The Skeleton Assembly Line

February 27th 2005
J.M.P. van Waveren

Abstract

Optimized routines to transform the skeletons and joints for a skeletal animation system are presented. The Intel Streaming SIMD Extensions are used to exploit parallelism with a compressed calculation. The optimized routines are well over two times faster than the implementation in C on a Pentium 4.

1. Introduction

Transformation matrices are often used when many points in space need to be transformed like the vertices of the mesh of an animated model. Matrices are typically more efficient on today's hardware when many positions in space need to be transformed. Skeletal animation systems often use row major 3x4 matrices for the joints of a skeleton. These joints are usually stored and animated in local space, relative to a parent joint, because of several advantageous properties. However, when the skeleton is used to transform the vertices of an animating mesh the joint matrices need to be transformed to global (object or world) space first.

1.1 Previous Work

Interpolating between vertex positions of two animation frames is simple and cheap. However, storing all the vertex positions for every animation frame consumes a lot of memory. The use of skeletons is more efficient when high detail polygonal meshes need to be animated. The vertex positions are only stored once and they are transformed with the joints of a skeleton. Such skeletal animation systems are not new [2, 3] and are used in many applications.

1.2 Layout

Section 2 shows some properties of the skeletons often used in skeletal animation systems. Section 3 describes how the joint matrices of a skeleton can be transformed from local space to global space. The opposite transformation from global space to local space is presented in section 4. General joint matrix multiplication as often used for matrix palette skinning is described in section 5. The results of the optimizations are presented in section 6 and several conclusions are drawn in section 7.

2. Skeletons

Transformation matrices are often used when many points in space need to be transformed like the vertices of the mesh of an animated model. Matrices are typically more efficient on today's hardware when many positions in space need to be transformed. A skeletal animation system often uses row major 3x4 matrices for the joints of a skeleton. These 3x4 matrices consist of a 3x3 orthonormal rotation matrix and a 3D translation vector.

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

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.

Each joint of a skeleton is usually stored and animated relative to a parent joint. Storing and animating joints in local space has several advantages. Some joints hardly or never change relative to their parents during certain animations which allows for better compression when joints are stored relative to their parent. Joint orientations and translations from multiple animations can also be blended together locally when joints are animated in local space. Furthermore specifying joints relative to their parents allows joints to be modified locally for instance for game controlled effects and Inverse Kinematics.

When a skeleton is used to transform the vertices of an animating mesh the joints of the skeleton need to be specified in global (object or world) space. The following section presents and optimizes a routine to transform joints matrices from local space to global space.

3. Transforming a Skeleton

The following routine transforms local joint matrices (relative to parent joints) to global joint matrices (object or world space). The joint matrices are stored in an ordered array where the parent joints come first. The routine works on this array and transforms the joint matrices in place. The parent for each joint is specified with an array with integers where each integer specifies the index of the parent joint in the array with joint matrices. Furthermore the index of the first and last joint of a sequence of joints that need to be transformed are specified. This allows the complete skeleton to be split up in multiple sequences of joints that are transformed. Joints inbetween such sequences can then be transformed separately and additional modifications can be applied where necessary.

```
void TransformJoint( JointMat &a, const idJointMat &b ) {
    float tmp[3];

    tmp[0] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 0] * b.mat[0 * 4 + 1] + a.mat[2 * 4 + 0] * b.mat[0 * 4 + 2];
    tmp[1] = a.mat[0 * 4 + 0] * b.mat[1 * 4 + 0] + a.mat[1 * 4 + 0] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 0] * b.mat[1 * 4 + 2];
    tmp[2] = a.mat[0 * 4 + 0] * b.mat[2 * 4 + 0] + a.mat[1 * 4 + 0] * b.mat[2 * 4 + 1] + a.mat[2 * 4 + 0] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 0] = tmp[0];
    a.mat[1 * 4 + 0] = tmp[1];
    a.mat[2 * 4 + 0] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 1] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 1] * b.mat[0 * 4 + 1] + a.mat[2 * 4 + 1] * b.mat[0 * 4 + 2];
    tmp[1] = a.mat[0 * 4 + 1] * b.mat[1 * 4 + 0] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 1] * b.mat[1 * 4 + 2];
    tmp[2] = a.mat[0 * 4 + 1] * b.mat[2 * 4 + 0] + a.mat[1 * 4 + 1] * b.mat[2 * 4 + 1] + a.mat[2 * 4 + 1] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 1] = tmp[0];
    a.mat[1 * 4 + 1] = tmp[1];
    a.mat[2 * 4 + 1] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 2] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 2] * b.mat[0 * 4 + 1] + a.mat[2 * 4 + 2] * b.mat[0 * 4 + 2];
    tmp[1] = a.mat[0 * 4 + 2] * b.mat[1 * 4 + 0] + a.mat[1 * 4 + 2] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 2] * b.mat[1 * 4 + 2];
    tmp[2] = a.mat[0 * 4 + 2] * b.mat[2 * 4 + 0] + a.mat[1 * 4 + 2] * b.mat[2 * 4 + 1] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 2];
```

```
    a.mat[0 * 4 + 2] = tmp[0];
    a.mat[1 * 4 + 2] = tmp[1];
    a.mat[2 * 4 + 2] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 3] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 3] * b.mat[0 * 4 + 1] + a.mat[2 * 4 + 3] * b.mat[0 * 4 + 2];
    tmp[1] = a.mat[0 * 4 + 3] * b.mat[1 * 4 + 0] + a.mat[1 * 4 + 3] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 3] * b.mat[1 * 4 + 2];
    tmp[2] = a.mat[0 * 4 + 3] * b.mat[2 * 4 + 0] + a.mat[1 * 4 + 3] * b.mat[2 * 4 + 1] + a.mat[2 * 4 + 3] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 3] = tmp[0];
    a.mat[1 * 4 + 3] = tmp[1];
    a.mat[2 * 4 + 3] = tmp[2];

    a.mat[0 * 4 + 3] += b.mat[0 * 4 + 3];
    a.mat[1 * 4 + 3] += b.mat[1 * 4 + 3];
    a.mat[2 * 4 + 3] += b.mat[2 * 4 + 3];
}

void TransformSkeleton( JointMat *jointMats, const int *parents, const int firstJoint, const int lastJoint ) {
    int i;

    for( i = firstJoint; i <= lastJoint; i++ ) {
        TransformJoint( jointMats[i], jointMats[parents[i]] );
    }
}
```

Transforming the joints of a skeleton as shown above is a typical example of a loop with a lot of mathematical operations which should be ideal for taking advantage of parallelism using SIMD code. However, there is no guarentee that the individual iterations are independent from each other. One iteration may transform the joint which is the parent of the joint transformed in the next iteration. As such parallelism cannot be exploited through increased throughput. Fortunately the transformation is ideal for exploiting parallelism with a compressed calculation. The multiplication of two 3x4 matrices involves many independent multiplications and additions that can be executed in parallel.

When multiplying matrices in SSE code it is common practice to load a single scalar from one matrix and shuffle it into all four elements of an SSE register. This register is then multiplied with a row of the other matrix. The following code shows how four scalars are loaded in this manner.

```
movss       xmm0, [esi+edx+ 0]
shufps      xmm0, xmm0, R_SHUFFLE_D( 0, 0, 0, 0 )
movss       xmm1, [esi+edx+ 4]
shufps      xmm1, xmm1, R_SHUFFLE_D( 0, 0, 0, 0 )
movss       xmm2, [esi+edx+ 8]
shufps      xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
movss       xmm3, [esi+edx+12]
shufps      xmm3, xmm3, R_SHUFFLE_D( 0, 0, 0, 0 )
```

Instead of using the above code it can be advantageous to use the SSE2 instruction 'pshufd' to load the scalars into SSE registers. Four scalars are loaded with a single move into a temporary register and then the 'pshufd' instruction is used to copy and spread the individual scalars to the other registers. 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.

```
movaps      xmm7, [esi+edx+ 0]
pshufd      xmm0, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufd      xmm1, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufd      xmm2, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
pshufd      xmm3, xmm7, R_SHUFFLE_D( 3, 3, 3, 3 )
```

Obviously when using the 'pshufd' instruction the total number of instructions required is less. However, a dependency is introduced on the temporary register into which the four scalars are loaded and on a Pentium 4 the 'pshufd' instruction trades latency for throughput. Whether or not using the 'pshufd' instruction provides an advantage depends on the surrounding instructions and the processor type. However, it is usually worthwhile to try both variants for loading scalars into SSE registers and time the results.

The SSE code for the above routine uses the 'pshufd' instruction to copy and spread scalars from one of the matrices into SSE registers. The complete SSE optimized code for transforming the joints in a skeleton can be found in appendix A.

4. Untransforming a Skeleton

In some cases it may be desired to transform the joints of a skeleton from global space back to local space. The following routine is basically the opposite of the routine presented in the previous section.

```
void UntransformJoint( JointMat &a, const idJointMat &b ) {
    float tmp[3];

    a.mat[0 * 4 + 3] -= b.mat[0 * 4 + 3];
    a.mat[1 * 4 + 3] -= b.mat[1 * 4 + 3];
    a.mat[2 * 4 + 3] -= b.mat[2 * 4 + 3];

    tmp[0] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 0] * b.mat[1 * 4 + 0] + a.mat[2 * 4 + 0] * b.mat[2 * 4 + 0];
    tmp[1] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 1] + a.mat[1 * 4 + 0] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 0] * b.mat[2 * 4 + 1];
    tmp[2] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 2] + a.mat[1 * 4 + 0] * b.mat[1 * 4 + 2] + a.mat[2 * 4 + 0] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 0] = tmp[0];
    a.mat[1 * 4 + 0] = tmp[1];
    a.mat[2 * 4 + 0] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 1] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 0] + a.mat[2 * 4 + 1] * b.mat[2 * 4 + 0];
    tmp[1] = a.mat[0 * 4 + 1] * b.mat[0 * 4 + 1] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 1] * b.mat[2 * 4 + 1];
    tmp[2] = a.mat[0 * 4 + 1] * b.mat[0 * 4 + 2] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 2] + a.mat[2 * 4 + 1] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 1] = tmp[0];
    a.mat[1 * 4 + 1] = tmp[1];
    a.mat[2 * 4 + 1] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 2] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 2] * b.mat[1 * 4 + 0] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 0];
    tmp[1] = a.mat[0 * 4 + 2] * b.mat[0 * 4 + 1] + a.mat[1 * 4 + 2] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 1];
    tmp[2] = a.mat[0 * 4 + 2] * b.mat[0 * 4 + 2] + a.mat[1 * 4 + 2] * b.mat[1 * 4 + 2] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 2] = tmp[0];
    a.mat[1 * 4 + 2] = tmp[1];
    a.mat[2 * 4 + 2] = tmp[2];

    tmp[0] = a.mat[0 * 4 + 3] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 3] * b.mat[1 * 4 + 0] + a.mat[2 * 4 + 3] * b.mat[2 * 4 + 0];
    tmp[1] = a.mat[0 * 4 + 3] * b.mat[0 * 4 + 1] + a.mat[1 * 4 + 3] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 3] * b.mat[2 * 4 + 1];
    tmp[2] = a.mat[0 * 4 + 3] * b.mat[0 * 4 + 2] + a.mat[1 * 4 + 3] * b.mat[1 * 4 + 2] + a.mat[2 * 4 + 3] * b.mat[2 * 4 + 2];
    a.mat[0 * 4 + 3] = tmp[0];
    a.mat[1 * 4 + 3] = tmp[1];
    a.mat[2 * 4 + 3] = tmp[2];
}

void UntransformSkeleton( JointMat *jointMats, const int *parents, const int firstJoint, const int lastJoint ) {
    int i;

    for( i = lastJoint; i >= firstJoint; i-- ) {
        UntransformJoint( jointMats[i], jointMats[parents[i]] );
    }
}
```

The exact same approach is used for the implementation of the SSE optimized code for the above routine and the routine specified in the previous section. The complete SSE optimized code for the above routine can be found in appendix B.

5. Transforming Joints

One approach to matrix palette skinning involves transforming a base pose of a mesh. The vertices for this base pose are stored in model space. 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 animations. The following routine can be used to multiply the joint matrices of the animated skeleton with the precalculated inverse joint matrices of the base pose. The routine multiplies two arrays with joint matrices and stores the result in another array.

```
void MultiplyJoints( JointMat &result, const JointMat &a, const JointMat &b ) {
    result.mat[0 * 4 + 0] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 0] + a.mat[0 * 4 + 1] * b.mat[1 * 4 + 0] + a.mat[0 * 4 + 2] * b.mat[2 * 4 + 0];
    result.mat[0 * 4 + 1] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 1] + a.mat[0 * 4 + 1] * b.mat[1 * 4 + 1] + a.mat[0 * 4 + 2] * b.mat[2 * 4 + 1];
    result.mat[0 * 4 + 2] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 2] + a.mat[0 * 4 + 1] * b.mat[1 * 4 + 2] + a.mat[0 * 4 + 2] * b.mat[2 * 4 + 2];
    result.mat[0 * 4 + 3] = a.mat[0 * 4 + 0] * b.mat[0 * 4 + 3] + a.mat[0 * 4 + 1] * b.mat[1 * 4 + 3] + a.mat[0 * 4 + 2] * b.mat[2 * 4 + 3]
+ a.mat[0 * 4 + 3];

    result.mat[1 * 4 + 0] = a.mat[1 * 4 + 0] * b.mat[0 * 4 + 0] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 0] + a.mat[1 * 4 + 2] * b.mat[2 * 4 + 0];
    result.mat[1 * 4 + 1] = a.mat[1 * 4 + 0] * b.mat[0 * 4 + 1] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 1] + a.mat[1 * 4 + 2] * b.mat[2 * 4 + 1];
    result.mat[1 * 4 + 2] = a.mat[1 * 4 + 0] * b.mat[0 * 4 + 2] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 2] + a.mat[1 * 4 + 2] * b.mat[2 * 4 + 2];
    result.mat[1 * 4 + 3] = a.mat[1 * 4 + 0] * b.mat[0 * 4 + 3] + a.mat[1 * 4 + 1] * b.mat[1 * 4 + 3] + a.mat[1 * 4 + 2] * b.mat[2 * 4 + 3]
+ a.mat[1 * 4 + 3];

    result.mat[2 * 4 + 0] = a.mat[2 * 4 + 0] * b.mat[0 * 4 + 0] + a.mat[2 * 4 + 1] * b.mat[1 * 4 + 0] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 0];
    result.mat[2 * 4 + 1] = a.mat[2 * 4 + 0] * b.mat[0 * 4 + 1] + a.mat[2 * 4 + 1] * b.mat[1 * 4 + 1] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 1];
    result.mat[2 * 4 + 2] = a.mat[2 * 4 + 0] * b.mat[0 * 4 + 2] + a.mat[2 * 4 + 1] * b.mat[1 * 4 + 2] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 2];
    result.mat[2 * 4 + 3] = a.mat[2 * 4 + 0] * b.mat[0 * 4 + 3] + a.mat[2 * 4 + 1] * b.mat[1 * 4 + 3] + a.mat[2 * 4 + 2] * b.mat[2 * 4 + 3]
+ a.mat[2 * 4 + 3];
}

void TransformJoints( JointMat *result, const JointMat *joints1, const JointMat *joints2, const int numJoints ) {
    int i;

    for ( i = 0; i < numJoints; i++ ) {
        MultiplyJoints( result[i], joints1[i], joints2[i] );
    }
}
```

Although this routine has no dependencies between iterations, the best way to exploit parallelism is still through a compressed calculation. It is easy to spot the independent multiplications and additions that can be executed in parallel. The same approach to SIMD as used for the previous routines can be used for this routine. The complete SSE optimized code for the above routine can be found in appendix B.

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 joints. The total number of clock cycles and the number of clock cycles per joint 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**
TransformSkeleton (C)	132224	130	166176	163
TransformSkeleton (SSE)	53580	53	54297	53
UntransformSkeleton (C)	133680	131	157104	154
UntransformSkeleton (SSE)	55488	55	57285	56
TransformJoints (C)	137980	135	151587	148
TransformJoints (SSE)	36520	36	48906	48

7. Conclusion

Transforming the joints of a skeleton involves the multiplication of 3x4 matrices. The transformations involve many independent multiplications and additions that can be executed in parallel and this parallelism can be exploited with a compressed calculation using the Intel Streaming SIMD Extensions.

The SSE optimized implementations are significantly faster than their C counterparts. In all cases the transformations are at least two times faster and in some cases close to three times faster.

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

Appendix A

```
/*
    SSE Optimized Skeleton Transform for Skeletal Animation Systems
    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.
*/

#define assert_16_byte_aligned( pointer )   assert( (((UINT_PTR)(pointer))&15) == 0 );
#define ALIGN16( x )                        __declspec(align(16)) x
#define ALIGN4_INIT4( X, I0, I1, I2, I3 )   ALIGN16( static X[4] ) = { I0, I1, I2, I3 }
#define R_SHUFFLE_D( x, y, z, w )           (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | ( (y) & 3 ) << 2 | ( (x) & 3 ))

ALIGN4_INIT4( unsigned long SIMD_SP_clearFirstThree, 0x00000000, 0x00000000, 0x00000000, 0xFFFFFFFF );

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

#define JOINTMAT_SIZE           (4*3*4)

void TransformSkeleton( JointMat *jointMats, const int *parents, const int firstJoint, const int lastJoint ) {

    assert_16_byte_aligned( jointMats );

    __asm {

        mov         ecx, firstJoint
        mov         eax, lastJoint
        sub         eax, ecx
        jl          done
        shl         ecx, 2                                  // ecx = firstJoint * 4
        mov         edi, parents
        add         edi, ecx                                // edx = &parents[firstJoint]
        lea         ecx, [ecx+ecx*2]
        shl         ecx, 2                                  // ecx = firstJoint * JOINTMAT_SIZE
        mov         esi, jointMats                          // esi = jointMats
        shl         eax, 2                                  // eax = ( lastJoint - firstJoint ) * 4
        add         edi, eax
        neg         eax

    loopJoint:

        mov         edx, [edi+eax]
        movaps      xmm0, [esi+ecx+ 0]                      // xmm0 = m0, m1, m2, t0
        lea         edx, [edx+edx*2]
        movaps      xmm1, [esi+ecx+16]                      // xmm1 = m2, m3, m4, t1
        shl         edx, 4                                  // edx = parents[i] * JOINTMAT_SIZE
        movaps      xmm2, [esi+ecx+32]                      // xmm2 = m5, m6, m7, t2

        movaps      xmm7, [esi+edx+ 0]
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm4, xmm0
        pshufd      xmm5, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm5, xmm1
        addps       xmm4, xmm5
```

```
        add         ecx, JOINTMAT_SIZE
        add         eax, 4

        pshufd      xmm6, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm6, xmm2
        addps       xmm4, xmm6
        andps       xmm7, SIMD_SP_clearFirstThree
        addps       xmm4, xmm7

        movaps      [esi+ecx-JOINTMAT_SIZE+ 0], xmm4

        movaps      xmm3, [esi+edx+16]
        pshufd      xmm5, xmm3, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm5, xmm0
        pshufd      xmm6, xmm3, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm6, xmm1
        addps       xmm5, xmm6
        pshufd      xmm4, xmm3, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm4, xmm2
        addps       xmm5, xmm4
        andps       xmm3, SIMD_SP_clearFirstThree
        addps       xmm5, xmm3

        movaps      [esi+ecx-JOINTMAT_SIZE+16], xmm5

        movaps      xmm7, [esi+edx+32]
        pshufd      xmm6, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm6, xmm0
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm4, xmm1
        addps       xmm6, xmm4
        pshufd      xmm3, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm3, xmm2
        addps       xmm6, xmm3
        andps       xmm7, SIMD_SP_clearFirstThree
        addps       xmm6, xmm7

        movaps      [esi+ecx-JOINTMAT_SIZE+32], xmm6

        jle         loopJoint
    done:
    }
}
```

Appendix B

```
/*
    SSE Optimized Skeleton Untransform for Skeletal Animation Systems
    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 UntransformSkeleton( JointMat *jointMats, const int *parents, const int firstJoint, const int lastJoint ) {

    assert_16_byte_aligned( jointMats );

    __asm {

        mov         edx, firstJoint
        mov         eax, lastJoint
        mov         ecx, eax
        sub         eax, edx
        jl          done
        mov         esi, jointMats                          // esi = jointMats
        lea         ecx, [ecx+ecx*2]
        shl         ecx, 4                                  // ecx = lastJoint * JOINTMAT_SIZE
        shl         edx, 2
        mov         edi, parents
        add         edi, edx                                // edi = &parents[firstJoint]
```

```
        shl         eax, 2                                  // eax = ( lastJoint - firstJoint ) * 4

    loopJoint:

        mov         edx, [edi+eax]
        movaps      xmm0, [esi+ecx+ 0]                      // xmm0 = m0, m1, m2, t0
        lea         edx, [edx+edx*2]
        movaps      xmm1, [esi+ecx+16]                      // xmm1 = m2, m3, m4, t1
        shl         edx, 4                                  // edx = parents[i] * JOINTMAT_SIZE
        movaps      xmm2, [esi+ecx+32]                      // xmm2 = m5, m6, m7, t2

        movss       xmm7, [esi+edx+12]
        pshufd      xmm7, xmm7, R_SHUFFLE_D( 1, 2, 3, 0 )
        subps       xmm0, xmm7
        movss       xmm6, [esi+edx+28]
        pshufd      xmm6, xmm6, R_SHUFFLE_D( 1, 2, 3, 0 )
        subps       xmm1, xmm6
        movss       xmm5, [esi+edx+44]
        pshufd      xmm5, xmm5, R_SHUFFLE_D( 1, 2, 3, 0 )
        subps       xmm2, xmm5

        sub         ecx, JOINTMAT_SIZE
        sub         eax, 4

        movaps      xmm7, [esi+edx+ 0]

        pshufd      xmm3, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm3, xmm0
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm4, xmm0
        pshufd      xmm5, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm5, xmm0

        movaps      xmm7, [esi+edx+16]

        pshufd      xmm0, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm0, xmm1
        addps       xmm3, xmm0
        pshufd      xmm6, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm6, xmm1
        addps       xmm4, xmm6
        pshufd      xmm0, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm0, xmm1
        addps       xmm5, xmm0

        movaps      xmm7, [esi+edx+32]

        pshufd      xmm6, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm6, xmm2
        addps       xmm3, xmm6

        movaps      [esi+ecx+JOINTMAT_SIZE+ 0], xmm3

        pshufd      xmm1, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm1, xmm2
        addps       xmm4, xmm1

        movaps      [esi+ecx+JOINTMAT_SIZE+16], xmm4

        pshufd      xmm6, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm6, xmm2
        addps       xmm5, xmm6

        movaps      [esi+ecx+JOINTMAT_SIZE+32], xmm5

        jge         loopJoint
    done:
    }
}
```

Appendix C

```
/*
    SSE Optimized Joint Transform for Skeletal Animation Systems
    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 TransformJoints( JointMat *result, const JointMat *joints1, const JointMat *joints2, const int numJoints ) {

    assert_16_byte_aligned( result );
    assert_16_byte_aligned( joints1 );
    assert_16_byte_aligned( joints2 );

    __asm {

        mov         eax, numJoints
        test        eax, eax
        jz          done
        mov         ecx, joints1
        mov         edx, joints2
        mov         edi, result
        imul        eax, JOINTMAT_SIZE
        add         ecx, eax
        add         edx, eax
        add         edi, eax
        neg         eax

    loopJoint:

        movaps      xmm0, [edx+eax+ 0]
        movaps      xmm1, [edx+eax+16]
        movaps      xmm2, [edx+eax+32]

        movaps      xmm7, [ecx+eax+ 0]
        pshufd      xmm3, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm3, xmm0
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm4, xmm1
        addps       xmm3, xmm4

        add         eax, JOINTMAT_SIZE

        pshufd      xmm5, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm5, xmm2
        addps       xmm3, xmm5
        andps       xmm7, SIMD_SP_clearFirstThree
        addps       xmm3, xmm7

        movaps      [edi+eax-JOINTMAT_SIZE+0], xmm3

        movaps      xmm7, [ecx+eax-JOINTMAT_SIZE+16]
        pshufd      xmm3, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm3, xmm0
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm4, xmm1
        addps       xmm3, xmm4
        pshufd      xmm5, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm5, xmm2
        addps       xmm3, xmm5
        andps       xmm7, SIMD_SP_clearFirstThree
        addps       xmm3, xmm7

        movaps      [edi+eax-JOINTMAT_SIZE+16], xmm3

        movaps      xmm7, [ecx+eax-JOINTMAT_SIZE+32]
        pshufd      xmm3, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
        mulps       xmm3, xmm0
        pshufd      xmm4, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
        mulps       xmm4, xmm1
```

```
        addps       xmm3, xmm4
        pshufd      xmm5, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
        mulps       xmm5, xmm2
        addps       xmm3, xmm5
        andps       xmm7, SIMD_SP_clearFirstThree
        addps       xmm3, xmm7

        movaps      [edi+eax-JOINTMAT_SIZE+32], xmm3

        jl          loopJoint
    done:
    }
}
```