Slerping Clock Cycles

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

Abstract

An optimized spherical linear interpolation (Slerp) between two quaternions is presented. This optimized Slerp transforms the more commonly used trigonometric functions to mathematically equivalent functions that can be replaced with fast and accurate polynomial approximations. Furthermore the Intel Streaming SIMD Extensions are used to further improve the performance. The end result is a Slerp routine which is more than 7 times faster than the commonly used implementation in C. Furthermore the optimized Slerp is very accurate and actually two times faster than linear interpolation with renormalization (Lerp) in C.

1. Introduction

Quaternions can describe any rotation about any axis in 3D space and, unlike Euler angles, quaternions do not present issues like "gimbal lock". With Euler angles there are orientations in which there may not exist a simple change to the angles to represent a certain local rotation. Quaternions are small and efficient and are often a good replacement for rotation matrices. They take up less space, only 4 scalars as opposed to 9 for a 3x3 rotation matrix. Quaternion multiplication is also more efficient than matrix multiplication and a quaternion can be easily and quickly converted to a rotation matrix where necessary. These properties make quaternions ideal for many algorithms and systems like for instance an animation system. Such an animation system often uses interpolation between quaternions to generate rotations in between key frames. Different animations can also be blended together to achieve smooth transitions from one animation to another. As it turns out interpolation between two general rotations is not trivial and can be computationally expensive. However, quaternions are generally the best representation for interpolating orientations and there are several different approaches with different properties and different computational costs.

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 to avoid common problems such as "gimbal lock" [6]. Quaternions have since found their way into many different systems among which animation, inverse kinematics and physics.

Quaternions are often used for the interpolation between general rotations. Usually three properties are desired when interpolating rotations: minimal torque, constant velocity, commutativity. There are three general approaches to quaternion interpolation, and each of these approaches gives two of the three desirable properties. First there is the spherical linear interpolation also known as Slerp which was popularized by Ken Shoemake [6]. Slerp has both constant velocity and minimal torque but is not commutative. Furthermore there is the linear interpolation with renormalization also known as Lerp which is also discussed by Ken Shoemake [6]. Lerp is commutative and has minimal torque but does not maintain a constant velocity. Finally there is the log-quaternion lerp, also known as exponential map interpolation as described by Grassia [11]. The exponential map interpolation is commutative and maintains a constant velocity but is not torque minimal.

The spherical linear interpolation (Slerp) of quaternions is often considered the optimal interpolation curve between two general rotations. The evaluation of the Slerp function involves several trigonometric functions and is computationally expensive. Several attempts to optimize the Slerp function with mixed results can be found in literature [27,28,29,30].

1.2 Layout

Section 2 shows some properties of quaternions. Section 3 describes spherical linear interpolation between two quaternions. Linear interpolation with renormalization is presented in section 4. In section 5 spherical linear interpolation between two quaternions is optimized. Section 6 goes into the details of implementing SIMD optimized code for spherical linear interpolation. The results of the various optimizations are presented in section 7 and several conclusions are drawn in section 8.

2. Quaternions

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. This interpretation shows that the quaternion (x, y, z, w) describes the same general rotation as the quaternion (-x, -y, -z, -w) because a rotation defined by an axis and an angle is equivalent to the rotation defined by the opposite axis and negated angle. From sin2(α) + cos2(α) = 1 and the fact that the axis of rotation is unit length follows that the following holds for quaternions that represent general rotations: x2 + y2 + z2 + w2 = 1, which describes the unit quaternion sphere.

A quaternion is small and efficient and can easily be converted to a rotation matrix. Therefore quaternions are often used in skeletal animation systems to describe the orientation of joints. Such an animation system often uses key frames described by quaternions and positions and requires interpolation between key frames to display smooth motion.

Quaternions can be used for the interpolation between general rotations by using four-dimensional vector interpolation. Given two quaternions *q0* and *q1* and a parameter *t* in the range [0, 1] the general formula for the interpolation between *q0* and *q1* is given by:

$$q(t) = f_0(t) \cdot q0 + f_1(t) \cdot q1$$

where f_0 and f_1 are scalar functions such that $f_0(0) = 1$, $f_0(1) = 0$, $f_1(0) = 0$ and $f_1(1) = 1$. The exact course of the functions f_0 and f_1 may vary based on the desired properties of the interpolation.

3. Slerp

The interpolation curve for spherical linear interpolation forms the shortest great arc on the quaternion unit sphere. Slerp has constant angular velocity and is often considered the optimal interpolation curve between two general rotations.

Given two quaternions *q0* and *q1* and a parameter *t* in the range [0,1] the spherical linear interpolation is defined as follows:

$$q(t) = \frac{\sin(\,(1 - t) \cdot \alpha\,)}{\sin(\,\alpha\,)} \cdot q0 + \frac{\sin(\,t \cdot \alpha\,)}{\sin(\,\alpha\,)} \cdot q1$$

Where α is the angle between q0 and q1 which can be calculated from the dot product of the two quaternions.

$$\cos(\,\alpha\,) = q0 \cdot q1$$

The following code implements the spherical linear interpolation between two quaternions.

```
void Slerp( const Quaternion &from, const Quaternion &to, float t, Quaternion &result ) {
    float cosom, absCosom, sinom, omega, scale0, scale1;

    cosom = from.x * to.x + from.y * to.y + from.z * to.z + from.w * to.w;
    absCosom = fabs( cosom );
    if ( ( 1.0f - absCosom ) > 1e-6f ) {
        omega = acos( absCosom );
        sinom = 1.0f / sin( omega );
        scale0 = sin( ( 1.0f - t ) * omega ) * sinom;
```

```
        scale1 = sin( t * omega ) * sinom;
    } else {
        scale0 = 1.0f - t;
        scale1 = t;
    }
    scale1 = ( cosom >= 0.0f ) ? scale1 : -scale1;
    result.x = scale0 * from.x + scale1 * to.x;
    result.y = scale0 * from.y + scale1 * to.y;
    result.z = scale0 * from.z + scale1 * to.z;
    result.w = scale0 * from.w + scale1 * to.w;
}
```

Although the above routine is fairly small, even in assembler code, the routine may consume a significant number of clock cycles on today's hardware. When used to interpolate between many quaternions, for instance in an animation system, this routine can easily cause performance problems.

4. Lerp

Spherical linear interpolation between two quaternions can be approximated with a linear interpolation with renormalization (Lerp). The interpolation traces out the exact same curve as Slerp, but does not maintain a constant speed across the arc. The speedup in the middle is due to the fact that the interpolation curve takes a short cut below the surface of the unit sphere.

The following code implements the Lerp. No trigonometric functions are used and the code is significantly faster than the Slerp code above.

```
Lerp( const Quaternion &from, const Quaternion &to, float t, Quaternion &result ) {
    float cosom, scale0, scale1, s;

    cosom = from.x * to.x + from.y * to.y + from.z * to.z + from.w * to.w;

    scale0 = 1.0f - t;
    scale1 = ( cosom >= 0.0f ) ? t : -t;

    result.x = scale0 * from.x + scale1 * to.x;
    result.y = scale0 * from.y + scale1 * to.y;
    result.z = scale0 * from.z + scale1 * to.z;
    result.w = scale0 * from.w + scale1 * to.w;

    s = 1.0f / sqrt( result.x * result.x + result.y * result.y + result.z * result.z + result.w * result.w );

    result.x *= s;
    result.y *= s;
    result.z *= s;
    result.w *= s;
}
```

For many purposes the non-constant velocity is not or hardly noticeable. Especially for rotations over small angles the above routine may be a good alternative to the slower Slerp. However, other applications may require the velocity to be a constant function across the arc and spherical linear interpolation may be the preferred method. Fortunately Slerp does not have to be slower than the above routine as shown in the next sections.

5. Optimizing Slerp

What exactly makes Slerp so slow? Slerp uses several trigonometric functions that are not particularly fast on today's hardware. The arc cosine is usually a math library function which evaluates a square root and an arc tangent function. On an Intel Pentium these translate to an 'fsqrt' and an 'fpatan' instruction respectively. Both instructions have high latency and stall the FPU for many clock cycles. Next Slerp calculates the reciprocal of the sine of the angle between the quaternions. On an Intel Pentium this calculation typically uses the 'fsin' instruction with a dependent 'fdiv' instruction. Both these instructions have high latency and throughput. Furthermore the quaternion scale factors are calculated with two more sine functions that also translate to expensive 'fsin' instructions. All together this amounts to many clock cycles spent evaluating trigonometric functions.

Fortunately the following fundamental identities can be used to transform the trigonometric functions into functions that can be evaluated much faster on today's hardware.

$$\sin^2(\alpha) + \cos^2(\alpha) = 1$$

$$\tan(\alpha) = \frac{\sin(\alpha)}{\cos(\alpha)}$$

The cosine of the angle between the two quaternions is known. Using the first identity shown above the sine can be trivially calculated from the cosine as follows.

$$\sin(\alpha) = \sqrt{1 - \cos^2(\alpha)}$$

Because the square of the cosine is used any sign information would be lost in the above calculation. However, Slerp uses the absolute value of the cosine so no special handling is required.

Once both the sine and cosine of the angle are available there really is no need to use an expensive arc cosine function to calculate the actual angle between the quaternions. The second identity shown above can be used to calculate the angle from the sine and the cosine. The following code shows the new Slerp.

```
void SlerpTransformed( const Quaternion &from, const Quaternion &to, float t, Quaternion &result ) {
    float cosom, absCosom, sinom, sinSqr, omega, scale0, scale1;

    cosom = from.x * to.x + from.y * to.y + from.z * to.z + from.w * to.w;
    absCosom = fabs( cosom );
    if ( ( 1.0f - absCosom ) > 1e-6f ) {
        sinSqr = 1.0f - absCosom * absCosom;
        sinom = 1.0f / sqrt( sinSqr );
        omega = atan2( sinSqr * sinom, absCosom );
        scale0 = sin( ( 1.0f - t ) * omega ) * sinom;
        scale1 = sin( t * omega ) * sinom;
    } else {
        scale0 = 1.0f - t;
        scale1 = t;
    }
    scale1 = ( cosom >= 0.0f ) ? scale1 : -scale1;
    result.x = scale0 * from.x + scale1 * to.x;
    result.y = scale0 * from.y + scale1 * to.y;
    result.z = scale0 * from.z + scale1 * to.z;
    result.w = scale0 * from.w + scale1 * to.w;
}
```

Looking at the FPU assembler code for the above routine there are now one 'fsqrt' instruction, one 'fdiv' instruction, one 'fpatan' instruction and two 'fsin' instructions. In the original routine there are one 'fsqrt' instruction, one 'fdiv' instruction, one 'fpatan' instruction and three 'fsin' instructions. In other words the new routine has one 'fsin' instruction less than the original routine.

The 1.0f / sqrt() can be replaced with a slightly faster approximation [16,17,18]. The following approximation does not use the expensive division and also avoids the expensive square root calculation.

```
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;
}
```

When looking at the parameters to the trigonometric functions some key observations can be made. Both parameters to the arc tangent function are always positive and the parameters to the sine functions are always in the range [0, PI/2]. This allows the trigonometric functions to be replaced with fast and accurate polynomial approximations without the need for time consuming range reductions.

When the angle is always in the range [0, PI/2] the sine function can be replaced with a polynomial approximation without the need for any logic [19,20,21,22]. The following function approximates the sine function for angles in the range [0, PI/2]. The maximum absolute error is 2.308×10^{-9}.

```
float SinZeroHalfPI( float a ) {
    float s, t;

    s = a * a;
    t = -2.39e-08f;
    t *= s;
    t += 2.7526e-06f;
    t *= s;
    t += -1.98409e-04f;
    t *= s;
    t += 8.3333315e-03f;
    t *= s;
    t += -1.666666664e-01f;
    t *= s;
    t += 1.0f;
    t *= a;
    return t;
}
```

When both parameters are always positive the arc tangent function can be replaced with a polynomial approximation without the need for range reduction [19,20,21,22]. The following function approximates the arc tangent function with a maximum absolute error of 1.359×10^{-8}.

```
float ATanPositive( float y, float x ) {
    float a, d, s, t;

    if ( y > x ) {
        a = -x / y;
        d = M_PI / 2;
    } else {
        a = y / x;
        d = 0.0f;
    }
    s = a * a;
    t = 0.0028662257f;
    t *= s;
```

```
    t += -0.0161657367f;
    t *= s;
    t += 0.0429096138f;
    t *= s;
    t += -0.0752896400f;
    t *= s;
    t += 0.1065626393f;
    t *= s;
    t += -0.1420889944f;
    t *= s;
    t += 0.1999355085f;
    t *= s;
    t += -0.3333314528f;
    t *= s;
    t += 1.0f;
    t *= a;
    t += d;
    return t;
}
```

The arc cosine in the original routine could also have been approximated directly with a polynomial without using the fundamental identities to transform the trigonometric functions. However, an accurate polynomial approximation of the arc cosine is much more expensive than an accurate polynomial approximation of the arc tangent.

The following code shows the optimized Slerp which uses the polynomial approximations for the trigonometric functions.

```
void SlerpOptimized( const Quaternion &from, const Quaternion &to, float t, Quaternion &result ) {
    float cosom, absCosom, sinom, sinSqr, omega, scale0, scale1;

    cosom = from.x * to.x + from.y * to.y + from.z * to.z + from.w * to.w;
    absCosom = fabs( cosom );
    if ( ( 1.0f - absCosom ) > 1e-6f ) {
        sinSqr = 1.0f - absCosom * absCosom;
        sinom = ReciprocalSqrt( sinSqr );
        omega = ATanPositive( sinSqr * sinom, absCosom );
        scale0 = SinZeroHalfPI( ( 1.0f - t ) * omega ) * sinom;
        scale1 = SinZeroHalfPI( t * omega ) * sinom;
    } else {
        scale0 = 1.0f - t;
        scale1 = t;
    }
    scale1 = ( cosom >= 0.0f ) ? scale1 : -scale1;
    result.x = scale0 * from.x + scale1 * to.x;
    result.y = scale0 * from.y + scale1 * to.y;
    result.z = scale0 * from.z + scale1 * to.z;
    result.w = scale0 * from.w + scale1 * to.w;
}
```

The above optimized Slerp is typically faster on today's hardware, especially if the two sine calculations are inlined and properly interleaved. Slerp can be made even faster on today's SIMD capable architectures as shown in the next section.

6. SSE Optimized Slerp

Most algorithms do not use a single isolated spherical linear interpolation between two quaternions. For instance a skeletal animation system usually requires interpolation between two key frames. Each of the key frames is a list with joints that define a pose of the skeleton. A joint from a key frame is stored as a quaternion for the orientation and a 4D vector for the position. The SSE optimized routine presented here will interpolate between two lists with joints as shown below.

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

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

void Vec4Lerp( const Vec4 &from, const Vec4 &to, const float t, Vec4 &result ) {
    float s = 1.0f - t;
    result.x = from.x * s + to.x * t;
    result.y = from.y * s + to.y * t;
    result.z = from.z * s + to.z * t;
    result.w = from.w * s + to.w * t;
}

void SlerpJoints( JointQuat *joints, const JointQuat *blendJoints, const float lerp, const int *index, const int
numJoints ) {
    int i;

    for ( i = 0; i < numJoints; i++ ) {
        int j = index[i];
        Slerp( joints[j].q, blendJoints[j].q, lerp, joints[j].q );
        Vec4Lerp( joints[j].t, blendJoints[j].t, lerp, joints[j].t );
    }
}
```

An additional index is used in the above code to allow a selection of joints from the two lists to be interpolated instead of the complete lists. This may be useful for an animation system where during certain animations only a subset of all the joints are animated.

The best approach to SIMD is usually to exploit parallelism through increased throughput. The routine presented here will interpolate between four pairs of joints per iteration.

The interpolation of the positions stored with the joints is best calculated individually for each pair of joints. After all this interpolation is no more than the addition of two scaled 4D vectors which trivially maps to SSE instructions.

For the interpolation of the quaternions the scalar instructions are best replaced with functionally equivalent SSE instructions. This requires a swizzle because the quaternions are stored per joint while the individual components of four quaternions need to be grouped into SSE registers. For this swizzle four quaternions are loaded into four SSE registers as a 4x4 matrix. This 4x4 matrix is then transposed in the registers with several unpack and shuffle instructions.

After the swizzle the scalar instructions of the optimized Slerp routine can be replaced with functionally equivalent SSE instructions. This is trivial for the most part but several details need to be worked out.

The 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 [24]. The following assembler code calculates the reciprocal square root of the four floating point numbers stored in the 'xmm0' register. The result is stored in the same register.

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

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

rsqrtps     xmm1, xmm0
mulps       xmm0, xmm1
mulps       xmm0, xmm1
subps       xmm0, SIMD_SP_rsqrt_c0
```

```
mulps       xmm1, SIMD_SP_rsqrt_c1
mulps       xmm0, xmm1
```

The polynomial approximation of the sine function in the range [0, PI/2] is trivially implemented with SSE instructions. The following code calculates four sines for the angles stored in 'xmm0' and the result is stored in 'xmm2'.

```
ALIGN4_INIT1( float SIMD_SP_sin_c0, -2.39e-08f );
ALIGN4_INIT1( float SIMD_SP_sin_c1,  2.7526e-06f );
ALIGN4_INIT1( float SIMD_SP_sin_c2, -1.98409e-04f );
ALIGN4_INIT1( float SIMD_SP_sin_c3,  8.3333315e-03f );
ALIGN4_INIT1( float SIMD_SP_sin_c4, -1.666666664e-01f );
ALIGN4_INIT1( float SIMD_SP_one, 1.0f );

movaps      xmm1, xmm0
mulps       xmm1, xmm1
movaps      xmm2, SIMD_SP_sin_c0
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_sin_c1
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_sin_c2
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_sin_c3
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_sin_c4
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_one
mulps       xmm2, xmm0
```

The logic at the beginning of the arc tangent approximation is a bit trickier to implement with SSE instructions. Basically the smallest of the two parameters is divided by the largest one. If 'y' is larger than 'x' the result of the division is negated and PI/2 is added to the result of the polynomial. The 'minps' and 'maxps' instructions can be used to select the minimum and the maximum of the two parameters. The 'cmpeqps' instruction can then be used to compare 'x' with the minimum of the two parameters. If 'x' is equal to the minimum then the 'cmpeqps' instruction fills the result register with all ones and if 'x' is the maximum then the result register is filled with all zeros. This bit mask can be used to either leave or flip the sign bit of the result of the division, and also to add either zero or PI/2 to the result of the polynomial. The following SSE code implements the logic for the arc tangent approximation.

```
#define IEEE_SP_SIGN    ((unsigned long) ( 1 << 31 ))

ALIGN4_INIT1( float SIMD_SP_halfPI, M_PI/2 );
ALIGN4_INIT1( unsigned long SIMD_SP_signBit, IEEE_SP_SIGN );

movaps      xmm3, xmm0
minps       xmm0, xmm1              // xmm0 = ( y > x ) ? x : y
maxps       xmm1, xmm3              // xmm1 = ( y > x ) ? y : x
cmpeqps     xmm3, xmm0              // xmm3 = ( y > x ) ? 0xFFFFFFFF : 0x00000000
divps       xmm0, xmm1              // xmm0 = ( y > x ) ? x / y : y / x
movaps      xmm1, xmm3
andps       xmm1, SIMD_SP_signBit   // xmm1 = ( y > x ) ? 0x80000000 : 0x00000000
xorps       xmm0, xmm1              // xmm0 = ( y > x ) ? -x / y : y / x
andps       xmm3, SIMD_SP_halfPI    // xmm3 = ( y > x ) ? PI/2 : 0.0f
```

Instead of using the slow 'divps' instruction the 'rcpps' instruction can be used. This instruction calculates the reciprocal of a number with 12 bits of precision. A Newton-Rapson iteration can be used to improve the accuracy of the reciprocal [24].

```
rcpps       xmm2, xmm1
mulps       xmm1, xmm2
mulps       xmm1, xmm2
addps       xmm2, xmm2
subps       xmm2, xmm1      // xmm2 = ( y > x ) ? 1 / y : 1 / x
```

```
mulps       xmm0, xmm2      // xmm0 = ( y > x ) ? x / y : y / x
```

The following SSE code implements the complete arc tangent function with two positive parameters. The parameters 'x' and 'y' are assumed to be stored in 'xmm0' and 'xmm1' respectively. The result is stored in the register 'xmm2'.

```
ALIGN4_INIT1( float SIMD_SP_atan_c0,  0.0028662257f );
ALIGN4_INIT1( float SIMD_SP_atan_c1, -0.0161657367f );
ALIGN4_INIT1( float SIMD_SP_atan_c2,  0.0429096138f );
ALIGN4_INIT1( float SIMD_SP_atan_c3, -0.0752896400f );
ALIGN4_INIT1( float SIMD_SP_atan_c4,  0.1065626393f );
ALIGN4_INIT1( float SIMD_SP_atan_c5, -0.1420889944f );
ALIGN4_INIT1( float SIMD_SP_atan_c6,  0.1999355085f );
ALIGN4_INIT1( float SIMD_SP_atan_c7, -0.3333314528f );

movaps      xmm3, xmm0
minps       xmm0, xmm1              // xmm0 = ( y > x ) ? x : y
maxps       xmm1, xmm3              // xmm1 = ( y > x ) ? y : x
cmpeqps     xmm3, xmm0              // xmm3 = ( y > x ) ? 0xFFFFFFFF : 0x00000000
rcpps       xmm2, xmm1
mulps       xmm1, xmm2
mulps       xmm1, xmm2
addps       xmm2, xmm2
subps       xmm2, xmm1              // xmm2 = ( y > x ) ? 1 / y : 1 / x
mulps       xmm0, xmm2              // xmm0 = ( y > x ) ? x / y : y / x
movaps      xmm1, xmm3
andps       xmm1, SIMD_SP_signBit   // xmm1 = ( y > x ) ? 0x80000000 : 0x00000000
xorps       xmm0, xmm1              // xmm0 = ( y > x ) ? -x / y : y / x
andps       xmm3, SIMD_SP_halfPI    // xmm3 = ( y > x ) ? PI/2 : 0.0f
movaps      xmm1, xmm0
mulps       xmm1, xmm1
movaps      xmm2, SIMD_SP_atan_c0
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c1
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c2
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c3
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c4
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c5
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c6
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_atan_c7
mulps       xmm2, xmm1
addps       xmm2, SIMD_SP_one
mulps       xmm2, xmm0
addps       xmm2, xmm3
```

The complete routine to interpolate between two lists with joints is listed in appendix A. The code in appendix A assumes the lists with joints are 16-byte aligned. Because each joint is 32 bytes in size it is better on a Pentium 4 to make the lists 32 or 64-byte aligned to assure the least number of cache lines are used per iteration.

For comparison an SSE optimized version of linear interpolation with renormalization has been implemented as well. The code for this routine is listed in appendix B.

7. Results

The various routines for interpolation between joints have been tested on an Intel® Pentium® 4 Processor on 130nm Technology and an Intel® Pentium® 4 Processor on 90nm Technology. The routines interpolated joints from two lists with 1024 joints each. 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. Keep in mind that the routines do not just interpolate quaternions but interpolate between joints which involves both a quaternion for the orientation and a 4D vector for the position.

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**
SlerpJoints (C)	1035248	1011	1041893	1018
SlerpJoints (SSE)	109112	107	131517	128
LerpJoints (C)	218996	213	253350	248
LerpJoints (SSE)	51080	50	52848	52

The maximum absolute error of the SSE optimized Slerp compared to the original Slerp is 4.768×10^{-7}. The performance of the optimized Slerp can easily be improved by using lower degree polynomials. However, this would obviously also decrease the accuracy and the routine implemented here favors high accuracy over the additional speed improvement.

8. Conclusion

When optimizing code the fastest algorithm that suits the needs of the application should be chosen first. For some applications linear interpolation with renormalization may be the perfect trade between speed and interpolation properties. Once the right algorithm has been chosen this algorithm should first be optimized on an algorithmic and mathematical level. Only the final step in the optimization process involves exploiting the instruction set of an SIMD capable architecture.

One might argue that the optimized spherical linear interpolation presented here is only faster at the cost of loosing accuracy since it uses various approximations. However, the use of floating point numbers means that most calculations loose precision one way or the other. The optimized Slerp presented here is significantly faster at a minimal loss of accuracy. Redundant and duplicate calculations are avoided and the Intel SSE instructions are used to get the most out of every clock cycle.

This article shows that spherical linear interpolation (Slerp) does not have to be significantly slower than linear interpolation with renormalization (Lerp), especially when the Intel SSE instruction set is used to exploit parallelism. Interestingly the SSE optimized Slerp is two times faster than the C code for the Lerp. In the end the SSE optimized Lerp uses the least number of clock cycles and may still be preferred when a non-constant velocity during the interpolation is not an issue. However, the SSE optimized Slerp is well over 7 times faster than the commonly used implementation in C and makes the interpolation significantly faster when a constant angular velocity is required.

9. 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. Practical parameterization of rotations using the exponential map
F. Sebastian Grassia
Journal of Graphics Tools, volume 3.3, 1998

12. Quaternion Algebra and Calculus
David Eberly
Magic Software, 2001
Available Online: http://www.magic-software.com

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

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

15. Incremental Spherical Linear Interpolation
Tony Barrera, Anders Hast and Ewert Bengtsson
SIGRAD 2004. The Annual SIGRAD Conference. Special Theme - Environmental Visualization. November 24-25, 2004, Gävle, Sweden
Linköping Electronic Conference Proceedings, ISSN 1650-3686 (print), 1650-3740 (www)
Available Online: http://www.ep.liu.se/ecp/013/004/

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

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

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

19. Rational approximations of functions
Bengt Carlson, M Goldstein
Los Alamos Scientific Laboratory of the University of California, 1955

20. Software Manual for the Elementary Functions
Jr. Cody, J. William, William Waite
Prentice Hall, 1980

21. Polynomial Approximations to Trigonometric Functions
Eddie Edwards
Game Programming Gems, 2000
Available Online: http://www.GameProgrammingGems.com

22. More Approximations to Trigonometric Functions
Robin Green
Game Programming Gems 3, 2002
Available Online: http://www.GameProgrammingGems.com

23. High-Speed Function Approximation Using a Minimax Quadratic Interpolator
Jose-Alenjandro Pineiro, Suart F. Oberman, Jean-Michel Muller, Javer D. Bruguera
IEE Transactions on Computers, vol 54, no. 3, March 2005
Available Online: http://perso.ens-lyon.fr/jean-michel.muller/QuadraticIEEETC0305.pdf

24. 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

25. Better 3D: The Writing Is on the Wal

Jeff Lander
Game Developer Magazine, March 1998
Available Online: http://www.darwin3d.com/gdm1998.htm#gdm0398

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

27. Hacking Quaternions
Jonathan Blow
The Inner Product, Game Developer Magazine, March 2002
Available Online: http://number-none.com/product/Hacking%20Quaternions/index.html

28. PolySlerp: a fast and accurate polynomial approximation of spherical linear interpolation (Slerp).
Thomas Busser
Game Developer Magazine, February 2004
Available Online:
http://www.highbeam.com/library/doc0.asp?DOCID=1G1:113526291&num=5

29. Understanding Slerp, Then Not Using It
Jonathan Blow
The Inner Product, Game Developer Magazine, April 2004
Available Online: http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/index.html

30. Faster Quaternion Interpolation Using Approximations
Andy Thomason
Game Programming Gems 5, 2005
Available Online: http://www.GameProgrammingGems.com

Appendix A

```
/*
    SSE Optimized Spherical Linear Interpolation between Quaternions
    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_INIT1( unsigned long SIMD_SP_signBit, IEEE_SP_SIGN );

ALIGN4_INIT1( float SIMD_SP_one, 1.0f );
ALIGN4_INIT1( float SIMD_SP_halfPI, M_PI/2 );

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

ALIGN4_INIT1( float SIMD_SP_sin_c0, -2.39e-08f );
ALIGN4_INIT1( float SIMD_SP_sin_c1,  2.7526e-06f );
ALIGN4_INIT1( float SIMD_SP_sin_c2, -1.98409e-04f );
ALIGN4_INIT1( float SIMD_SP_sin_c3,  8.3333315e-03f );
ALIGN4_INIT1( float SIMD_SP_sin_c4, -1.666666664e-01f );

ALIGN4_INIT1( float SIMD_SP_atan_c0,  0.0028662257f );
ALIGN4_INIT1( float SIMD_SP_atan_c1, -0.0161657367f );
ALIGN4_INIT1( float SIMD_SP_atan_c2,  0.0429096138f );
ALIGN4_INIT1( float SIMD_SP_atan_c3, -0.0752896400f );
ALIGN4_INIT1( float SIMD_SP_atan_c4,  0.1065626393f );
ALIGN4_INIT1( float SIMD_SP_atan_c5, -0.1420889944f );
ALIGN4_INIT1( float SIMD_SP_atan_c6,  0.1999355085f );
ALIGN4_INIT1( float SIMD_SP_atan_c7, -0.3333314528f );

#define TRANSPOSE_4x4( reg0, reg1, reg2, reg3, reg4 )                                       \
    __asm   movaps      reg4, reg2                              /* reg4 =  8,  9, 10, 11 */     \
    __asm   unpcklps    reg2, reg3                              /* reg2 =  8, 12,  9, 13 */     \
    __asm   unpckhps    reg4, reg3                              /* reg4 = 10, 14, 11, 15 */     \
    __asm   movaps      reg3, reg0                              /* reg3 =  0,  1,  2,  3 */     \
    __asm   unpcklps    reg0, reg1                              /* reg0 =  0,  4,  1,  5 */     \
    __asm   unpckhps    reg3, reg1                              /* reg3 =  2,  6,  3,  7 */     \
    __asm   movaps      reg1, reg0                              /* reg1 =  0,  4,  1,  5 */     \
    __asm   shufps      reg0, reg2, R_SHUFFLE_PS( 0, 1, 0, 1 )  /* reg0 =  0,  4,  8, 12 */     \
    __asm   shufps      reg1, reg2, R_SHUFFLE_PS( 2, 3, 2, 3 )  /* reg1 =  1,  5,  9, 13 */     \
    __asm   movaps      reg2, reg3                              /* reg2 =  2,  6,  3,  7 */     \
    __asm   shufps      reg2, reg4, R_SHUFFLE_PS( 0, 1, 0, 1 )  /* reg2 =  2,  6, 10, 14 */     \
    __asm   shufps      reg3, reg4, R_SHUFFLE_PS( 2, 3, 2, 3 )  /* reg3 =  3,  7, 11, 15 */

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

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

struct JointQuat {
    Quaternion  q;
    Vec4        t;
};

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

void SlerpJoints( JointQuat *joints, const JointQuat *blendJoints, const float lerp, const int *index, const int
numJoints ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( blendJoints );
    assert_16_byte_aligned( JOINTQUAT_Q_OFFSET );
    assert_16_byte_aligned( JOINTQUAT_T_OFFSET );

    ALIGN16( float jointQuat0[4]; )
    ALIGN16( float jointQuat1[4]; )
    ALIGN16( float jointQuat2[4]; )
    ALIGN16( float jointQuat3[4]; )
    ALIGN16( float blendQuat0[4]; )
```

```
ALIGN16( float blendQuat1[4]; )
ALIGN16( float blendQuat2[4]; )
ALIGN16( float blendQuat3[4]; )
int a0, a1, a2, a3;

__asm {
    movss       xmm7, lerp
    cmpnless    xmm7, SIMD_SP_zero
    movmskps    ecx, xmm7
    test        ecx, 1
    jz          done1

    mov         eax, numJoints
    shl         eax, 2
    mov         esi, joints
    mov         edi, blendJoints
    mov         edx, index

    add         edx, eax
    neg         eax
    jz          done1

    movss       xmm7, lerp
    cmpnltss    xmm7, SIMD_SP_one
    movmskps    ecx, xmm7
    test        ecx, 1
    jz          lerpJoints

loopCopy:
    mov         ecx, [edx+eax]
    shl         ecx, JOINTQUAT_SIZE_SHIFT

    add         eax, 1*4

    movaps      xmm0, [edi+ecx+JOINTQUAT_Q_OFFSET]
    movaps      xmm1, [edi+ecx+JOINTQUAT_T_OFFSET]
    movaps      [esi+ecx+JOINTQUAT_Q_OFFSET], xmm0
    movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm1

    jl          loopCopy

    jmp         done1

lerpJoints:
    add         eax, 4*4
    jge         done4

loopJoint4:
    movss       xmm3, lerp
    shufps      xmm3, xmm3, R_SHUFFLE_PS( 0, 0, 0, 0 )

    mov         ecx, [edx+eax-4*4]
    shl         ecx, JOINTQUAT_SIZE_SHIFT
    mov         a0, ecx

    // lerp first translations
    movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
    subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
    mulps       xmm7, xmm3
    addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
    movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

    // load first quaternions
    movaps      xmm0, [esi+ecx+JOINTQUAT_Q_OFFSET]
    movaps      xmm4, [edi+ecx+JOINTQUAT_Q_OFFSET]

    mov         ecx, [edx+eax-3*4]
    shl         ecx, JOINTQUAT_SIZE_SHIFT
    mov         a1, ecx

    // lerp second translations
    movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
    subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
    mulps       xmm7, xmm3
    addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
    movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

    // load second quaternions
    movaps      xmm1, [esi+ecx+JOINTQUAT_Q_OFFSET]
    movaps      xmm5, [edi+ecx+JOINTQUAT_Q_OFFSET]
```

```
mov         ecx, [edx+eax-2*4]
shl         ecx, JOINTQUAT_SIZE_SHIFT
mov         a2, ecx

// lerp third translations
movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
mulps       xmm7, xmm3
addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

// load third quaternions
movaps      xmm2, [esi+ecx+JOINTQUAT_Q_OFFSET]
movaps      xmm6, [edi+ecx+JOINTQUAT_Q_OFFSET]

mov         ecx, [edx+eax-1*4]
shl         ecx, JOINTQUAT_SIZE_SHIFT
mov         a3, ecx

// lerp fourth translations
movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
mulps       xmm7, xmm3
addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

// load fourth quaternions
movaps      xmm3, [esi+ecx+JOINTQUAT_Q_OFFSET]

TRANSPOSE_4x4( xmm0, xmm1, xmm2, xmm3, xmm7 )

movaps      jointQuat0, xmm0
movaps      jointQuat1, xmm1
movaps      jointQuat2, xmm2
movaps      jointQuat3, xmm3

movaps      xmm7, [edi+ecx+JOINTQUAT_Q_OFFSET]

TRANSPOSE_4x4( xmm4, xmm5, xmm6, xmm7, xmm3 )

movaps      blendQuat0, xmm4
movaps      blendQuat1, xmm5
movaps      blendQuat2, xmm6
movaps      blendQuat3, xmm7

// lerp quaternions
mulps       xmm0, xmm4
mulps       xmm1, xmm5
addps       xmm0, xmm1
mulps       xmm2, xmm6
addps       xmm0, xmm2
movaps      xmm3, jointQuat3
mulps       xmm3, blendQuat3
addps       xmm0, xmm3                  // xmm0 = cosom

movaps      xmm1, xmm0
movaps      xmm2, xmm0
andps       xmm1, SIMD_SP_signBit       // xmm1 = signBit
xorps       xmm0, xmm1
mulps       xmm2, xmm2

xorps       xmm4, xmm4
movaps      xmm3, SIMD_SP_one
subps       xmm3, xmm2                  // xmm3 = scale0
cmpeqps     xmm4, xmm3
andps       xmm4, SIMD_SP_tiny          // if values are zero replace them with a tiny number
andps       xmm3, SIMD_SP_absMask       // make sure the values are positive
orps        xmm3, xmm4

movaps      xmm2, xmm3
rsqrtps     xmm4, xmm2
mulps       xmm2, xmm4
mulps       xmm2, xmm4
subps       xmm2, SIMD_SP_rsqrt_c0
mulps       xmm4, SIMD_SP_rsqrt_c1
mulps       xmm2, xmm4
mulps       xmm3, xmm2                  // xmm3 = sqrt( scale0 )

// omega0 = atan2( xmm3, xmm0 )
movaps      xmm4, xmm0
minps       xmm0, xmm3
```

```
        maxps       xmm3, xmm4
        cmpeqps     xmm4, xmm0
        rcpps       xmm5, xmm3
        mulps       xmm3, xmm5
        mulps       xmm3, xmm5
        addps       xmm5, xmm5
        subps       xmm5, xmm3                  // xmm5 = 1 / y or 1 / x
        mulps       xmm0, xmm5                  // xmm0 = x / y or y / x
        movaps      xmm3, xmm4
        andps       xmm3, SIMD_SP_signBit
        xorps       xmm0, xmm3                  // xmm0 = -x / y or y / x
        andps       xmm4, SIMD_SP_halfPI        // xmm4 = HALF_PI or 0.0f
        movaps      xmm3, xmm0
        mulps       xmm3, xmm3                  // xmm3 = s
        movaps      xmm5, SIMD_SP_atan_c0
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c1
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c2
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c3
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c4
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c5
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c6
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_atan_c7
        mulps       xmm5, xmm3
        addps       xmm5, SIMD_SP_one
        mulps       xmm5, xmm0
        addps       xmm5, xmm4                  // xmm5 = omega0

        movss       xmm6, lerp                  // xmm6 = lerp
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 0, 0, 0, 0 )
        mulps       xmm6, xmm5                  // xmm6 = omega1
        subps       xmm5, xmm6                  // xmm5 = omega0

        // scale0 = sin( xmm5 ) * xmm2
        // scale1 = sin( xmm6 ) * xmm2
        movaps      xmm3, xmm5
        movaps      xmm7, xmm6
        mulps       xmm3, xmm3
        mulps       xmm7, xmm7
        movaps      xmm4, SIMD_SP_sin_c0
        movaps      xmm0, SIMD_SP_sin_c0
        mulps       xmm4, xmm3
        mulps       xmm0, xmm7
        addps       xmm4, SIMD_SP_sin_c1
        addps       xmm0, SIMD_SP_sin_c1
        mulps       xmm4, xmm3
        mulps       xmm0, xmm7
        addps       xmm4, SIMD_SP_sin_c2
        addps       xmm0, SIMD_SP_sin_c2
        mulps       xmm4, xmm3
        mulps       xmm0, xmm7
        addps       xmm4, SIMD_SP_sin_c3
        addps       xmm0, SIMD_SP_sin_c3
        mulps       xmm4, xmm3
        mulps       xmm0, xmm7
        addps       xmm4, SIMD_SP_sin_c4
        addps       xmm0, SIMD_SP_sin_c4
        mulps       xmm4, xmm3
        mulps       xmm0, xmm7
        addps       xmm4, SIMD_SP_one
        addps       xmm0, SIMD_SP_one
        mulps       xmm5, xmm4
        mulps       xmm6, xmm0
        mulps       xmm5, xmm2                  // xmm5 = scale0
        mulps       xmm6, xmm2                  // xmm6 = scale1

        xorps       xmm6, xmm1

        movaps      xmm0, jointQuat0
        mulps       xmm0, xmm5
        movaps      xmm1, blendQuat0
        mulps       xmm1, xmm6
        addps       xmm0, xmm1

        movaps      xmm1, jointQuat1
```

```
        mulps       xmm1, xmm5
        movaps      xmm2, blendQuat1
        mulps       xmm2, xmm6
        addps       xmm1, xmm2

        movaps      xmm2, jointQuat2
        mulps       xmm2, xmm5
        movaps      xmm3, blendQuat2
        mulps       xmm3, xmm6
        addps       xmm2, xmm3

        movaps      xmm3, jointQuat3
        mulps       xmm3, xmm5
        movaps      xmm4, blendQuat3
        mulps       xmm4, xmm6
        addps       xmm3, xmm4

        add         eax, 4*4

        // transpose xmm0, xmm1, xmm2, xmm3 to memory
        movaps      xmm7, xmm0
        movaps      xmm6, xmm2

        unpcklps    xmm0, xmm1
        unpcklps    xmm2, xmm3

        mov         ecx, a0
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm0
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm2

        mov         ecx, a1
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm0
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm2

        unpckhps    xmm7, xmm1
        unpckhps    xmm6, xmm3

        mov         ecx, a2
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm7
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm6

        mov         ecx, a3
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm7
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm6

        jle         loopJoint4

    done4:
        sub         eax, 4*4
        jz          done1

    loopJoint1:
        movss       xmm3, lerp
        shufps      xmm3, xmm3, R_SHUFFLE_PS( 0, 0, 0, 0 )

        mov         ecx, [edx+eax]
        shl         ecx, JOINTQUAT_SIZE_SHIFT

        // lerp first translations
        movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
        subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        mulps       xmm7, xmm3
        addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

        // load first quaternions
        movaps      xmm0, [esi+ecx+JOINTQUAT_Q_OFFSET]
        movaps      xmm1, [edi+ecx+JOINTQUAT_Q_OFFSET]

        movaps      jointQuat0, xmm0
        movaps      blendQuat0, xmm1

        // lerp quaternions
        mulps       xmm1, xmm0
        movhlps     xmm0, xmm1
        addps       xmm1, xmm0
        movaps      xmm0, xmm1
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 1, 0, 2, 3 )
        addss       xmm0, xmm1                  // xmm0 = cosom

        movss       xmm1, xmm0
```

```
        movss       xmm2, xmm0
        andps       xmm1, SIMD_SP_signBit       // xmm1 = signBit
        xorps       xmm0, xmm1
        mulss       xmm2, xmm2

        xorps       xmm4, xmm4
        movss       xmm3, SIMD_SP_one
        subss       xmm3, xmm2                  // xmm3 = scale0
        cmpeqss     xmm4, xmm3
        andps       xmm4, SIMD_SP_tiny          // if values are zero replace them with a tiny number
        andps       xmm3, SIMD_SP_absMask       // make sure the values are positive
        orps        xmm3, xmm4

        movss       xmm2, xmm3
        rsqrtss     xmm4, xmm2
        mulss       xmm2, xmm4
        mulss       xmm2, xmm4
        subss       xmm2, SIMD_SP_rsqrt_c0
        mulss       xmm4, SIMD_SP_rsqrt_c1
        mulss       xmm2, xmm4
        mulss       xmm3, xmm2                  // xmm3 = sqrt( scale0 )

        // omega0 = atan2( xmm3, xmm0 )
        movss       xmm4, xmm0
        minss       xmm0, xmm3
        maxss       xmm3, xmm4
        cmpeqss     xmm4, xmm0
        rcpss       xmm5, xmm3
        mulss       xmm3, xmm5
        mulss       xmm3, xmm5
        addss       xmm5, xmm5
        subss       xmm5, xmm3                  // xmm5 = 1 / y or 1 / x
        mulss       xmm0, xmm5                  // xmm0 = x / y or y / x
        movss       xmm3, xmm4
        andps       xmm3, SIMD_SP_signBit
        xorps       xmm0, xmm3                  // xmm0 = -x / y or y / x
        andps       xmm4, SIMD_SP_halfPI        // xmm4 = HALF_PI or 0.0f
        movss       xmm3, xmm0
        mulss       xmm3, xmm3                  // xmm3 = s
        movss       xmm5, SIMD_SP_atan_c0
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c1
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c2
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c3
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c4
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c5
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c6
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_atan_c7
        mulss       xmm5, xmm3
        addss       xmm5, SIMD_SP_one
        mulss       xmm5, xmm0
        addss       xmm5, xmm4                  // xmm5 = omega0

        movss       xmm6, lerp                  // xmm6 = lerp
        mulss       xmm6, xmm5                  // xmm6 = omega1
        subss       xmm5, xmm6                  // xmm5 = omega0

        // scale0 = sin( xmm5 ) * xmm2
        // scale1 = sin( xmm6 ) * xmm2
        movss       xmm3, xmm5
        movss       xmm7, xmm6
        mulss       xmm3, xmm3
        mulss       xmm7, xmm7
        movss       xmm4, SIMD_SP_sin_c0
        movss       xmm0, SIMD_SP_sin_c0
        mulss       xmm4, xmm3
        mulss       xmm0, xmm7
        addss       xmm4, SIMD_SP_sin_c1
        addss       xmm0, SIMD_SP_sin_c1
        mulss       xmm4, xmm3
        mulss       xmm0, xmm7
        addss       xmm4, SIMD_SP_sin_c2
        addss       xmm0, SIMD_SP_sin_c2
        mulss       xmm4, xmm3
        mulss       xmm0, xmm7
```

```
        addss       xmm4, SIMD_SP_sin_c3
        addss       xmm0, SIMD_SP_sin_c3
        mulss       xmm4, xmm3
        mulss       xmm0, xmm7
        addss       xmm4, SIMD_SP_sin_c4
        addss       xmm0, SIMD_SP_sin_c4
        mulss       xmm4, xmm3
        mulss       xmm0, xmm7
        addss       xmm4, SIMD_SP_one
        addss       xmm0, SIMD_SP_one
        mulss       xmm5, xmm4
        mulss       xmm6, xmm0
        mulss       xmm5, xmm2                  // xmm5 = scale0
        mulss       xmm6, xmm2                  // xmm6 = scale1

        xorps       xmm6, xmm1

        shufps      xmm5, xmm5, R_SHUFFLE_PS( 0, 0, 0, 0 )
        mulps       xmm5, jointQuat0
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 0, 0, 0, 0 )
        mulps       xmm6, blendQuat0
        addps       xmm5, xmm6

        movaps      [esi+ecx+JOINTQUAT_Q_OFFSET], xmm5

        add         eax, 1*4
        jl          loopJoint1

    done1:
    }
}
```

Appendix B

```
/*
    SSE Optimized Linear Interpolation between Quaternions
    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 LerpJoints( JointQuat *joints, const JointQuat *blendJoints, const float lerp, const int *index, const int
numJoints ) {

    assert_16_byte_aligned( joints );
    assert_16_byte_aligned( blendJoints );
    assert_16_byte_aligned( JOINTQUAT_Q_OFFSET );
    assert_16_byte_aligned( JOINTQUAT_T_OFFSET );

    ALIGN16( float jointQuat3[4]; )
    ALIGN16( float blendQuat3[4]; )
    ALIGN16( float scaledLerp; )
    int a0, a1, a2, a3;

    __asm {
        movss       xmm7, lerp
        cmpnless    xmm7, SIMD_SP_zero
        movmskps    ecx, xmm7
        test        ecx, 1
        jz          done1

        mov         eax, numJoints
        shl         eax, 2
        mov         esi, joints
        mov         edi, blendJoints
        mov         edx, index

        add         edx, eax
```

```
        neg         eax
        jz          done1

        movss       xmm7, lerp
        cmpnltss    xmm7, SIMD_SP_one
        movmskps    ecx, xmm7
        test        ecx, 1
        jz          lerpJoints

    loopCopy:
        mov         ecx, [edx+eax]
        shl         ecx, JOINTQUAT_SIZE_SHIFT

        add         eax, 1*4

        movaps      xmm0, [edi+ecx+JOINTQUAT_Q_OFFSET]
        movaps      xmm1, [edi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_Q_OFFSET], xmm0
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm1

        jl          loopCopy

        jmp         done1

    lerpJoints:
        movss       xmm7, lerp
        movss       xmm6, SIMD_SP_one
        subss       xmm6, xmm7
        divss       xmm7, xmm6
        movss       scaledLerp, xmm7

        add         eax, 4*4
        jge         done4

    loopJoint4:
        movss       xmm3, lerp
        shufps      xmm3, xmm3, R_SHUFFLE_PS( 0, 0, 0, 0 )

        mov         ecx, [edx+eax-4*4]
        shl         ecx, JOINTQUAT_SIZE_SHIFT
        mov         a0, ecx

        // lerp first translations
        movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
        subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        mulps       xmm7, xmm3
        addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

        // load first quaternions
        movaps      xmm0, [esi+ecx+JOINTQUAT_Q_OFFSET]
        movaps      xmm4, [edi+ecx+JOINTQUAT_Q_OFFSET]

        mov         ecx, [edx+eax-3*4]
        shl         ecx, JOINTQUAT_SIZE_SHIFT
        mov         a1, ecx

        // lerp second translations
        movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
        subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        mulps       xmm7, xmm3
        addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

        // load second quaternions
        movaps      xmm1, [esi+ecx+JOINTQUAT_Q_OFFSET]
        movaps      xmm5, [edi+ecx+JOINTQUAT_Q_OFFSET]

        mov         ecx, [edx+eax-2*4]
        shl         ecx, JOINTQUAT_SIZE_SHIFT
        mov         a2, ecx

        // lerp third translations
        movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
        subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        mulps       xmm7, xmm3
        addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

        // load third quaternions
        movaps      xmm2, [esi+ecx+JOINTQUAT_Q_OFFSET]
```

```
movaps      xmm6, [edi+ecx+JOINTQUAT_Q_OFFSET]

mov         ecx, [edx+eax-1*4]
shl         ecx, JOINTQUAT_SIZE_SHIFT
mov         a3, ecx

// lerp fourth translations
movaps      xmm7, [edi+ecx+JOINTQUAT_T_OFFSET]
subps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
mulps       xmm7, xmm3
addps       xmm7, [esi+ecx+JOINTQUAT_T_OFFSET]
movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm7

// load fourth quaternions
movaps      xmm3, [esi+ecx+JOINTQUAT_Q_OFFSET]

TRANSPOSE_4x4( xmm0, xmm1, xmm2, xmm3, xmm7 )

movaps      jointQuat3, xmm3

movaps      xmm7, [edi+ecx+JOINTQUAT_Q_OFFSET]

TRANSPOSE_4x4( xmm4, xmm5, xmm6, xmm7, xmm3 )

movaps      blendQuat3, xmm7

// lerp quaternions
movaps      xmm3, xmm0
mulps       xmm3, xmm4
movaps      xmm7, xmm1
mulps       xmm7, xmm5
addps       xmm3, xmm7
movaps      xmm7, xmm2
mulps       xmm7, xmm6
addps       xmm3, xmm7
movaps      xmm7, jointQuat3
mulps       xmm7, blendQuat3
addps       xmm3, xmm7                  // xmm3 = cosom
andps       xmm3, SIMD_SP_signBit       // xmm3 = signBit
movss       xmm7, scaledLerp
shufps      xmm7, xmm7, R_SHUFFLE_PS( 0, 0, 0, 0 )
xorps       xmm7, xmm3                  // xmm7 = scaledLerp ^ signBit

mulps       xmm4, xmm7
addps       xmm4, xmm0
movaps      xmm0, xmm4
mulps       xmm0, xmm0

mulps       xmm5, xmm7
addps       xmm5, xmm1
movaps      xmm1, xmm5
mulps       xmm1, xmm1
addps       xmm0, xmm1

mulps       xmm6, xmm7
addps       xmm6, xmm2
movaps      xmm2, xmm6
mulps       xmm2, xmm2
addps       xmm0, xmm2

mulps       xmm7, blendQuat3
addps       xmm7, jointQuat3
movaps      xmm1, xmm7
mulps       xmm1, xmm1
addps       xmm0, xmm1

rsqrtps     xmm2, xmm0
mulps       xmm0, xmm2
mulps       xmm0, xmm2
subps       xmm0, SIMD_SP_rsqrt_c0
mulps       xmm2, SIMD_SP_rsqrt_c1
mulps       xmm0, xmm2

mulps       xmm4, xmm0
mulps       xmm5, xmm0
mulps       xmm6, xmm0
mulps       xmm7, xmm0

add         eax, 4*4

// transpose xmm4, xmm5, xmm6, xmm7 to memory
```

```
        movaps      xmm2, xmm4
        movaps      xmm3, xmm6

        unpcklps    xmm4, xmm5
        unpcklps    xmm6, xmm7

        mov         ecx, a0
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm4
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm6

        mov         ecx, a1
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm4
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm6

        unpckhps    xmm2, xmm5
        unpckhps    xmm3, xmm7

        mov         ecx, a2
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm2
        movlps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm3

        mov         ecx, a3
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+0], xmm2
        movhps      [esi+ecx+JOINTQUAT_Q_OFFSET+8], xmm3

        jle         loopJoint4

    done4:
        sub         eax, 4*4
        jz          done1

        movss       xmm6, lerp
        shufps      xmm6, xmm6, R_SHUFFLE_PS( 0, 0, 0, 0 )
        movss       xmm7, scaledLerp
        shufps      xmm7, xmm7, R_SHUFFLE_PS( 0, 0, 0, 0 )

    loopJoint1:

        mov         ecx, [edx+eax]
        shl         ecx, JOINTQUAT_SIZE_SHIFT

        // lerp translations
        movaps      xmm3, [edi+ecx+JOINTQUAT_T_OFFSET]
        subps       xmm3, [esi+ecx+JOINTQUAT_T_OFFSET]
        mulps       xmm3, xmm6
        addps       xmm3, [esi+ecx+JOINTQUAT_T_OFFSET]
        movaps      [esi+ecx+JOINTQUAT_T_OFFSET], xmm3

        // load quaternions
        movaps      xmm0, [esi+ecx+JOINTQUAT_Q_OFFSET]
        movaps      xmm1, [edi+ecx+JOINTQUAT_Q_OFFSET]

        // lerp quaternions
        movaps      xmm2, xmm0
        mulps       xmm2, xmm1
        movhlps     xmm3, xmm2
        addps       xmm2, xmm3
        movaps      xmm3, xmm2
        shufps      xmm3, xmm3, R_SHUFFLE_PS( 1, 0, 2, 3 )
        addss       xmm3, xmm2                  // xmm3 = cosom
        shufps      xmm3, xmm3, R_SHUFFLE_PS( 0, 0, 0, 0 )
        andps       xmm3, SIMD_SP_signBit       // xmm3 = signBit
        xorps       xmm3, xmm7                  // xmm3 = scaledLerp ^ signBit

        mulps       xmm1, xmm3
        addps       xmm1, xmm0                  // xmm1 = jointQuat + scale * blendQuat

        movaps      xmm0, xmm1
        mulps       xmm0, xmm0
        movhlps     xmm2, xmm0
        addps       xmm0, xmm2
        movaps      xmm2, xmm0
        shufps      xmm2, xmm2, R_SHUFFLE_PS( 1, 0, 2, 3 )
        addss       xmm0, xmm2

        rsqrtss     xmm2, xmm0
        mulss       xmm0, xmm2
        mulss       xmm0, xmm2
        subss       xmm0, SIMD_SP_rsqrt_c0
        mulss       xmm2, SIMD_SP_rsqrt_c1
        mulss       xmm0, xmm2
```

```
        shufps      xmm0, xmm0, R_SHUFFLE_PS( 0, 0, 0, 0 )
        mulps       xmm1, xmm0
        movaps      [esi+ecx+JOINTQUAT_Q_OFFSET], xmm1

        add         eax, 1*4
        jl          loopJoint1

    done1:

    }
}
```