From Quaternion to Matrix and Back

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

Abstract

Optimized routines for the conversion between quaternions and matrices are presented. First the regular C/C++ routines presented in literature are optimized and/or restructured to make it easier for the compiler to generate optimized assembler code. Next the best approach to SIMD is determined and the SIMD optimizations are partially prototyped in regular C/C++ code. Finally the Intel Streaming SIMD Extensions are used to get the most out of every clock cycle.

1. Introduction

Quaternions are often used in skeletal animation systems for the interpolation between general rotations. When interpolating between animation key frames quaternions provide an efficient means to interpolate the general rotations of joints in a skeleton. However, matrices are more efficient when many points or vertices need to be transformed, and the joints in a skeleton typically transform many vertices of a polygonal mesh. As such the desire arises to convert quaternions to matrices. Sometimes it may also be desired to modify a skeleton using matrices. Therefore it may also be useful to convert matrices to quaternions.

1.1 Previous Work

The quaternion was first introduced by William Rowan Hamilton (1805 - 1865) as a successor to complex numbers [1]. Arthur Cayley (1821 - 1895) contributed further by describing rotations with quaternion multiplication [2]. Ken Shoemake popularized quaternions in the world of computer graphics [6]. Quaternions have since found their way into many different systems among which animation, inverse kinematics and physics.

In skeletal animation systems quaternions are often used to interpolate between joint orientations specified with key frames or animation curves [7,9,10]. On the other hand rotation matrices are often used when many points in space need to be transformed like the vertices of the skin of an animated model. Rotation matrices are typically more efficient on today's hardware when many positions need to be transformed. Because both quaternions and rotation matrices are useful and efficient for certain calculations the desire arises to convert between these representations. These conversions were introduced by Ken Shoemake [6,7,8] in the context of computer graphics.

1.2 Layout

Section 2 shows some properties of quaternions and rotation matrices. Section 3 describes the conversion from joint quaternions to joint matrices. The conversion from joint matrices to joint quaternions is presented in section 4. The results of the optimizations are presented in section 5 and several conclusions are drawn in section 6.

2. Quaternions and Rotation Matrices.

The unit quaternion sphere is equivalent to the space of general rotations. Throughout this article quaternions will represent general rotations. The four components of a quaternion are denoted (x, y, z, w) and the quaternion will be represented in code as follows.

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

A quaternion (x, y, z, w) which represents a general rotation can be interpreted geometrically as follows.

$$x = X \cdot \sin(\alpha / 2)$$
$$y = Y \cdot \sin(\alpha / 2)$$
$$z = Z \cdot \sin(\alpha / 2)$$
$$w = \cos(\alpha / 2)$$

Here (X, Y, Z) is the unit length axis of rotation in 3D space and α is the angle of rotation about the axis in radians.

A general rotation can also be defined with a 3x3 orthonormal matrix. Each row and each column of the matrix is a 3D vector of unit length. The rows of the matrix are orthogonal to each other and the same goes for the columns.

Quaternions and rotation matrices are often used in skeletal animation systems to describe the orientation and translation of joints in a skeleton. Joints using a quaternion for the orientation will be represented in code as follows.

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

struct JointQuat {
    Quaternion  q;
    Vec4        t;
};
```

Joints using a rotation matrix for the orientation will be represented in code as follows.

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

This is a 3x4 matrix where the first three elements of each row are from a row-major rotation matrix and the last element of every row is the translation over one axis.

3. Quaternion to Matrix

For the quaternion (x, y, z, w) the corresponding rotation matrix M is defined as follows [6].

$$M = \begin{vmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{vmatrix}$$

By grouping the common products the joint quaternion to joint matrix conversion can be implemented as follows.

```
void ConvertJointQuatsToJointMats( JointMat *jointMats, const JointQuat *jointQuats, const int numJoints ) {

    for ( int i = 0; i < numJoints; i++ ) {
        const float *q = &jointQuats[i].q;
        float *m = jointMats[i].mat;

        m[0*4+3] = q[4];
        m[1*4+3] = q[5];
        m[2*4+3] = q[6];

        float x2 = q[0] + q[0];
        float y2 = q[1] + q[1];
        float z2 = q[2] + q[2];
        {
            float xx2 = q[0] * x2;
            float yy2 = q[1] * y2;
            float zz2 = q[2] * z2;

            m[0*4+0] = 1.0f - yy2 - zz2;
            m[1*4+1] = 1.0f - xx2 - zz2;
            m[2*4+2] = 1.0f - xx2 - yy2;
        }
        {
            float yz2 = q[1] * z2;
            float wx2 = q[3] * x2;

            m[2*4+1] = yz2 - wx2;
            m[1*4+2] = yz2 + wx2;
        }
        {
            float xy2 = q[0] * y2;
            float wz2 = q[3] * z2;

            m[1*4+0] = xy2 - wz2;
            m[0*4+1] = xy2 + wz2;
        }
        {
            float xz2 = q[0] * z2;
            float wy2 = q[3] * y2;

            m[0*4+2] = xz2 - wy2;
            m[2*4+0] = xz2 + wy2;
        }
    }
}
```

The above routine localizes variable dependencies with additional braces to make it easier for the compiler to produce optimized FPU code.

One thing becomes immediately apparent when examining the above routine. The number of mathematical operations is minimal compared to the number of data move operations. Furthermore the way the quaternion components are scattered into a matrix makes it hard to exploit parallelism through increased throughput. The required swizzle of the quaternion components and de-swizzle of the calculated matrix elements easily nullifies any gain from executing four operations at once for the few mathematical operations used in the conversion.

Instead of exploiting parallelism through increased throughput, parallelism can also be exploited with a compressed calculation. As it turns out it is not that hard to find common operations that can be executed in parallel, but it is not trivial to arrange them in such a way that consecutive operations in the conversion can be executed with SIMD instructions without requiring excessive swizzling. However, the following prototype can be constructed which has several advantageous properties.

```
void ConvertJointQuatsToJointMats( JointMat *jointMats, const JointQuat *jointQuats, const int numJoints ) {

    for ( int i = 0; i < numJoints; i++ ) {
        const float *q = &jointQuats[i].q;
        float *m = jointMats[i].mat;

        float x2 = q[0] + q[0];
        float y2 = q[1] + q[1];
        float z2 = q[2] + q[2];
        float w2 = q[3] + q[3];

        float yy2 = q[1] * y2;
        float xy2 = q[0] * y2;
        float xz2 = q[0] * z2;
        float yz2 = q[1] * z2;

        float zz2 = q[2] * z2;
        float wz2 = q[3] * z2;
        float wy2 = q[3] * y2;
        float wx2 = q[3] * x2;

        float xx2 = q[0] * x2;

        m[0*4+0] = - yy2 - zz2 + 1.0f;
        m[0*4+1] =   xy2 + wz2;
        m[0*4+2] =   xz2 - wy2;
        m[0*4+3] = q[4];

        m[1*4+0] =   xy2 - wz2;
        m[1*4+1] = - xx2 - zz2 + 1.0f;
        m[1*4+2] =   yz2 + wx2;
        m[1*4+3] = q[5];

        m[2*4+0] =   xz2 + wy2;
        m[2*4+1] =   yz2 - wx2;
        m[2*4+2] = - xx2 - yy2 + 1.0f;
        m[2*4+3] = q[6];
    }
}
```

The above routine should not be used as a replacement for the former routine because it is significantly slower when compiled to FPU code. However, the above routine does provide a good starting point for an SSE optimized version.

The conversion counts 9 multiplications that can be executed with three SSE instructions. Because of the way the multiplications are arranged in the above routine, the first row of the matrix can be calculated directly from the first 8 products. The second row can be calculated by replacing one of the first 8 products with the 9th product. As such the swizzling required during the conversion is minimized. Because the elements of the first two rows are calculated by adding and subtracting products, the sign of some of the products is changed with the 'xorps' instruction which allows a single 'subps' instruction to be used per row. Only the first three elements of the first two rows are calculated from the 9 products. Because of the way the products are arranged the 'subps' instructions used for the first two rows also calculate two elements for the last row in the fourth elements of the SSE registers. The last diagonal element is then calculated separately and combined with these fourth elements to form the third row.

The complete SSE optimized code for the conversion can be found in appendix A. The code assumes that both the list with joints and the list with matrices are at least 16 byte aligned.

The SSE2 instruction 'pshufd' is used to swizzle the quaternion components before multiplying them. This 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 advantage of using the 'pshufd' instruction is that the complete contents of one SSE register can be copied and swizzled into another SSE register.

4. Matrix to Quaternion

Converting a rotation matrix to a quaternion is a bit more challenging. The quaternion components always appear in pairs in the rotation matrix and some manipulation is required to extract them. To avoid sign loss only one component of the quaternion is extracted using the diagonal and divided into cross-diagonal sums. The algorithm avoids precision loss due to near-zero divides by looking for a component of large magnitude as divisor, first w, then x, y, or z. When the trace of the matrix (sum of diagonal elements) is greater than zero, |w| is greater than 1/2, which is as small as the largest component can be. Otherwise, the largest diagonal element corresponds to the largest of |x|, |y|, or |z|, one of which must be larger than |w|, and at least 1/2. The following routine converts JointQuats to JointMats using the quaternion to matrix conversion.

```
float ReciprocalSqrt( float x ) {
    long i;
    float y, r;

    y = x * 0.5f;
    i = *(long *)( &x );
    i = 0x5f3759df - ( i >> 1 );
    r = *(float *)( &i );
    r = r * ( 1.5f - r * r * y );
    return r;
}

void ConvertJointMatsToJointQuats( JointQuat *jointQuats, const JointMat *jointMats, const int numJoints ) {

    for ( int i = 0; i < numJoints; i++ ) {

        float *q = &jointQuats[i].q;
        const float *m = jointMats[i].mat;

        if ( m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] > 0.0f ) {

            float t = + m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] + 1.0f;
            float s = ReciprocalSqrt( t ) * 0.5f;

            q[3] = s * t;
            q[2] = ( m[0 * 4 + 1] - m[1 * 4 + 0] ) * s;
            q[1] = ( m[2 * 4 + 0] - m[0 * 4 + 2] ) * s;
            q[0] = ( m[1 * 4 + 2] - m[2 * 4 + 1] ) * s;

        } else if ( m[0 * 4 + 0] > m[1 * 4 + 1] && m[0 * 4 + 0] > m[2 * 4 + 2] ) {

            float t = + m[0 * 4 + 0] - m[1 * 4 + 1] - m[2 * 4 + 2] + 1.0f;
            float s = ReciprocalSqrt( t ) * 0.5f;

            q[0] = s * t;
            q[1] = ( m[0 * 4 + 1] + m[1 * 4 + 0] ) * s;
            q[2] = ( m[2 * 4 + 0] + m[0 * 4 + 2] ) * s;
            q[3] = ( m[1 * 4 + 2] - m[2 * 4 + 1] ) * s;

        } else if ( m[1 * 4 + 1] > m[2 * 4 + 2] ) {

            float t = - m[0 * 4 + 0] + m[1 * 4 + 1] - m[2 * 4 + 2] + 1.0f;
            float s = ReciprocalSqrt( t ) * 0.5f;

            q[1] = s * t;
            q[0] = ( m[0 * 4 + 1] + m[1 * 4 + 0] ) * s;
            q[3] = ( m[2 * 4 + 0] - m[0 * 4 + 2] ) * s;
            q[2] = ( m[1 * 4 + 2] + m[2 * 4 + 1] ) * s;

        } else {

            float t = - m[0 * 4 + 0] - m[1 * 4 + 1] + m[2 * 4 + 2] + 1.0f;
            float s = ReciprocalSqrt( t ) * 0.5f;

            q[2] = s * t;
            q[3] = ( m[0 * 4 + 1] - m[1 * 4 + 0] ) * s;
            q[0] = ( m[2 * 4 + 0] + m[0 * 4 + 2] ) * s;
            q[1] = ( m[1 * 4 + 2] + m[2 * 4 + 1] ) * s;

        }

        q[4] = m[0 * 4 + 3];
        q[5] = m[1 * 4 + 3];
        q[6] = m[2 * 4 + 3];
        q[7] = 0.0f;
    }
}
```

The above routine may appear to be quite different from the commonly used implementation as presented by Ken Shoemake [6]. However, the above routine just unrolls the four cases for the different divisors. The routine is typically faster because it does not use any variable indexing into arrays. The above routine also uses a fast reciprocal square root approximation [14,15,16].

When examining the above code a key observation can be made. The code for each of the four cases is almost the same. The only differences are a couple of signs and the order in which the components of the quaternion are stored. To emphasize these differences the above routine can be rewritten to the following routine.

```
void ConvertJointMatsToJointQuats( JointQuat *jointQuats, const JointMat *jointMats, const int numJoints ) {

    for ( int i = 0; i < numJoints; i++ ) {
        float s0, s1, s2;
        int k0, k1, k2, k3;

        float *q = &jointQuats[i].q;
        const float *m = jointMats[i].mat;

        if ( m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] > 0.0f ) {

            k0 = 3;
            k1 = 2;
            k2 = 1;
            k3 = 0;
            s0 = 1.0f;
            s1 = 1.0f;
            s2 = 1.0f;

        } else if ( m[0 * 4 + 0] > m[1 * 4 + 1] && m[0 * 4 + 0] > m[2 * 4 + 2] ) {

            k0 = 0;
            k1 = 1;
            k2 = 2;
            k3 = 3;
            s0 = 1.0f;
            s1 = -1.0f;
            s2 = -1.0f;

        } else if ( m[1 * 4 + 1] > m[2 * 4 + 2] ) {
            k0 = 1;
            k1 = 0;
            k2 = 3;
            k3 = 2;
            s0 = -1.0f;
            s1 = 1.0f;
            s2 = -1.0f;

        } else {

            k0 = 2;
            k1 = 3;
            k2 = 0;
            k3 = 1;
            s0 = -1.0f;
            s1 = -1.0f;
            s2 = 1.0f;

        }

        float t = s0 * m[0 * 4 + 0] + s1 * m[1 * 4 + 1] + s2 * m[2 * 4 + 2] + 1.0f;
        float s = ReciprocalSqrt( t ) * 0.5f;

        q[k0] = s * t;
        q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;
        q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;
        q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        q[4] = m[0 * 4 + 3];
        q[5] = m[1 * 4 + 3];
        q[6] = m[2 * 4 + 3];
        q[7] = 0.0f;
    }
}
```

In the above code each case sets 4 indices (k0, k1, k2, k3) and three sign multipliers (s0, s1, s2). The indices are used to determine the order in which the different quaternion components are stored and the sign multipliers are used to change the signs in the calculation. The above routine should not be used as a replacement for the former routine because it is significantly slower when compiled to FPU code. However, the above routine does provide a blue print for an SSE optimized version.

The best approach to SIMD for the joint matrix to joint quaternion conversion is to exploit parallelism through increased throughput. The routine presented here will operate on four conversion per iteration and the scalar instructions are replaced with functionally equivalent SSE instructions. This requires a swizzle because the matrices are stored per joint while some of the individual elements of four matrices need to be grouped into SSE registers. Furthermore the conditionally executed code for the four different cases has to be replaced with a single sequence of instructions for all cases.

The initial swizzle loads the diagonal elements of four matrices into three SSE registers. The swizzle loads one element at a time and shuffles it into one of the SSE registers. The diagonal elements are stored in the xmm5, xmm6 and xmm7 register. Based on the diagonal elements the three conditions are evaluated and the results are stored in the xmm0, xmm2, and xmm4 register as follows:

```
movaps      xmm0, xmm5
addps       xmm0, xmm6
addps       xmm0, xmm7
cmpnltps    xmm0, SIMD_SP_zero    // xmm0 = m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] > 0.0f

movaps      xmm1, xmm5
movaps      xmm2, xmm5
cmpnltps    xmm1, xmm6
cmpnltps    xmm2, xmm7
andps       xmm2, xmm1            // xmm2 = m[0 * 4 + 0] > m[1 * 4 + 1] && m[0 * 4 + 0] > m[2 * 4 + 2]
```

```
movaps      xmm4, xmm6
cmpnltps    xmm4, xmm7            // xmm4 = m[1 * 4 + 1] > m[2 * 4 + 2]
```

From the three conditions four masks are calculated for the four cases. These masks are stored in the xmm0, xmm1, xmm2 and xmm3 register. Based on the chosen divisor only one of these registers will be filled with all one bits and the other registers will be all zeros. The masks are calculated as follows.

```
movaps      xmm1, xmm0
andnps      xmm1, xmm2
orps        xmm2, xmm0
movaps      xmm3, xmm2
andnps      xmm2, xmm4
orps        xmm3, xmm2
xorps       xmm3, SIMD_SP_not
```

The components of a quaternion are stored in a different order based on the chosen divisor. The indices k0 through k3 in the C/C++ blue print basically specify a swizzle to store the components of a quaternion. The correct swizzle corresponding to the chosen divisor can be selected using the four masks calculated above. The four different swizzles are stored as 8 bit indices in 16 byte constants as follows.

```
#define ALIGN4_INIT1( X, I ) __declspec(align(16)) static X[4] = { I, I, I, I }

ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle0, (3<<0)|(2<<8)|(1<<16)|(0<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle1, (0<<0)|(1<<8)|(2<<16)|(3<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle2, (1<<0)|(0<<8)|(3<<16)|(2<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle3, (2<<0)|(3<<8)|(0<<16)|(1<<24) );
```

One of the swizzles can be selected by using a binary 'and' of each of the above swizzle constants with one of the four masks and using a binary 'or' on the results. The following SSE code selects one of the swizzles for each of the four conversions and stores the result in a local byte array called 'shuffle'.

```
ALIGN16( byte shuffle[16]; )

andps       xmm0, SIMD_DW_mat2quatShuffle0
movaps      xmm4, xmm1
andps       xmm4, SIMD_DW_mat2quatShuffle1
orps        xmm0, xmm4
movaps      xmm4, xmm2
andps       xmm4, SIMD_DW_mat2quatShuffle2
orps        xmm0, xmm4
movaps      xmm4, xmm3
andps       xmm4, SIMD_DW_mat2quatShuffle3
orps        xmm4, xmm0
movaps      shuffle, xmm4
```

Next to the swizzle the three signs for each of the four cases need to be calculated as well. The following SSE code calculates sign bits from the four masks for the four conversions and stores them in the xmm0, xmm1 and xmm2 register.

```
ALIGN4_INIT1( unsigned long SIMD_SP_signBit, IEEE_SP_SIGN );

movaps      xmm0, xmm2
orps        xmm0, xmm3                  // xmm0 = xmm2 | xmm3   = s0
orps        xmm2, xmm1                  // xmm2 = xmm1 | xmm2   = s2
orps        xmm1, xmm3                  // xmm1 = xmm1 | xmm3   = s1
andps       xmm0, SIMD_SP_signBit
andps       xmm1, SIMD_SP_signBit
andps       xmm2, SIMD_SP_signBit
```

The scalar instructions of the first part of the conversion can now be replaced with functionally equivalent SSE instructions. The 'xorps' instruction can be used with the three sign bits for each of the four conversions to flip the signs of some of the matrix elements.

Intel SSE instruction set has an instruction to calculate the reciprocal square root with 12 bits of precision. A simple Newton-Rapson iteration can be used to improve the accuracy [17]. The following assembler code calculates the reciprocal square root of the four floating point numbers stored in the 'xmm5' register. The result is stored in the 'xmm6' register.

```
ALIGN4_INIT1( float SIMD_SP_rsqrt_c0,  3.0f );
ALIGN4_INIT1( float SIMD_SP_rsqrt_c1, -0.5f );

rsqrtps     xmm6, xmm5
mulps       xmm5, xmm6
mulps       xmm5, xmm6
subps       xmm5, SIMD_SP_rsqrt_c0
mulps       xmm6, SIMD_SP_rsqrt_c1
mulps       xmm6, xmm5
```

The conversion uses the reciprocal square root multiplied with a half. As such the second constant of the Newton-Rapson iteration is pre-multiplied with a half to get half the reciprocal square root at no additional cost.

```
ALIGN4_INIT1( float SIMD_SP_rsqrt_c0,  3.0f );
ALIGN4_INIT1( float SIMD_SP_mat2quat_rsqrt_c1, -0.5f * 0.5f );

rsqrtps     xmm6, xmm5
mulps       xmm5, xmm6
mulps       xmm5, xmm6
subps       xmm5, SIMD_SP_rsqrt_c0
mulps       xmm6, SIMD_SP_mat2quat_rsqrt_c1
mulps       xmm6, xmm5
```

SSE scalar code is used for the last part of the conversion that uses the off-diagonal elements of the matrix. For this part of the conversion it does not pay off to use SIMD instructions because the swizzle and de-swizzle required to pack and unpack the off-diagonal elements would nullify any gains from executing four operations at once.

To store the components of the quaternion the 'shuffle' byte array is used to get the correct index for the chosen divisor. The index is loaded into a general purpose register and used to get the address of the quaternion component.

```
movzx       ecx, byte ptr shuffle[0*4+0]          // ecx = k0
movss       [edi+ecx*4-4*JOINTQUAT_SIZE], xmm7    // q[k0] = s * t;
```

The complete routine for the conversion from joint matrices to joint quaternions is listed in appendix B. The code makes no assumptions about alignment but for the best performance the list with matrices and the list with joints should be at least 16 byte aligned.

5. 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**
ConvertJointQuatsToJointMats (C)	55528	54	63279	62
ConvertJointQuatsToJointMats (SSE)	30916	30	34362	34
ConvertJointMatsToJointQuats (C)	176332	172	176553	173
ConvertJointMatsToJointQuats (SSE)	62460	61	73710	72

6. Conclusion

Two optimized conversions were presented, from joint quaternion to joint matrix and from joint matrix to joint quaternion. Each of the conversions uses a different approach to SIMD. The optimized conversion from joint quaternion to joint matrix uses a compressed calculation. The optimized conversion from joint matrix to joint quaternion exploits parallelism through increased throughput.

For both conversions the SIMD optimized routines were first prototyped using regular C/C++. Rewriting the C/C++ code often helps to analyze the algorithm and to decide upon the best approach to exploiting parallelism with SIMD code.

Optimizing the conversions turned out to be not quite trivial but after giving it some thought the results are quite satisfying. The SSE optimized conversion from joint quaternion to joint matrix consumes over 44% less clock cycles than the optimized C version. The SSE optimized conversion from joint matrix to joint quaternion is more than two times faster than the optimized C version.

7. References

1. On quaternions; or on a new system of imaginaries in algebra.
   Sir William Rowan Hamilton
   Philosophical Magazine xxv, pp. 10-13, July 1844
   The Collected Mathematical Papers, Vol. 3, pp. 355-362, Cambridge University Press, 1967

2. On certain results relating to quaternions.
   Arthur Cayley
   Philosophical Magazine xxvi, pp. 141-145, February 1845
   The collected mathematical papers of Arthur Cayley, Vol. 1, pp. 123-126, Cambridge University Press, 1889
   Available Online: http://name.umdl.umich.edu/ABS3153

3. Complexity of Quaternion Multiplication
Thomas D. Howell, Jean-Claude Lafon
Department of Computer Science, Cornell University, Ithaca, N.Y., TR-75-245, June 1975

4. Application of Quaternions
Gernot Hoffmann
January 20, 2002
Original report "Anleiting zum praktischen Gebrauch von Quaternionen", February 1978
Available Online: http://www.fho-emden.de/~hoffmann

5. Application of Quaternions to Computation with Rotations
Eugene Slamin
Working Paper, Stanford AI Lab, 1979

6. Animating rotation with quaternion curves.
Ken Shoemake
Computer Graphics 19(3):245-254, 1985
Available Online: http://portal.acm.org/citation.cfm?doid=325334.325242

7. Quaternion calculus and fast animation.
Ken Shoemake
SIGGRAPH Course Notes, 10:101-121, 1987

8. Quaternions
Ken Shoemake
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1994
Available Online: ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/

9. Quaternion Calculus for Modeling Rotations in 3D Space
Hartmut Liefke
Department of Computer and Information Science, University of Pennsylvania, April 1998

10. Quaternions, Interpolation and Animation
Erik B. Dam, Martin Koch, Martin Lillholm
Department of Computer Science, University of Copenhagen, Denmark, July 1998
Technical Report DIKU-TR-98/5

11. Quaternion Algebra and Calculus
David Eberly
Magic Software, 2001

Available Online: http://www.magic-software.com

12. Rotation Representations and Performance Issues
    David Eberly
    Magic Software, 2002
    Available Online: http://www.magic-software.com

13. A Linear Algebraic Approach to Quaternions
    David Eberly
    Magic Software, September 16, 2002
    Available Online: http://www.magic-software.com

14. Computing the Inverse Square Root
    Ken Turkowski
    Graphics Gems V
    Morgan Kaufmann Publishers, 1st edition, January 15 1995
    ISBN: 0125434553

15. Fast Inverse Square Root
    David Eberly
    Magic Software, Inc. January 26, 2002
    Available Online: http://www.magic-software.com

16. Fast Inverse Square Root
    Chris Lomont
    Department of Mathematics, Purdue University, Indiana, February 2003
    Available Online: http://www.math.purdue.edu/~clomont

17. Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method
    Intel
    Application Note 803, order nr. 243637-002 version 2.1, January 1999
    Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/resources/appnotes/19061.htm

Appendix A

```
/*
    SSE Optimized Quaternion to Matrix Conversion
    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_INIT1( X, I )                ALIGN16( static X[4] = { I, I, I, I } )
#define R_SHUFFLE_PS( x, y, z, w )          (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | ( (y) & 3 ) << 2 | ( (x) & 3 ))

#define IEEE_SP_ZERO                        0
#define IEEE_SP_SIGN                        ((unsigned long) ( 1 << 31 ))

ALIGN4_INIT4( unsigned long SIMD_SP_quat2mat_x0, IEEE_SP_ZERO, IEEE_SP_SIGN, IEEE_SP_SIGN, IEEE_SP_SIGN );
ALIGN4_INIT4( unsigned long SIMD_SP_quat2mat_x1, IEEE_SP_SIGN, IEEE_SP_ZERO, IEEE_SP_SIGN, IEEE_SP_SIGN );
ALIGN4_INIT4( unsigned long SIMD_SP_quat2mat_x2, IEEE_SP_ZERO, IEEE_SP_SIGN, IEEE_SP_SIGN, IEEE_SP_SIGN );

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

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

struct JointQuat {
    Quaternion  q;
    Vec4        t;
};

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

#define JOINTQUAT_SIZE          (8*4)
#define JOINTQUAT_SIZE_SHIFT    (5)
#define JOINTQUAT_Q_OFFSET      (0*4)
#define JOINTQUAT_T_OFFSET      (4*4)
#define JOINTMAT_SIZE           (4*3*4)

void ConvertJointQuatsToJointMats( JointMat *jointMats, const JointQuat *jointQuats, const int numJoints ) {

    assert_16_byte_aligned( jointMats );
    assert_16_byte_aligned( jointQuats );

    __asm {
        mov         eax, numJoints
        shl         eax, JOINTQUAT_SIZE_SHIFT
        mov         esi, jointQuats
        mov         edi, jointMats

        add         esi, eax
        neg         eax
        jz          done

    loopQuat:
        movaps      xmm0, [esi+eax+JOINTQUAT_Q_OFFSET]      // xmm0 =  q.x,  q.y,  q.z,  q.w
        movaps      xmm6, [esi+eax+JOINTQUAT_T_OFFSET]      // xmm6 =  t.x,  t.y,  t.z,  t.w

        add         edi, JOINTMAT_SIZE

        movaps      xmm1, xmm0                              // xmm1 =    x,    y,    z,    w
        addps       xmm1, xmm1                              // xmm1 =   x2,   y2,   z2,   w2

        add         eax, JOINTQUAT_SIZE

        // calculate the 9 products
        pshufd      xmm2, xmm0, R_SHUFFLE_D( 1, 0, 0, 1 )   // xmm2 =    y,    x,    x,    y
        pshufd      xmm3, xmm1, R_SHUFFLE_D( 1, 1, 2, 2 )   // xmm3 =   y2,   y2,   z2,   z2
        mulps       xmm2, xmm3                              // xmm2 =  yy2,  xy2,  xz2,  yz2

        pshufd      xmm4, xmm0, R_SHUFFLE_D( 2, 3, 3, 3 )   // xmm4 =    z,    w,    w,    w
        pshufd      xmm5, xmm1, R_SHUFFLE_D( 2, 2, 1, 0 )   // xmm5 =   z2,   z2,   y2,   x2
        mulps       xmm4, xmm5                              // xmm4 =  zz2,  wz2,  wy2,  wx2

        mulss       xmm0, xmm1                              // xmm0 =  xx2,   y2,   z2,   w2

        // calculate the last two elements of the third row
        movss       xmm7, SIMD_SP_one                       // xmm7 =          1,           0,           0,       0
        subss       xmm7, xmm0                              // xmm7 =     -xx2+1,           0,           0,       0
        subss       xmm7, xmm2                              // xmm7 = -xx2-yy2+1,           0,           0,       0
        shufps      xmm7, xmm6, R_SHUFFLE_PS( 0, 1, 2, 3 )  // xmm7 = -xx2-yy2+1,           0,         t.z,     t.w

        // calcluate first row
        xorps       xmm2, SIMD_SP_quat2mat_x0               // xmm2 =        yy2,        -xy2,        -xz2,    -yz2
        xorps       xmm4, SIMD_SP_quat2mat_x1               // xmm4 =       -zz2,         wz2,        -wy2,    -wx2
        addss       xmm4, SIMD_SP_one                       // xmm4 =     -zz2+1,         wz2,        -wy2,    -wx2
        movaps      xmm3, xmm4                              // xmm3 =     -zz2+1,         wz2,        -wy2,    -wx2
        subps       xmm3, xmm2                              // xmm3 = -yy2-zz2+1,     xy2+wz2,     xz2-wy2, yz2-wx2
        movaps      [edi-JOINTMAT_SIZE+0*16+0*4], xmm3      // row0 = -yy2-zz2+1,     xy2+wz2,     xz2-wy2, yz2-wx2
        movss       [edi-JOINTMAT_SIZE+0*16+3*4], xmm6      // row0 = -yy2-zz2+1,     xy2+wz2,     xz2-wy2,     t.x

        // calculate second row
        movss       xmm2, xmm0                              // xmm2 =        xx2,        -xy2,        -xz2,    -yz2
        xorps       xmm4, SIMD_SP_quat2mat_x2               // xmm4 =     -zz2+1,        -wz2,         wy2,     wx2
        subps       xmm4, xmm2                              // xmm4 = -xx2-zz2+1,     xy2-wz2,     xz2+wy2, yz2+wx2
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 1, 2, 3, 0 )  // xmm6 =        t.y,         t.z,         t.w,     t.x
        shufps      xmm4, xmm4, R_SHUFFLE_PS( 1, 0, 3, 2 )  // xmm4 =    xy2-wz2, -xx2-zz2+1,      yz2+wx2, xz2+wy2
        movaps      [edi-JOINTMAT_SIZE+1*16+0*4], xmm4      // row1 =    xy2-wz2, -xx2-zz2+1,      yz2+wx2, xz2+wy2
        movss       [edi-JOINTMAT_SIZE+1*16+3*4], xmm6      // row1 =    xy2-wz2, -xx2-zz2+1,      yz2+wx2,     t.y

        // calculate third row
        movhlps     xmm3, xmm4                              // xmm3 =    yz2+wx2,     xz2+wy2,     xz2-wy2, yz2-wx2
        shufps      xmm3, xmm7, R_SHUFFLE_PS( 1, 3, 0, 2 )  // xmm3 =    xz2+wy2,     yz2-wx2, -xx2-yy2+1,      t.z
        movaps      [edi-JOINTMAT_SIZE+2*16+0*4], xmm3      // row2 =    xz2+wy2,     yz2-wx2, -xx2-yy2+1,      t.z

        jl          loopQuat

    done:
    }
}
```

Appendix B

```
/*
    SSE Optimized Matrix to Quaternion Conversion
    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.
*/

ALIGN4_INIT1( unsigned long SIMD_SP_signBit, IEEE_SP_SIGN );
ALIGN4_INIT1( unsigned long SIMD_SP_not, 0xFFFFFFFF );

ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle0, (3<<0)|(2<<8)|(1<<16)|(0<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle1, (0<<0)|(1<<8)|(2<<16)|(3<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle2, (1<<0)|(0<<8)|(3<<16)|(2<<24) );
ALIGN4_INIT1( unsigned long SIMD_DW_mat2quatShuffle3, (2<<0)|(3<<8)|(0<<16)|(1<<24) );

ALIGN4_INIT1( float SIMD_SP_zero, 0.0f );
ALIGN4_INIT1( float SIMD_SP_one, 1.0f );

ALIGN4_INIT1( float SIMD_SP_mat2quat_rsqrt_c1, -0.5f * 0.5f );

ALIGN4_INIT1( float SIMD_SP_rsqrt_c0,  3.0f );
ALIGN4_INIT1( float SIMD_SP_rsqrt_c1, -0.5f );
```

```
void ConvertJointMatsToJointQuats( JointQuat *jointQuats, const JointMat *jointMats, const int numJoints ) {

    ALIGN16( byte shuffle[16]; )

    __asm {
        mov         eax, numJoints
        mov         esi, jointMats
        mov         edi, jointQuats
        and         eax, ~3
        jz          done4
        imul        eax, JOINTMAT_SIZE
        add         esi, eax
        neg         eax

    loopMat4:
        movss       xmm5, [esi+eax+3*JOINTMAT_SIZE+0*16+0*4]
        movss       xmm6, [esi+eax+3*JOINTMAT_SIZE+1*16+1*4]
        movss       xmm7, [esi+eax+3*JOINTMAT_SIZE+2*16+2*4]

        shufps      xmm5, xmm5, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 3, 0, 1, 2 )

        movss       xmm0, [esi+eax+2*JOINTMAT_SIZE+0*16+0*4]
        movss       xmm1, [esi+eax+2*JOINTMAT_SIZE+1*16+1*4]
        movss       xmm2, [esi+eax+2*JOINTMAT_SIZE+2*16+2*4]

        movss       xmm5, xmm0
        movss       xmm6, xmm1
        movss       xmm7, xmm2

        shufps      xmm5, xmm5, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 3, 0, 1, 2 )

        movss       xmm0, [esi+eax+1*JOINTMAT_SIZE+0*16+0*4]
        movss       xmm1, [esi+eax+1*JOINTMAT_SIZE+1*16+1*4]
        movss       xmm2, [esi+eax+1*JOINTMAT_SIZE+2*16+2*4]

        movss       xmm5, xmm0
        movss       xmm6, xmm1
        movss       xmm7, xmm2

        shufps      xmm5, xmm5, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 3, 0, 1, 2 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 3, 0, 1, 2 )

        movss       xmm0, [esi+eax+0*JOINTMAT_SIZE+0*16+0*4]
        movss       xmm1, [esi+eax+0*JOINTMAT_SIZE+1*16+1*4]
        movss       xmm2, [esi+eax+0*JOINTMAT_SIZE+2*16+2*4]

        movss       xmm5, xmm0
        movss       xmm6, xmm1
        movss       xmm7, xmm2

        // -------------------

        movaps      xmm0, xmm5
        addps       xmm0, xmm6
        addps       xmm0, xmm7
        cmpnltps    xmm0, SIMD_SP_zero                      // xmm0 = m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] > 0.0f

        movaps      xmm1, xmm5
        movaps      xmm2, xmm5
        cmpnltps    xmm1, xmm6
        cmpnltps    xmm2, xmm7
        andps       xmm2, xmm1                              // xmm2 = m[0 * 4 + 0] > m[1 * 4 + 1] && m[0 * 4 + 0] > m[2
* 4 + 2]

        movaps      xmm4, xmm6
        cmpnltps    xmm4, xmm7                              // xmm4 = m[1 * 4 + 1] > m[2 * 4 + 2]

        movaps      xmm1, xmm0
        andnps      xmm1, xmm2
        orps        xmm2, xmm0
        movaps      xmm3, xmm2
        andnps      xmm2, xmm4
        orps        xmm3, xmm2
        xorps       xmm3, SIMD_SP_not

        andps       xmm0, SIMD_DW_mat2quatShuffle0
        movaps      xmm4, xmm1
        andps       xmm4, SIMD_DW_mat2quatShuffle1
        orps        xmm0, xmm4
        movaps      xmm4, xmm2
        andps       xmm4, SIMD_DW_mat2quatShuffle2
        orps        xmm0, xmm4
        movaps      xmm4, xmm3
        andps       xmm4, SIMD_DW_mat2quatShuffle3
        orps        xmm4, xmm0

        movaps      shuffle, xmm4

        movaps      xmm0, xmm2
        orps        xmm0, xmm3                              // xmm0 = xmm2 | xmm3   = s0
        orps        xmm2, xmm1                              // xmm2 = xmm1 | xmm2   = s2
        orps        xmm1, xmm3                              // xmm1 = xmm1 | xmm3   = s1

        andps       xmm0, SIMD_SP_signBit
        andps       xmm1, SIMD_SP_signBit
        andps       xmm2, SIMD_SP_signBit

        xorps       xmm5, xmm0
        xorps       xmm6, xmm1
        xorps       xmm7, xmm2
        addps       xmm5, xmm6
        addps       xmm7, SIMD_SP_one
        addps       xmm5, xmm7                              // xmm5 = t

        movaps      xmm7, xmm5                              // xmm7 = t
        rsqrtps     xmm6, xmm5
        mulps       xmm5, xmm6
        mulps       xmm5, xmm6
        subps       xmm5, SIMD_SP_rsqrt_c0
        mulps       xmm6, SIMD_SP_mat2quat_rsqrt_c1
        mulps       xmm6, xmm5                              // xmm6 = s

        mulps       xmm7, xmm6                              // xmm7 = s * t
        xorps       xmm6, SIMD_SP_signBit                   // xmm6 = -s

        // -------------------

        add         edi, 4*JOINTQUAT_SIZE

        movzx       ecx, byte ptr shuffle[0*4+0]            // ecx = k0
        movss       [edi+ecx*4-4*JOINTQUAT_SIZE], xmm7      // q[k0] = s * t;

        movzx       edx, byte ptr shuffle[0*4+1]            // edx = k1
        movss       xmm4, [esi+eax+0*JOINTMAT_SIZE+1*16+0*4]
        xorps       xmm4, xmm2
        subss       xmm4, [esi+eax+0*JOINTMAT_SIZE+0*16+1*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-4*JOINTQUAT_SIZE], xmm4      // q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;

        movzx       ecx, byte ptr shuffle[0*4+2]            // ecx = k2
        movss       xmm3, [esi+eax+0*JOINTMAT_SIZE+0*16+2*4]
        xorps       xmm3, xmm1
        subss       xmm3, [esi+eax+0*JOINTMAT_SIZE+2*16+0*4]
        mulss       xmm3, xmm6
        movss       [edi+ecx*4-4*JOINTQUAT_SIZE], xmm3      // q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;

        movzx       edx, byte ptr shuffle[0*4+3]            // edx = k3
        movss       xmm4, [esi+eax+0*JOINTMAT_SIZE+2*16+1*4]
        xorps       xmm4, xmm0
        subss       xmm4, [esi+eax+0*JOINTMAT_SIZE+1*16+2*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-4*JOINTQUAT_SIZE], xmm4      // q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        mov         ecx, [esi+eax+0*JOINTMAT_SIZE+0*16+3*4]
        mov         [edi-4*JOINTQUAT_SIZE+16], ecx          // q[4] = m[0 * 4 + 3];
        mov         edx, [esi+eax+0*JOINTMAT_SIZE+1*16+3*4]
        mov         [edi-4*JOINTQUAT_SIZE+20], edx          // q[5] = m[1 * 4 + 3];
        mov         ecx, [esi+eax+0*JOINTMAT_SIZE+2*16+3*4]
        mov         [edi-4*JOINTQUAT_SIZE+24], ecx          // q[6] = m[2 * 4 + 3];
        mov         dword ptr [edi-4*JOINTQUAT_SIZE+28], 0  // q[7] = 0.0f;

        shufps      xmm6, xmm6, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm1, xmm1, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm2, xmm2, R_SHUFFLE_PS( 1, 2, 3, 0 )
        movzx       ecx, byte ptr shuffle[1*4+0]            // ecx = k0
        movss       [edi+ecx*4-3*JOINTQUAT_SIZE], xmm7      // q[k0] = s * t;

        movzx       edx, byte ptr shuffle[1*4+1]            // edx = k1
        movss       xmm4, [esi+eax+1*JOINTMAT_SIZE+1*16+0*4]
        xorps       xmm4, xmm2
        subss       xmm4, [esi+eax+1*JOINTMAT_SIZE+0*16+1*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-3*JOINTQUAT_SIZE], xmm4      // q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;

        movzx       ecx, byte ptr shuffle[1*4+2]            // ecx = k2
        movss       xmm3, [esi+eax+1*JOINTMAT_SIZE+0*16+2*4]
        xorps       xmm3, xmm1
        subss       xmm3, [esi+eax+1*JOINTMAT_SIZE+2*16+0*4]
        mulss       xmm3, xmm6
        movss       [edi+ecx*4-3*JOINTQUAT_SIZE], xmm3      // q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;

        movzx       edx, byte ptr shuffle[1*4+3]            // edx = k3
        movss       xmm4, [esi+eax+1*JOINTMAT_SIZE+2*16+1*4]
        xorps       xmm4, xmm0
        subss       xmm4, [esi+eax+1*JOINTMAT_SIZE+1*16+2*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-3*JOINTQUAT_SIZE], xmm4      // q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        mov         ecx, [esi+eax+1*JOINTMAT_SIZE+0*16+3*4]
        mov         [edi-3*JOINTQUAT_SIZE+16], ecx          // q[4] = m[0 * 4 + 3];
        mov         edx, [esi+eax+1*JOINTMAT_SIZE+1*16+3*4]
        mov         [edi-3*JOINTQUAT_SIZE+20], edx          // q[5] = m[1 * 4 + 3];
        mov         ecx, [esi+eax+1*JOINTMAT_SIZE+2*16+3*4]
        mov         [edi-3*JOINTQUAT_SIZE+24], ecx          // q[6] = m[2 * 4 + 3];
        mov         dword ptr [edi-3*JOINTQUAT_SIZE+28], 0  // q[7] = 0.0f;

        shufps      xmm6, xmm6, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm1, xmm1, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm2, xmm2, R_SHUFFLE_PS( 1, 2, 3, 0 )

        movzx       ecx, byte ptr shuffle[2*4+0]            // ecx = k0
        movss       [edi+ecx*4-2*JOINTQUAT_SIZE], xmm7      // q[k0] = s * t;

        movzx       edx, byte ptr shuffle[2*4+1]            // edx = k1
        movss       xmm4, [esi+eax+2*JOINTMAT_SIZE+1*16+0*4]
        xorps       xmm4, xmm2
        subss       xmm4, [esi+eax+2*JOINTMAT_SIZE+0*16+1*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-2*JOINTQUAT_SIZE], xmm4      // q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;

        movzx       ecx, byte ptr shuffle[2*4+2]            // ecx = k2
        movss       xmm3, [esi+eax+2*JOINTMAT_SIZE+0*16+2*4]
        xorps       xmm3, xmm1
        subss       xmm3, [esi+eax+2*JOINTMAT_SIZE+2*16+0*4]
        mulss       xmm3, xmm6
        movss       [edi+ecx*4-2*JOINTQUAT_SIZE], xmm3      // q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;

        movzx       edx, byte ptr shuffle[2*4+3]            // edx = k3
        movss       xmm4, [esi+eax+2*JOINTMAT_SIZE+2*16+1*4]
        xorps       xmm4, xmm0
        subss       xmm4, [esi+eax+2*JOINTMAT_SIZE+1*16+2*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-2*JOINTQUAT_SIZE], xmm4      // q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        mov         ecx, [esi+eax+2*JOINTMAT_SIZE+0*16+3*4]
        mov         [edi-2*JOINTQUAT_SIZE+16], ecx          // q[4] = m[0 * 4 + 3];
        mov         edx, [esi+eax+2*JOINTMAT_SIZE+1*16+3*4]
        mov         [edi-2*JOINTQUAT_SIZE+20], edx          // q[5] = m[1 * 4 + 3];
        mov         ecx, [esi+eax+2*JOINTMAT_SIZE+2*16+3*4]
        mov         [edi-2*JOINTQUAT_SIZE+24], ecx          // q[6] = m[2 * 4 + 3];
        mov         dword ptr [edi-2*JOINTQUAT_SIZE+28], 0  // q[7] = 0.0f;

        shufps      xmm6, xmm6, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm1, xmm1, R_SHUFFLE_PS( 1, 2, 3, 0 )
        shufps      xmm2, xmm2, R_SHUFFLE_PS( 1, 2, 3, 0 )

        movzx       ecx, byte ptr shuffle[3*4+0]            // ecx = k0
        movss       [edi+ecx*4-1*JOINTQUAT_SIZE], xmm7      // q[k0] = s * t;

        movzx       edx, byte ptr shuffle[3*4+1]            // edx = k1
        movss       xmm4, [esi+eax+3*JOINTMAT_SIZE+1*16+0*4]
        xorps       xmm4, xmm2
        subss       xmm4, [esi+eax+3*JOINTMAT_SIZE+0*16+1*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-1*JOINTQUAT_SIZE], xmm4      // q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;

        movzx       ecx, byte ptr shuffle[3*4+2]            // ecx = k2
        movss       xmm3, [esi+eax+3*JOINTMAT_SIZE+0*16+2*4]
        xorps       xmm3, xmm1
        subss       xmm3, [esi+eax+3*JOINTMAT_SIZE+2*16+0*4]
        mulss       xmm3, xmm6
        movss       [edi+ecx*4-1*JOINTQUAT_SIZE], xmm3      // q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;

        movzx       edx, byte ptr shuffle[3*4+3]            // edx = k3
        movss       xmm4, [esi+eax+3*JOINTMAT_SIZE+2*16+1*4]
        xorps       xmm4, xmm0
        subss       xmm4, [esi+eax+3*JOINTMAT_SIZE+1*16+2*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-1*JOINTQUAT_SIZE], xmm4      // q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        mov         ecx, [esi+eax+3*JOINTMAT_SIZE+0*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+16], ecx          // q[4] = m[0 * 4 + 3];
        mov         edx, [esi+eax+3*JOINTMAT_SIZE+1*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+20], edx          // q[5] = m[1 * 4 + 3];
        mov         ecx, [esi+eax+3*JOINTMAT_SIZE+2*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+24], ecx          // q[6] = m[2 * 4 + 3];
        mov         dword ptr [edi-1*JOINTQUAT_SIZE+28], 0  // q[7] = 0.0f;

        add         eax, 4*JOINTMAT_SIZE
        jl          loopMat4

    done4:
        mov         eax, numJoints
        and         eax, 3
        jz          done1
        imul        eax, JOINTMAT_SIZE
        add         esi, eax
        neg         eax

    loopMat1:
        movss       xmm5, [esi+eax+0*JOINTMAT_SIZE+0*16+0*4]
        movss       xmm6, [esi+eax+0*JOINTMAT_SIZE+1*16+1*4]
        movss       xmm7, [esi+eax+0*JOINTMAT_SIZE+2*16+2*4]

        // -------------------

        movaps      xmm0, xmm5
        addss       xmm0, xmm6
        addss       xmm0, xmm7
        cmpnltss    xmm0, SIMD_SP_zero                      // xmm0 = m[0 * 4 + 0] + m[1 * 4 + 1] + m[2 * 4 + 2] > 0.0f

        movaps      xmm1, xmm5
        movaps      xmm2, xmm5
        cmpnltss    xmm1, xmm6
        cmpnltss    xmm2, xmm7
        andps       xmm2, xmm1                              // xmm2 = m[0 * 4 + 0] > m[1 * 4 + 1] && m[0 * 4 + 0] > m[2
* 4 + 2]

        movaps      xmm4, xmm6
        cmpnltss    xmm4, xmm7                              // xmm3 = m[1 * 4 + 1] > m[2 * 4 + 2]

        movaps      xmm1, xmm0
        andnps      xmm1, xmm2
        orps        xmm2, xmm0
        movaps      xmm3, xmm2
        andnps      xmm2, xmm4
        orps        xmm3, xmm2
        xorps       xmm3, SIMD_SP_not

        andps       xmm0, SIMD_DW_mat2quatShuffle0
        movaps      xmm4, xmm1
        andps       xmm4, SIMD_DW_mat2quatShuffle1
        orps        xmm0, xmm4
        movaps      xmm4, xmm2
        andps       xmm4, SIMD_DW_mat2quatShuffle2
        orps        xmm0, xmm4
        movaps      xmm4, xmm3
        andps       xmm4, SIMD_DW_mat2quatShuffle3
        orps        xmm4, xmm0

        movss       shuffle, xmm4
        movaps      xmm0, xmm2
        orps        xmm0, xmm3                              // xmm0 = xmm2 | xmm3   = s0
        orps        xmm2, xmm1                              // xmm2 = xmm1 | xmm2   = s2
        orps        xmm1, xmm3                              // xmm1 = xmm1 | xmm3   = s1

        andps       xmm0, SIMD_SP_signBit
        andps       xmm1, SIMD_SP_signBit
        andps       xmm2, SIMD_SP_signBit

        xorps       xmm5, xmm0
        xorps       xmm6, xmm1
        xorps       xmm7, xmm2
        addss       xmm5, xmm6
        addss       xmm7, SIMD_SP_one
        addss       xmm5, xmm7                              // xmm5 = t

        movss       xmm7, xmm5                              // xmm7 = t
        rsqrtss     xmm6, xmm5
        mulss       xmm5, xmm6
        mulss       xmm5, xmm6
        subss       xmm5, SIMD_SP_rsqrt_c0
        mulss       xmm6, SIMD_SP_mat2quat_rsqrt_c1
        mulss       xmm6, xmm5                              // xmm5 = s

        mulss       xmm7, xmm6                              // xmm7 = s * t
        xorps       xmm6, SIMD_SP_signBit                   // xmm6 = -s

        // -------------------

        movzx       ecx, byte ptr shuffle[0]                // ecx = k0
        add         edi, JOINTQUAT_SIZE
        movss       [edi+ecx*4-1*JOINTQUAT_SIZE], xmm7      // q[k0] = s * t;

        movzx       edx, byte ptr shuffle[1]                // edx = k1
        movss       xmm4, [esi+eax+0*JOINTMAT_SIZE+1*16+0*4]
        xorps       xmm4, xmm2
        subss       xmm4, [esi+eax+0*JOINTMAT_SIZE+0*16+1*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-1*JOINTQUAT_SIZE], xmm4      // q[k1] = ( m[0 * 4 + 1] - s2 * m[1 * 4 + 0] ) * s;

        movzx       ecx, byte ptr shuffle[2]                // ecx = k2
        movss       xmm3, [esi+eax+0*JOINTMAT_SIZE+0*16+2*4]
        xorps       xmm3, xmm1
        subss       xmm3, [esi+eax+0*JOINTMAT_SIZE+2*16+0*4]
        mulss       xmm3, xmm6
        movss       [edi+ecx*4-1*JOINTQUAT_SIZE], xmm3      // q[k2] = ( m[2 * 4 + 0] - s1 * m[0 * 4 + 2] ) * s;

        movzx       edx, byte ptr shuffle[3]                // edx = k3
        movss       xmm4, [esi+eax+0*JOINTMAT_SIZE+2*16+1*4]
        xorps       xmm4, xmm0
        subss       xmm4, [esi+eax+0*JOINTMAT_SIZE+1*16+2*4]
        mulss       xmm4, xmm6
        movss       [edi+edx*4-1*JOINTQUAT_SIZE], xmm4      // q[k3] = ( m[1 * 4 + 2] - s0 * m[2 * 4 + 1] ) * s;

        mov         ecx, [esi+eax+0*JOINTMAT_SIZE+0*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+16], ecx          // q[4] = m[0 * 4 + 3];
        mov         edx, [esi+eax+0*JOINTMAT_SIZE+1*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+20], edx          // q[5] = m[1 * 4 + 3];
        mov         ecx, [esi+eax+0*JOINTMAT_SIZE+2*16+3*4]
        mov         [edi-1*JOINTQUAT_SIZE+24], ecx          // q[6] = m[2 * 4 + 3];
        mov         dword ptr [edi-1*JOINTQUAT_SIZE+28], 0  // q[7] = 0.0f;

        add         eax, JOINTMAT_SIZE
        jl          loopMat1

    done1:
    }
}
```