weight of one, and the 'nextVertexOffset' would be used to jump right to the first influence for the next vertex. Some snapping may occur when the number of joint influences changes for a vertex. However, for distant models this may not be or hardly noticeable while decreasing the number of processed influences may improve performance considerably.

3. Skinning Without Normals Or Tangents

The following approach to matrix palette skinning is best used when only vertex positions are needed. For instance for collision detection the vertex normals and tangent vectors are usually not required. The vertex positions may also be used to create shadow volumes or render shadow maps for a skinned mesh that is not visible itself but still casts shadows on visible geometry. This approach can also be used to skin all meshes for shadow volumes while the visible mesh is skinned on the GPU with normals and tangents. For some applications this may be the perfect trade between using the CPU and the GPU.

For this approach a set of base vectors are stored relative to the joints that transform them. If a vertex is influenced by more than one joint, multiple base vectors are stored, one for each joint with its weighting. Each base vector is multiplied by its associated weight and transformed with the appropriate joint matrix. The following pseudo code shows how one vertex is transformed in this manner.

```
position = matrix0 * base0 * weight0;
position += matrix1 * base1 * weight1;
position += matrix2 * base2 * weight2;
...
```

The 'base?' and 'weight?' variables are the base vectors and weights for each influence. The 'matrix?' variables are the joint matrices associated with the influences.

Because both a base vector and a weight are stored for every influence the base vectors can be premultiplied with the weights which eliminates several multiplications from the real-time skinning process. The following pseudo code shows the transformation of one vertex with the weighted base vectors.

```
position = matrix0 * weightedBase0;
position += matrix1 * weightedBase1;
position += matrix2 * weightedBase2;
...
```

The following C/C++ code implements the complete skinning routine.

```
void MulMatVec( Vec4 &result, const JointMat &m, const Vec4 &v ) const {
    result.x = m.mat[0 * 4 + 0] * v.x + m.mat[0 * 4 + 1] * v.y + m.mat[0 * 4 + 2] * v.z + m.mat[0 * 4 + 3] * v.w;
    result.y = m.mat[1 * 4 + 0] * v.x + m.mat[1 * 4 + 1] * v.y + m.mat[1 * 4 + 2] * v.z + m.mat[1 * 4 + 3] * v.w;
    result.z = m.mat[2 * 4 + 0] * v.x + m.mat[2 * 4 + 1] * v.y + m.mat[2 * 4 + 2] * v.z + m.mat[2 * 4 + 3] * v.w;
}

void MadMatVec( Vec4 &result, const JointMat &m, const Vec4 &v ) const {
    result.x += m.mat[0 * 4 + 0] * v.x + m.mat[0 * 4 + 1] * v.y + m.mat[0 * 4 + 2] * v.z + m.mat[0 * 4 + 3] * v.w;
    result.y += m.mat[1 * 4 + 0] * v.x + m.mat[1 * 4 + 1] * v.y + m.mat[1 * 4 + 2] * v.z + m.mat[1 * 4 + 3] * v.w;
    result.z += m.mat[2 * 4 + 0] * v.x + m.mat[2 * 4 + 1] * v.y + m.mat[2 * 4 + 2] * v.z + m.mat[2 * 4 + 3] * v.w;
}

void TransformVerts( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4 *base, const JointWeight
*weights, int numWeights ) {
    int i, j;
    const byte *jointsPtr = (byte *)joints;

    for( j = 0, i = 0; i < numVerts; i++, j++ ) {
        idVec4 v;

        MulMatVec( v, ( *(JointMat *) ( jointsPtr + weights[j].jointMatOffset ) ), base[j] );
        while( weights[j].nextVertexOffset != sizeof( JointWeight ) ) {
            j++;
            MadMatVec( v, ( *(JointMat *) ( jointsPtr + weights[j].jointMatOffset ) ), base[j] );
        }

        verts[i].position.x = v.x;
        verts[i].position.y = v.y;
        verts[i].position.z = v.z;
    }
}
```

Because the number of joint influences may be different for each vertex it is not trivial to exploit parallelism through increased throughput. The vertices could be grouped based on the number of influences but this can decrease the performance on today's graphics hardware where locality of vertices is essential for efficient caching.

Fortunately the above routine is well suited for exploiting parallelism with a compressed calculation. The matrix vector multiplications involve many independent operations that can be executed in parallel. The Intel Streaming SIMD Extensions can be used to exploit this parallelism.

Each matrix vector multiplication involves 9 horizontal additions that would typically require a swizzle before they can be executed in parallel with SSE instructions. However, these horizontal additions can be postponed until after all the joint influences are added together. Instead of accumulating the transformed base vectors the partial matrix vector products are accumulated and the horizontal additions are performed after looping over the joint influences.

The complete SSE optimized routine is listed in appendix A. The swizzle for the horizontal additions minimizes the number of instructions and dependencies. The SSE2 instruction 'pshufd' is used to separate the last two scalars that need to be added together. The 'pshufd' instruction is meant to be used for double word integer data. However, since every 32 bits floating point bit pattern represents a valid integer this instruction can be used on floating point data without problems.

The routine listed in appendix A assumes the list with joint matrices and the list with base vectors are 16 byte aligned. The routine works with any alignment for the list with weights and the list with vertices. However, for optimal performance the list with vertices should be at least 16 byte aligned and the size of vertex objects should be at least a multiple of 16 bytes such that consecutive vertices in an array are all aligned on a 16 byte boundary.

4. Skinning With Normals And Tangents

When not only the vertex position is needed but also the vertex normal and tangent vectors, it is more efficient to first accumulate weighted joint matrices and use this new matrix to transform the vertex position, normal and tangents in model space. For this approach a base pose of a mesh is transformed. The vertices of this base pose are stored only once in model space even when influenced by multiple joints. These vertices are not transformed directly by the joint matrices of the animated skeleton but the joint matrices are first multiplied with the inverse joint matrices for the base pose. These inverse joint matrices can be precalculated because the same base pose is used during all animation. The joint matrices of the animated skeleton can then be multiplied with these inverse joint matrices of the base pose before skinning the mesh. The following pseudo code shows how a single vertex is transformed.

```
accumulated = matrix0 * weight0;
accumulated += matrix1 * weight1;
accumulated += matrix2 * weight2;
...
position = accumulated * basePosition;
normal = accumulated * baseNormal;
tangent = accumulated * baseTangent;
```

The weights are the same as used for the approach discussed in the previous section. However, the joint matrices have been premultiplied with the inverse joint matrices of the base pose. The following C/C++ code implements the complete skinning routine.

```
void MulMatScalar( JointMat &result, const JointMat &mat, const float s ) {
    result.mat[0 * 4 + 0] = s * mat.mat[0 * 4 + 0];
    result.mat[0 * 4 + 1] = s * mat.mat[0 * 4 + 1];
    result.mat[0 * 4 + 2] = s * mat.mat[0 * 4 + 2];
    result.mat[0 * 4 + 3] = s * mat.mat[0 * 4 + 3];
    result.mat[1 * 4 + 0] = s * mat.mat[1 * 4 + 0];
    result.mat[1 * 4 + 1] = s * mat.mat[1 * 4 + 1];
    result.mat[1 * 4 + 2] = s * mat.mat[1 * 4 + 2];
    result.mat[1 * 4 + 3] = s * mat.mat[1 * 4 + 3];
    result.mat[2 * 4 + 0] = s * mat.mat[2 * 4 + 0];
    result.mat[2 * 4 + 1] = s * mat.mat[2 * 4 + 1];
    result.mat[2 * 4 + 2] = s * mat.mat[2 * 4 + 2];
    result.mat[2 * 4 + 3] = s * mat.mat[2 * 4 + 3];
}

void MadMatScalar( JointMat &result, const JointMat &mat, const float s ) {
    result.mat[0 * 4 + 0] += s * mat.mat[0 * 4 + 0];
    result.mat[0 * 4 + 1] += s * mat.mat[0 * 4 + 1];
    result.mat[0 * 4 + 2] += s * mat.mat[0 * 4 + 2];
    result.mat[0 * 4 + 3] += s * mat.mat[0 * 4 + 3];
    result.mat[1 * 4 + 0] += s * mat.mat[1 * 4 + 0];
    result.mat[1 * 4 + 1] += s * mat.mat[1 * 4 + 1];
    result.mat[1 * 4 + 2] += s * mat.mat[1 * 4 + 2];
    result.mat[1 * 4 + 3] += s * mat.mat[1 * 4 + 3];
    result.mat[2 * 4 + 0] += s * mat.mat[2 * 4 + 0];
    result.mat[2 * 4 + 1] += s * mat.mat[2 * 4 + 1];
    result.mat[2 * 4 + 2] += s * mat.mat[2 * 4 + 2];
    result.mat[2 * 4 + 3] += s * mat.mat[2 * 4 + 3];
}

void MulMatVec( Vec4 &result, cont JointMat &m, const Vec4 &v ) const {
    result.x = m.mat[0 * 4 + 0] * v.x + m.mat[0 * 4 + 1] * v.y + m.mat[0 * 4 + 2] * v.z + m.mat[0 * 4 + 3] *
v.w;
    result.y = m.mat[1 * 4 + 0] * v.x + m.mat[1 * 4 + 1] * v.y + m.mat[1 * 4 + 2] * v.z + m.mat[1 * 4 + 3] *
v.w;
    result.z = m.mat[2 * 4 + 0] * v.x + m.mat[2 * 4 + 1] * v.y + m.mat[2 * 4 + 2] * v.z + m.mat[2 * 4 + 3] *
v.w;
}

void TransformVertsAndTangents( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4 *base,
const JointWeight *weights, const int numWeights ) {
    int i, j;
    const byte *jointsPtr = (byte *)joints;

    for( j = i = 0; i < numVerts; i++, j++ ) {
        JointMat mat;

        MulMatScalar( mat, *(JointMat *) ( jointsPtr + weights[j].jointMatOffset ), weights[j].weight );
        while( weights[j].nextVertexOffset != JOINTWEIGHT_SIZE ) {
            j++;
            MadMatScalar( mat, *(JointMat *) ( jointsPtr + weights[j].jointMatOffset ), weights[j].weight );
        }

        MulMatVec( verts[i].position, mat, base[i*3+0] );
        MulMatVec( verts[i].normal, mat, base[i*3+1] );
        MulMatVec( verts[i].tangent, mat, base[i*3+2] );
    }
}
```

This routine only transforms the position, normal and first tangent vector. Where necessary the second tangent vector can be trivially derived on the GPU with a cross product between the normal and first tangent vector. If it is not possible to calculate the second tangent vector on the GPU the above routine can be trivially extended to transform an additional vector.

The normal and tangent vector may become denormalized when influenced by multiple joint matrices because the vectors are interpolated linearly with the scaled matrices. However, most vertices are influenced by very few joint matrices and the denormalization is minimal. For most applications the denormalization is also not a serious problem. Where necessary the vectors can be trivially re-normalized on the GPU with very few instructions.

The above routine is also optimized using the Intel Streaming SIMD Extensions to exploit parallelism with a compressed calculation. The complete SSE optimized routine is listed in appendix B. The routine assumes the same alignment of the input arrays as the routine presented in the previous section.

5. SSE3

For the matrix vector multiplications both skinning routines use horizontal additions with several swizzle instructions. The following code is for instance used to perform the first eight of the in total nine horizontal additions of the floating point values stored in the registers 'xmm0', 'xmm1' and 'xmm2.

```
// xmm0 = m0, m1, m2, t0
// xmm1 = m3, m4, m5, t1
// xmm2 = m6, m7, m8, t2

movaps      xmm6, xmm0    // xmm6 =    m0,    m1,          m2,          t0
unpcklps    xmm6, xmm1    // xmm6 =    m0,    m3,          m1,          m4
unpckhps    xmm0, xmm1    // xmm1 =    m2,    m5,          t0,          t1
addps       xmm6, xmm0    // xmm6 = m0+m2, m3+m5,       m1+t0,       m4+t1

movaps      xmm7, xmm2    // xmm7 =    m6,    m7,          m8,          t2
movlhps     xmm2, xmm6    // xmm2 =    m6,    m7,       m0+m2,       m3+m5
movhlps     xmm6, xmm7    // xmm6 =    m8,    t2,       m1+t0,       m4+t1
addps       xmm2, xmm6    // xmm2 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1
```

The SSE3 instruction set available on the Intel® Pentium® 4 Processor on 90nm Technology has an instruction to horizontally add the floating point values of two registers. The code below does exactly the same as the above code. However, the code below uses only two 'haddps' instructions.

```
haddps      xmm0, xmm1    // xmm0 = m0+m1, m2+t0,      m3+m4,        m5+t1
haddps      xmm2, xmm0    // xmm2 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1
```

The 'haddps' instruction has high latency on the Pentium 4 but nevertheless changing the above horizontal additions to use this instruction makes the skinning routine faster. The last horizontal addition as shown in the following code is not replaced by the 'haddps' instruction.

```
pshufd      xmm3, xmm2, R_SHUFFLE_D( 1, 0, 2, 3 )
addss       xmm3, xmm2
```

Using the 'haddps' instruction instead of the above code would introduce a dependency and together with the high latency of the 'haddps' instruction the skinning routine would actually become slower.

The routines using the SSE3 instruction 'haddps' are listed in appendix C and D.

6. Results

The various routines have been tested on an Intel® Pentium® 4 Processor on 130nm Technology and an Intel® Pentium® 4 Processor on 90nm Technology. The routines operated on a list of 1024 vertices with 2 joint weights per vertex. The total number of clock cycles and the number of clock cycles per vertex for each routine on the different CPUs are listed in the following table.

Hot Cache Clock Cycle Counts				
Routine	**P4 130nm total clock cycles**	**P4 130nm clock cycles per element**	**P4 90nm total clock cycles**	**P4 90nm clock cycles per element**
TransformVerts (C)	107008	105	130005	127
TransformVerts (SSE)	33784	33	43956	43
TransformVerts (SSE3)	-	-	41963	41
TransformVertsAndTangents (C)	196056	192	230103	225
TransformVertsAndTangents (SSE)	76324	75	89775	88
TransformVertsAndTangents (SSE3)	-	-	81855	80

7. Conclusion

Whether or not skinning is best performed on the GPU or the CPU should be decided on a per application or even on a per object basis. For objects that are best skinned on the CPU the routines presented here can be used for optimal performance.

The approach to matrix palette skinning presented in section 3 delivers the best performance when only the vertex positions are required. However, when vertex normals and tangents are required as well the second approach to matrix palette skinning presented in section 4 should be used for optimal performance.

The Intel SIMD Extension have been used to optimize both approaches to matrix palette skinning on the CPU. The SSE optimized routines are between two and three times faster than their C counterparts.

8. References

1. Slashing Through Real-Time Character Animation
   Jeff Lander
   Game Developer Magazine, April 1998
   Available Online: http://www.darwin3d.com/gdm1998.htm#gdm0498

2. Skin Them Bones: Game Programming for the Web Generation
   Jeff Lander
   Game Developer Magazine, May 1998
   Available Online: http://www.darwin3d.com/gdm1998.htm#gdm0598

3. Over My Dead, Polygonal Body
   Jeff Lander
   Game Developer Magazine, October 1999
   Available Online: http://www.darwin3d.com/gdm1999.htm#gdm1099

4. Run-Time Skin Deformation
   Jason Weber
   Game Developers Conference proceedings, 2000
   Available Online:
   http://www.intel.com/technology/systems/3d/skeletal.htm

5. Real-Time Character Animation for Computer Games
   Eike F. Anderson
   National Centre for Computer Animation, Bournemouth University, 2001

6. Character Animation for Real-time Applications
   Michael Putz, Klaus Hufnagl
   Central European Seminar on Computer Graphics, 2002

Available Online: http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2002/

7. 3D Games, Vol. 2: Animation and Advanced Real-Time Rendering
Alan Watt and Fabio Policarpo
Addison Wesley, January 17, 2003
ISBN: 0201787067

8. Optimized CPU-based Skinning for 3D Games
Leigh Davies
Intel, August 2004
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/games/172123.htm

9. Mesh Skinning
Sébastian Dominé
nVidia, 2001
Available Online: http://developer.nvidia.com/object/skinning.html

10. Optimized Stencil Shadow Volumes
Cass Everitt, Mark J. Kilgard
Game Developer Conference, 2003
Available Online:
http://developer.nvidia.com/docs/IO/8230/GDC2003_ShadowVolumes.pdf

11. Optimizing the Graphics Pipeline
Cem Cebenoyan, Matthias Wloka
Game Developer Conference, 2003
Available Online:
http://developer.nvidia.com/docs/IO/8230/GDC2003_PipelinePerformance.pdf

12. GPU Gems - 28. Graphics Pipeline Performance
Cem Cebenoyan
Randima Fernando (editor)
Addison-Wesley, 2004

Appendix A

```
/*
    SSE Optimized Skinning Without Normals or Tangents
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

struct Vec4 {
    float   x, y, z, w;
};

struct JointMat {
    float   mat[3*4];
};

struct JointWeight {
    float   weight;                 // joint weight
    int     jointMatOffset;         // offset in bytes to the joint matrix
    int     nextVertexOffset;       // offset in bytes to the first weight for the next vertex
};

struct Vertex {
    Vec4    position;
    Vec4    normal;
    Vec4    tangent;
};

// offsets for SIMD code
#define BASEVECTOR_SIZE                         (4*4)       // sizeof( idVec4 )
#define JOINTWEIGHT_SIZE                        (3*4)       // sizeof( JointWeight )
#define JOINTWEIGHT_WEIGHT_OFFSET               (0*4)       // offsetof( JointWeight, weight )
#define JOINTWEIGHT_JOINTMATOFFSET_OFFSET       (1*4)       // offsetof( JointWeight, jointMatOffset )
#define JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET     (2*4)       // offsetof( JointWeight, nextVertexOffset )
#define VERTEX_SIZE                             (12*4)      // sizeof( Vertex )
#define VERTEX_POSITION_OFFSET                  (0*4)       // offsetof( Vertex, position )
#define VERTEX_NORMAL_OFFSET                    (4*4)       // offsetof( Vertex, normal )
#define VERTEX_TANGENT_OFFSET                   (8*4)       // offsetof( Vertex, tangent )

void TransformVerts( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4 *base, const
JointWeight *weights, const int numWeights ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( base );

    __asm
    {
        mov         eax, numVerts
        test        eax, eax
        jz          done
        imul        eax, VERTEX_SIZE

        mov         ecx, verts
        mov         edx, weights
        mov         esi, base
        mov         edi, joints

        add         ecx, eax
        neg         eax

    loopVert:
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        movaps      xmm2, [esi]
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm0, xmm2
        add         esi, BASEVECTOR_SIZE
        movaps      xmm1, xmm2

        mulps       xmm0, [edi+ebx+ 0]                      // xmm0 = m0, m1, m2, t0
        mulps       xmm1, [edi+ebx+16]                      // xmm1 = m3, m4, m5, t1
        mulps       xmm2, [edi+ebx+32]                      // xmm2 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        je          doneWeight

    loopWeight:
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        movaps      xmm5, [esi]
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm3, xmm5
        add         esi, BASEVECTOR_SIZE
        movaps      xmm4, xmm5
        mulps       xmm3, [edi+ebx+ 0]                      // xmm3 = m0, m1, m2, t0
        mulps       xmm4, [edi+ebx+16]                      // xmm4 = m3, m4, m5, t1
        mulps       xmm5, [edi+ebx+32]                      // xmm5 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        addps       xmm0, xmm3
        addps       xmm1, xmm4
        addps       xmm2, xmm5

        jne         loopWeight

    doneWeight:
        add         eax, VERTEX_SIZE

        movaps      xmm6, xmm0                              // xmm6 =    m0,    m1,          m2,          t0
        unpcklps    xmm6, xmm1                              // xmm6 =    m0,    m3,          m1,          m4
        unpckhps    xmm0, xmm1                              // xmm1 =    m2,    m5,          t0,          t1
        addps       xmm6, xmm0                              // xmm6 = m0+m2, m3+m5,       m1+t0,       m4+t1

        movaps      xmm7, xmm2                              // xmm7 =    m6,    m7,          m8,          t2
        movlhps     xmm2, xmm6                              // xmm2 =    m6,    m7,       m0+m2,       m3+m5
        movhlps     xmm6, xmm7                              // xmm6 =    m8,    t2,       m1+t0,       m4+t1
        addps       xmm6, xmm2                              // xmm6 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+0], xmm6

        pshufd      xmm5, xmm6, R_SHUFFLE_D( 1, 0, 2, 3 )   // xmm7 = m7+t2, m6+m8
        addss       xmm5, xmm6                              // xmm5 = m6+m8+m7+t2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+8], xmm5

        jl          loopVert
    done:
    }
}
```

Appendix B

```
/*
    SSE Optimized Skinning With Normals and Tangents
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

void TransformVertsAndTangents( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4 *base,
const JointWeight *weights, const int numWeights ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( base );

    __asm
    {
        mov         eax, numVerts
        test        eax, eax
        jz          done
        imul        eax, VERTEX_SIZE

        mov         ecx, verts
        mov         edx, weights
        mov         esi, base
        mov         edi, joints

        add         ecx, eax
        neg         eax

    loopVert:
        movss       xmm0, [edx+JOINTWEIGHT_WEIGHT_OFFSET]
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 0, 0, 0, 0 )
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm1, xmm0
        add         esi, 3*BASEVECTOR_SIZE
        movaps      xmm2, xmm0

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        mulps       xmm0, [edi+ebx+ 0]                      // xmm0 = m0, m1, m2, t0
        mulps       xmm1, [edi+ebx+16]                      // xmm1 = m3, m4, m5, t1
        mulps       xmm2, [edi+ebx+32]                      // xmm2 = m6, m7, m8, t2

        je          doneWeight

    loopWeight:
        movss       xmm3, [edx+JOINTWEIGHT_WEIGHT_OFFSET]
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        shufps      xmm3, xmm3, R_SHUFFLE_PS( 0, 0, 0, 0 )
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm4, xmm3
        movaps      xmm5, xmm3

        mulps       xmm3, [edi+ebx+ 0]                      // xmm3 = m0, m1, m2, t0
        mulps       xmm4, [edi+ebx+16]                      // xmm4 = m3, m4, m5, t1
        mulps       xmm5, [edi+ebx+32]                      // xmm5 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        addps       xmm0, xmm3
        addps       xmm1, xmm4
        addps       xmm2, xmm5

        jne         loopWeight

    doneWeight:
        add         eax, VERTEX_SIZE

        // transform vertex
        movaps      xmm3, [esi-3*BASEVECTOR_SIZE]
        movaps      xmm4, xmm3
        movaps      xmm5, xmm3

        mulps       xmm3, xmm0
        mulps       xmm4, xmm1
        mulps       xmm5, xmm2

        movaps      xmm6, xmm3                              // xmm6 =    m0,    m1,          m2,          t0
        unpcklps    xmm6, xmm4                              // xmm6 =    m0,    m3,          m1,          m4
        unpckhps    xmm3, xmm4                              // xmm4 =    m2,    m5,          t0,          t1
        addps       xmm6, xmm3                              // xmm6 = m0+m2, m3+m5,       m1+t0,       m4+t1

        movaps      xmm7, xmm5                              // xmm7 =    m6,    m7,          m8,          t2
        movlhps     xmm5, xmm6                              // xmm5 =    m6,    m7,       m0+m2,       m3+m5
        movhlps     xmm6, xmm7                              // xmm6 =    m8,    t2,       m1+t0,       m4+t1
        addps       xmm6, xmm5                              // xmm6 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+0], xmm6

        pshufd      xmm7, xmm6, R_SHUFFLE_D( 1, 0, 2, 3 )   // xmm7 = m7+t2, m6+m8
        addss       xmm7, xmm6                              // xmm7 = m6+m8+m7+t2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+8], xmm7

        // transform normal
        movaps      xmm3, [esi-2*BASEVECTOR_SIZE]
        movaps      xmm4, xmm3
        movaps      xmm5, xmm3

        mulps       xmm3, xmm0
        mulps       xmm4, xmm1
        mulps       xmm5, xmm2

        movaps      xmm6, xmm3                              // xmm6 =    m0,    m1,          m2,          t0
        unpcklps    xmm6, xmm4                              // xmm6 =    m0,    m3,          m1,          m4
        unpckhps    xmm3, xmm4                              // xmm3 =    m2,    m5,          t0,          t1
        addps       xmm6, xmm3                              // xmm6 = m0+m2, m3+m5,       m1+t0,       m4+t1

        movaps      xmm7, xmm5                              // xmm7 =    m6,    m7,          m8,          t2
        movlhps     xmm5, xmm6                              // xmm5 =    m6,    m7,       m0+m2,       m3+m5
        movhlps     xmm6, xmm7                              // xmm6 =    m8,    t2,       m1+t0,       m4+t1
        addps       xmm6, xmm5                              // xmm6 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1
        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_NORMAL_OFFSET+0], xmm6

        pshufd      xmm7, xmm6, R_SHUFFLE_D( 1, 0, 2, 3 )   // xmm7 = m7+t2, m6+m8
        addss       xmm7, xmm6                              // xmm7 = m6+m8+m7+t2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_NORMAL_OFFSET+8], xmm7

        // transform first tangent
        movaps      xmm3, [esi-1*BASEVECTOR_SIZE]

        mulps       xmm0, xmm3
        mulps       xmm1, xmm3
        mulps       xmm2, xmm3

        movaps      xmm6, xmm0                              // xmm6 =    m0,    m1,          m2,          t0
        unpcklps    xmm6, xmm1                              // xmm6 =    m0,    m3,          m1,          m4
        unpckhps    xmm0, xmm1                              // xmm1 =    m2,    m5,          t0,          t1
        addps       xmm6, xmm0                              // xmm6 = m0+m2, m3+m5,       m1+t0,       m4+t1

        movaps      xmm7, xmm2                              // xmm7 =    m6,    m7,          m8,          t2
        movlhps     xmm2, xmm6                              // xmm2 =    m6,    m7,       m0+m2,       m3+m5
        movhlps     xmm6, xmm7                              // xmm6 =    m8,    t2,       m1+t0,       m4+t1
        addps       xmm6, xmm2                              // xmm6 = m6+m8, m7+t2, m0+m1+m2+t0, m3+m4+m5+t1

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_TANGENT_OFFSET+0], xmm6

        pshufd      xmm7, xmm6, R_SHUFFLE_D( 1, 0, 2, 3 )   // xmm7 = m7+t2, m6+m8
        addss       xmm7, xmm6                              // xmm7 = m6+m8+m7+t2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_TANGENT_OFFSET+8], xmm7

        jl          loopVert
    done:
    }
}
```

Appendix C

```
/*
    SSE3 Optimized Skinning Without Normals or Tangents
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

void TransformVerts_SSE3( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4 *base, const
JointWeight *weights, const int numWeights ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( base );

    __asm
    {
        mov         eax, numVerts
        test        eax, eax
        jz          done
        imul        eax, VERTEX_SIZE

        mov         ecx, verts
        mov         edx, weights
        mov         esi, base
        mov         edi, joints

        add         ecx, eax
        neg         eax

    loopVert:
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        movaps      xmm2, [esi]
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm0, xmm2
        add         esi, BASEVECTOR_SIZE
        movaps      xmm1, xmm2

        mulps       xmm0, [edi+ebx+ 0]                      // xmm0 = m0, m1, m2, t0
        mulps       xmm1, [edi+ebx+16]                      // xmm1 = m3, m4, m5, t1
        mulps       xmm2, [edi+ebx+32]                      // xmm2 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        je          doneWeight

    loopWeight:
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        movaps      xmm5, [esi]
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm3, xmm5
        add         esi, BASEVECTOR_SIZE
        movaps      xmm4, xmm5

        mulps       xmm3, [edi+ebx+ 0]                      // xmm3 = m0, m1, m2, t0
        mulps       xmm4, [edi+ebx+16]                      // xmm4 = m3, m4, m5, t1
        mulps       xmm5, [edi+ebx+32]                      // xmm5 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        addps       xmm0, xmm3
        addps       xmm1, xmm4
        addps       xmm2, xmm5

        jne         loopWeight

    doneWeight:
        add         eax, VERTEX_SIZE

        haddps      xmm0, xmm1
        haddps      xmm2, xmm0

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+0], xmm2

        pshufd      xmm3, xmm2, R_SHUFFLE_D( 1, 0, 2, 3 )
        addss       xmm3, xmm2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+8], xmm3

        jl          loopVert
    done:
    }
}
```

Appendix D

```
/*
    SSE3 Optimized Skinning With Normals and Tangents
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

void TransformVertsAndTangents_SSE3( Vertex *verts, const int numVerts, const JointMat *joints, const Vec4
*base, const JointWeight *weights, const int numWeights ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( base );

    __asm
    {
        mov         eax, numVerts
        test        eax, eax
        jz          done
        imul        eax, VERTEX_SIZE

        mov         ecx, verts
        mov         edx, weights
        mov         esi, base
        mov         edi, joints

        add         ecx, eax
        neg         eax

    loopVert:
        movss       xmm2, [edx+JOINTWEIGHT_WEIGHT_OFFSET]
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        shufps      xmm2, xmm2, R_SHUFFLE_PS( 0, 0, 0, 0 )
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm0, xmm2
        add         esi, 3*BASEVECTOR_SIZE
        movaps      xmm1, xmm2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        mulps       xmm0, [edi+ebx+ 0]                      // xmm0 = m0, m1, m2, t0
        mulps       xmm1, [edi+ebx+16]                      // xmm1 = m3, m4, m5, t1
        mulps       xmm2, [edi+ebx+32]                      // xmm2 = m6, m7, m8, t2

        je          doneWeight

    loopWeight:
        movss       xmm5, [edx+JOINTWEIGHT_WEIGHT_OFFSET]
        mov         ebx, dword ptr [edx+JOINTWEIGHT_JOINTMATOFFSET_OFFSET]
        shufps      xmm5, xmm5, R_SHUFFLE_PS( 0, 0, 0, 0 )
        add         edx, JOINTWEIGHT_SIZE
        movaps      xmm3, xmm5
        movaps      xmm4, xmm5

        mulps       xmm3, [edi+ebx+ 0]                      // xmm3 = m0, m1, m2, t0
        mulps       xmm4, [edi+ebx+16]                      // xmm4 = m3, m4, m5, t1
        mulps       xmm5, [edi+ebx+32]                      // xmm5 = m6, m7, m8, t2

        cmp         dword ptr [edx-JOINTWEIGHT_SIZE+JOINTWEIGHT_NEXTVERTEXOFFSET_OFFSET], JOINTWEIGHT_SIZE

        addps       xmm0, xmm3
        addps       xmm1, xmm4
        addps       xmm2, xmm5

        jne         loopWeight

    doneWeight:
        add         eax, VERTEX_SIZE

        // transform vertex, normal and first tangent
        movaps      xmm3, [esi-3*BASEVECTOR_SIZE]
        movaps      xmm4, xmm3
        movaps      xmm5, xmm3

        mulps       xmm3, xmm0
        mulps       xmm4, xmm1
        mulps       xmm5, xmm2

        haddps      xmm3, xmm4

        movaps      xmm6, [esi-2*BASEVECTOR_SIZE]

        haddps      xmm5, xmm3

        movaps      xmm3, xmm6

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+0], xmm5

        pshufd      xmm4, xmm5, R_SHUFFLE_D( 1, 0, 2, 3 )
        addss       xmm4, xmm5

        movaps      xmm5, xmm6

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_POSITION_OFFSET+8], xmm4

        mulps       xmm3, xmm0
        mulps       xmm6, xmm1
        mulps       xmm5, xmm2

        movaps      xmm7, [esi-1*BASEVECTOR_SIZE]

        haddps      xmm3, xmm6

        mulps       xmm0, xmm7
        mulps       xmm1, xmm7
        mulps       xmm2, xmm7

        haddps      xmm5, xmm3

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_NORMAL_OFFSET+0], xmm5

        haddps      xmm0, xmm1

        pshufd      xmm4, xmm5, R_SHUFFLE_D( 1, 0, 2, 3 )
        addss       xmm4, xmm5

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_NORMAL_OFFSET+8], xmm4

        haddps      xmm2, xmm0

        movhps      [ecx+eax-VERTEX_SIZE+VERTEX_TANGENT_OFFSET+0], xmm2

        pshufd      xmm7, xmm2, R_SHUFFLE_D( 1, 0, 2, 3 )
        addss       xmm7, xmm2

        movss       [ecx+eax-VERTEX_SIZE+VERTEX_TANGENT_OFFSET+8], xmm7

        jl          loopVert

    done:
    }
}
```