Deriving Triangle Plane Equations

April 8th 2005
J.M.P. van Waveren

Abstract

An optimized routine to derive triangle plane equations is presented. The Intel Streaming SIMD Extensions are used to exploit parallelism through increased throughput. The optimized routine is well over four times faster than the commonly used implementation in C on a Pentium 4.

1. Introduction

A plane divides space in two regions or half spaces and is a well known geometric primitive that is commonly used in many applications. Planes are often used to describe geometric properties and to define geometric relationships. Many applications require such geometric relationships to be evaluated in real time.

The equations of planes are often used in real-time rendering applications. Backface culling is the process of determining the polygons that face away from the viewer and as such should not be rendered because the polygons are only considered visible from one side. The plane equation of a polygon can be used to cull a polygon that faces away from the viewer by determining whether or not the viewer is in the half space at the front of the polygon [8].

Several rendering algorithms require the detection of silhouette edges based on the facing of triangles. The construction of shadow volumes uses the triangle plane equations to find the triangles that face towards or away from the light source and to determine the shadow silhouette of the geometry [14]. Silhouette edge detection based on plane equations is also used for cartoon rendering [13].

Many shading algorithms among which Gouraud and Phong shading use the surface normal vectors at vertex positions of polygonal or triangle meshes [8]. A vertex normal can be calculated by averaging or weighing the plane normal vectors of all the polygons or triangles that use the vertex [9,10,11,12]. Surface normals can be calculated from the vertex normals through interpolation. These surface normals can also be used to create normal maps for bump map rendering.

Reducing the level of detail of geometry can also be accomplished based on the plane equations of polygons or triangles. Near planar components can often be merged and replaced by lower detail geometry without significantly changing the silhouette of the overall geometric shape.

Collision detection is another area where plane equations are commonly used to identify geometric relationships. The intersections of geometric primitives with planes are typically used to determine the regions in contact or to calculate when objects are going to collide.

When a polygonal surface animates the equations of the polygon planes change. In particular a non-degenerate triangle is by definition planar and if a triangle mesh animates the plane equations of the triangles change. As such these plane equations have to be recalculated continuously when they are used to evaluate geometric properties in real time. In this article the Intel Streaming SIMD Extensions are used to optimize an algorithm that derives plane equations for the triangles in a triangle mesh.

1.1 Previous Work

Weingartner and Klimovitski [15] describe methods for facet normal based triangle culling and facet normal compression. The facet normals are derived from the triangle vertices using the Intel Streaming SIMD Extensions.

1.2 Layout

Section 2 goes into the details of plane equations. Section 3 describes the basic algorithm used to derive triangle plane equations from the triangle vertices. Sections 4 describes how this algorithm can be optimized using the Intel Streaming SIMD Extensions. The results of the optimizations are presented in section 5 and several conclusions are drawn in section 6.

2. Plane Equations

A plane is a two-dimensional doubly ruled surface spanned by two linearly independent vectors. The generalization of the plane to higher dimensions is called a hyperplane. A plane divides space in two regions or half spaces. The general equation of a plane in R^3 is defined as follows.

$$ax + by + cz + d = 0$$

Where 'a', 'b' and 'c' describe a vector orthogonal to the plane, and 'd' is the distance of the plane from the origin as a multiple of the vector (a, b, c). The (x, y, z) in the above equation specify an arbitrary point in the plane. A plane will be represented in code as follows.

```
struct Plane {
    float   a, b, c, d;
};
```

It is often convenient to specify planes in so called Hessian normal form. This form is obtained by dividing the plane equation by the length of the vector (a, b, c) as follows [3].

$$n_x x + n_y y + n_z z + p = \frac{ax + by + cz + d}{(a^2 + b^2 + c^2)^{½}}$$

This is equivalent to making $a^2 + b^2 + c^2 = 1$ in the general plane equation. In other words the vector orthogonal to the plane is unit length and the plane constant 'd' is adjusted appropriately. Given the Hessian normal form the point-plane distance from the point (x, y, z) to a plane is given by the following simple equation.

$$D = n_x x + n_y y + n_z z + p$$

If the point (x, y, z) is in the half-space determined by the direction (n_x, n_y, n_z) then $D > 0$. If the point is in the other half-space then $D < 0$.

If three points in R^3 are taken there may be a straight line in R^3 which passes through all three points. If such a line exists then these three points are said to be collinear. Given three points in R^3 which are not collinear, there is exactly one plane which passes through all three points.

Two linearly independent vectors in the plane can be derived from three points which are not collinear by subtracting one of the points from the other two points. The cross product of the two linearly independent vectors can then be used to obtain a vector orthogonal to the plane. This vector provides the 'a', 'b' and 'c' in the general plane equation. The plane constant 'd' can be calculated with the dot product of the vector orthogonal to the plane with one of the three points.

When a plane equation is derived for a triangle specified by three points the vector orthogonal to the plane can point in one of two directions. By consistently specifying the points that define the triangle in counter-clockwise order this vector can be made to always point into the half space at the front of the triangle.

3. Deriving Triangle Planes

The routine presented here derives triangle plane equations for the triangles of an arbitrary triangle mesh. The triangle mesh is specified as an array with vertices and an array with indices. For each triangle this array with indices contains three elements with the numbers of the vertices that create the triangle. The three vertices of each triangle are specified in counter-clockwise order. A vertex in the array with vertices is represented in code as follows.

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

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

The vertex uses 4D vectors for the position and normal while 3D vectors may be sufficient. However, using 4D vectors improves memory alignment and the last component of the 4D vectors could also be used for other purposes. For some of the vertex properties 3D vectors could be used and interleaved with new properties, like 4 byte colors, to maintain alignment. However, simple 4D vectors are used here to achieve good memory alignment with minimal complexity.

Two linearly independent vectors span a plane. Two such linearly independent vectors in the plane can be derived from three triangle vertices by subtracting the position of one vertex from the positions of the other two vertices. From these two vectors a new vector can be derived which is orthogonal to both vectors and normal to the plane. This orthogonal vector is calculated by taking the cross product of the two linearly independent vectors. Furthermore this orthogonal vector is normalized to create a plane equation in Hessian normal form. The plane distance from the origin along the normal vector is then calculated with a dot product between the normal vector and the position of one of the triangle vertices.

```
void DeriveTrianglePlanes( Plane *planes, const Vertex *verts, const int numVerts, const int *indices,
const int numIndices ) {
    int i;

    for ( i = 0; i < numIndices; i += 3, planes++ ) {
        const Vertex *v0, *v1, *v2;
        float d0[3], d1[3], n[3], s;

        v0 = verts + indices[i + 0];
        v1 = verts + indices[i + 1];
        v2 = verts + indices[i + 2];

        d0[0] = v1->position.x - v0->position.x;
        d0[1] = v1->position.y - v0->position.y;
        d0[2] = v1->position.z - v0->position.z;

        d1[0] = v2->position.x - v0->position.x;
        d1[1] = v2->position.y - v0->position.y;
        d1[2] = v2->position.z - v0->position.z;

        n[0] = d1[1] * d0[2] - d1[2] * d0[1];
        n[1] = d1[2] * d0[0] - d1[0] * d0[2];
        n[2] = d1[0] * d0[1] - d1[1] * d0[0];

        s = 1.0f / sqrt( n[0] * n[0] + n[1] * n[1] + n[2] * n[2] );

        planes->a = n[0] * s;
        planes->b = n[1] * s;
        planes->c = n[2] * s;
        planes->d = -( planes->a * v0->position.x + planes->b * v0->position.y + planes->c * v0-
>position.z );
    }
}
```

Although the mathematics involved in deriving plane equations for triangles is not very complex the above routine can consume a considerable amount of time when many triangles have to be processed. Fortunately the above routine can be optimized using the Intel Streaming SIMD Extensions as shown in the next section.

4. Deriving Triangle Planes With SSE

The best approach to SIMD for deriving triangle plane equations is to exploit parallelism through increased throughput. The routine presented here will operate on four triangles per iteration and the scalar instructions are replaced with functionally equivalent SSE instructions. This requires a swizzle because the vertex coordinates are stored per vertex and each triangle may reference three arbitrary vertices while the coordinates of the twelve vertices used by four triangles need to be grouped into SSE registers. The following SSE code swizzles the coordinates of four vertices into three SSE registers.

```
movlps      xmm1, [esi+ecx+VERTEX_POSITION_OFFSET+0]     /* xmm1 =  0,  1,  X,  X */
movss       xmm2, [esi+ecx+VERTEX_POSITION_OFFSET+8]     /* xmm2 =  2,  X,  X,  X */
movhps      xmm1, [esi+ebx+VERTEX_POSITION_OFFSET+0]     /* xmm1 =  0,  1,  4,  5 */
movhps      xmm2, [esi+ebx+VERTEX_POSITION_OFFSET+8]     /* xmm2 =  2,  X,  6,  X */
movlps      xmm6, [esi+edx+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9,  X,  X */
movss       xmm7, [esi+edx+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X,  X,  X */
movhps      xmm6, [esi+eax+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9, 12, 13 */
movhps      xmm7, [esi+eax+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X, 14,  X */
movaps      xmm0, xmm1                                   /* xmm0 =  0,  1,  4,  5 */
shufps      xmm0, xmm6, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm0 =  0,  4,  8, 12 */
shufps      xmm1, xmm6, R_SHUFFLE_PS( 1, 3, 1, 3 )       /* xmm1 =  1,  5,  9, 13 */
shufps      xmm2, xmm7, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm2 =  2,  6, 10, 14 */
```

After swizzling 12 vertices into SSE registers the scalar instructions can be replaced with functionally equivalent SSE instructions.

To create plane equations in Hessian normal form the vector orthogonal to the plane is divided by its length. This requires the calculation of a reciprocal square root. The Intel SSE instruction set has an instruction to calculate the reciprocal square root with 12 bits of precision. Using an approximation for the reciprocal square root does not change the accuracy of the plane equation. It only means the plane equation will not be in true Hessian normal form although very close. If necessary a simple Newton-Rapson iteration can be used to improve the accuracy of the reciprocal square root [16]. When the triangle plane equations do not need to be in Hessian normal form the normalization of the vector orthogonal to the plane can be completely omitted from the optimized code which makes the routine even faster.

The complete routine for deriving triangle plane equations is listed in appendix A. The code works with any alignment but for the best performance the list with vertices should be at least 16 byte aligned. The size of vertex objects (Vertex) should be a multiple of 16 bytes such that consecutive vertices in an array are all aligned on a 16 byte boundary. The routine currently assumes the size of the vertex objects (Vertex) is a power of two. If this is not the case the shift with VERTEX_SIZE_SHIFT must be replaced with an integer multiplication. However, this is not recommended because integer multiplications are more expensive than bitwise logical shifts.

5. Results

The 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 triangles using 1024 vertices. The total

number of clock cycles and the number of clock cycles per triangle 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**
DeriveTrianglePlanes (C)	141192	138	149483	146
DeriveTrianglePlanes (SSE)	34080	33	36128	35

6. Conclusion

The plane equations of triangles are often used to evaluate geometric relationships in real-time applications. When a triangle mesh animates the plane equations of the triangles change. Deriving the triangle plane equations from the animating triangle vertices can be performance critical in such applications.

Deriving triangle plane equations from triangle vertices can be optimized using the Intel Streaming SIMD Extensions. The optimized algorithm presented here is well over four times faster than the commonly used algorithm in C.

7. References

1. Solid Mensuration with Proofs - §4 - Lines and Planes in Space
   W. F. Kern, J. R. Bland
   2nd ed., New York: Wiley, pp. 9-12, 1948

2. A New Approach to the Shaded Picture Problem
   M. E. Newell, R. G. Newell, T. L. Sancha
   In Proceedings of the ACM National Conference, pp. 443-450, 1972

3. VNR Concise Encyclopedia of Mathematics, 2nd Edition
   W. Gellert, S. Gottwald, M. Hellwich, H. Kästner, H. Künstner (Eds.).
   New York: Van Nostrand Reinhold, pp. 539-543, 1989
   Available Online:
   http://www.amazon.com/exec/obidos/ASIN/0442205902/weisstein-20

4. Newell's Method for Computing the Plane Equation of a Polygon
   Filippo Tampieri
   Graphics Gems III, David Kirk, Academic Press, pp. 231-232, 1992
   Available Online: http://www.graphicsgems.org/

5. Fast Polygon Area and Newell Normal Computation
   Daniel Sunday
   Journal of Graphics Tools, vol. 7, no. 2, pp. 9-13, 2002

Available Online: http://www.acm.org/jgt/papers/Sunday02/

6. CRC Standard Mathematical Tables and Formulae, 31st Edition
   Daniel Zwillinger
   Chapman & Hall/CRC, 31st edition, November 27, 2002
   Available Online:
   http://www.amazon.com/exec/obidos/ASIN/1584882913/weisstein-20

7. Geometric Tools For Computer Graphics - 12.6 Plane Through Three Points
   Philip J. Schneider, David H. Eberly
   Morgan Kaufmann Publishers, pp. 669-670, 2003 by Elsevier Science

8. Computer Graphics
   Donald Hearn, M. Pauline Baker
   Prentice Hall International Inc, pp. 523, 1994

9. Computing Surface Normals for 3D Models
   Andrew S. Glassner, ed.
   Graphics Gems, Academic Press, pp. 562-566, 1990.
   Available Online: http://www.graphicsgems.org/

10. Building Vertex Normals from an Unstructured Polygon List
    Andrew S. Glassner
    Paul S. Heckbert, ed.
    Graphics Gems IV, Academic Press, pp. 60-73, 1994.
    Available Online: http://www.graphicsgems.org/

11. Computing Vertex Normals from Polygonal Facets
    Grit Thürmer, Charles A. Wüthrich
    Journal of Graphics Tools, vol. 3, no. 1, pp. 43-46, 1998.
    Available Online: http://portal.acm.org/citation.cfm?id=317263

12. Weights for Computing Vertex Normals from Facet Normals
    Nelson Max
    Journal of Graphics Tools, vol. 4, no. 2, pp. 1-6, 1999.
    Available Online: http://portal.acm.org/citation.cfm?id=334710

13. Cartoon Rendering: Real-time Silhouette Edge Detection and Rendering
    Carl S. Marshall
    Game Programming Gems 2, Mark Deloura (editor), Charles River Media, 2001

14. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering
    Cass Everitt, Mark J. Kilgard

nVidia Corporation, 2002
Available Online:
http://developer.nvidia.com/object/robust_shadow_volumes.html

15. Power Programming with the Streaming SIMD Extensions 2
Markus. Weingartner, Alex. Klimovitski
Intel Developer Forum Conference, Spring 2001

16. 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 Calculation of Triangle Plane Equations
    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 );

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

struct Vertex {
    Vec4    position;
    Vec4    normal;
};

#define VERTEX_SIZE_SHIFT                   5
#define VERTEX_SIZE                         (8*4)       // sizeof( Vertex )
#define VERTEX_POSITION_OFFSET              (0*4)       // offsetof( Vertex, position )
#define VERTEX_NORMAL_OFFSET                (4*4)       // offsetof( Vertex, normal )

void DeriveTrianglePlanes( Plane *planes, const Vertex *verts, const int numVerts, const int *indices,
const int numIndices ) {
    int d, a;
    int n = numIndexes / 3;
    ALIGN16( float x0[4] );
    ALIGN16( float x1[4] );
    ALIGN16( float x2[4] );

    __asm {
```

```
        push        ebx
        mov         eax, n
        shl         eax, 4
        mov         esi, verts
        mov         edi, indexes
        mov         edx, planes

        add         edx, eax
        neg         eax

        mov         d, edx

        add         eax, 4*16
        jge         done4

    loopPlane4:
        mov         a, eax

        mov         ecx, [edi+0*12+0]
        shl         ecx, VERTEX_SIZE_SHIFT
        mov         ebx, [edi+1*12+0]
        shl         ebx, VERTEX_SIZE_SHIFT
        mov         edx, [edi+2*12+0]
        shl         edx, VERTEX_SIZE_SHIFT
        mov         eax, [edi+3*12+0]
        shl         eax, VERTEX_SIZE_SHIFT

        movlps      xmm4, [esi+ecx+VERTEX_POSITION_OFFSET+0]     /* xmm4 =  0,  1,  X,  X */
        movss       xmm5, [esi+ecx+VERTEX_POSITION_OFFSET+8]     /* xmm5 =  2,  X,  X,  X */
        movhps      xmm4, [esi+ebx+VERTEX_POSITION_OFFSET+0]     /* xmm4 =  0,  1,  4,  5 */
        movhps      xmm5, [esi+ebx+VERTEX_POSITION_OFFSET+8]     /* xmm5 =  2,  X,  6,  X */
        movlps      xmm6, [esi+edx+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9,  X,  X */
        movss       xmm7, [esi+edx+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X,  X,  X */
        movhps      xmm6, [esi+eax+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9, 12, 13 */
        movhps      xmm7, [esi+eax+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X, 14,  X */
        movaps      xmm3, xmm4                                   /* xmm3 =  0,  1,  4,  5 */
        shufps      xmm3, xmm6, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm3 =  0,  4,  8, 12 */
        shufps      xmm4, xmm6, R_SHUFFLE_PS( 1, 3, 1, 3 )       /* xmm4 =  1,  5,  9, 13 */
        shufps      xmm5, xmm7, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm5 =  2,  6, 10, 14 */

        mov         ecx, [edi+0*12+4]
        shl         ecx, VERTEX_SIZE_SHIFT
        mov         ebx, [edi+1*12+4]
        shl         ebx, VERTEX_SIZE_SHIFT
        mov         edx, [edi+2*12+4]
        shl         edx, VERTEX_SIZE_SHIFT
        mov         eax, [edi+3*12+4]
        shl         eax, VERTEX_SIZE_SHIFT

        movaps      x0, xmm3
        movaps      x1, xmm4
        movaps      x2, xmm5

        movlps      xmm1, [esi+ecx+VERTEX_POSITION_OFFSET+0]     /* xmm1 =  0,  1,  X,  X */
        movss       xmm2, [esi+ecx+VERTEX_POSITION_OFFSET+8]     /* xmm2 =  2,  X,  X,  X */
        movhps      xmm1, [esi+ebx+VERTEX_POSITION_OFFSET+0]     /* xmm1 =  0,  1,  4,  5 */
        movhps      xmm2, [esi+ebx+VERTEX_POSITION_OFFSET+8]     /* xmm2 =  2,  X,  6,  X */
        movlps      xmm6, [esi+edx+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9,  X,  X */
        movss       xmm7, [esi+edx+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X,  X,  X */
        movhps      xmm6, [esi+eax+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9, 12, 13 */
        movhps      xmm7, [esi+eax+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X, 14,  X */
        movaps      xmm0, xmm1                                   /* xmm0 =  0,  1,  4,  5 */
        shufps      xmm0, xmm6, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm0 =  0,  4,  8, 12 */
        shufps      xmm1, xmm6, R_SHUFFLE_PS( 1, 3, 1, 3 )       /* xmm1 =  1,  5,  9, 13 */
        shufps      xmm2, xmm7, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm2 =  2,  6, 10, 14 */

        mov         ecx, [edi+0*12+8]
        shl         ecx, VERTEX_SIZE_SHIFT
        mov         ebx, [edi+1*12+8]
        shl         ebx, VERTEX_SIZE_SHIFT
        mov         edx, [edi+2*12+8]
        shl         edx, VERTEX_SIZE_SHIFT
        mov         eax, [edi+3*12+8]
        shl         eax, VERTEX_SIZE_SHIFT

        subps       xmm0, xmm3
        subps       xmm1, xmm4
        subps       xmm2, xmm5

        movlps      xmm4, [esi+ecx+VERTEX_POSITION_OFFSET+0]     /* xmm4 =  0,  1,  X,  X */
        movss       xmm5, [esi+ecx+VERTEX_POSITION_OFFSET+8]     /* xmm5 =  2,  X,  X,  X */
```

```
        movhps      xmm4, [esi+ebx+VERTEX_POSITION_OFFSET+0]     /* xmm4 =  0,  1,  4,  5 */
        movhps      xmm5, [esi+ebx+VERTEX_POSITION_OFFSET+8]     /* xmm5 =  2,  X,  6,  X */
        movlps      xmm6, [esi+edx+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9,  X,  X */
        movss       xmm7, [esi+edx+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X,  X,  X */
        movhps      xmm6, [esi+eax+VERTEX_POSITION_OFFSET+0]     /* xmm6 =  8,  9, 12, 13 */
        movhps      xmm7, [esi+eax+VERTEX_POSITION_OFFSET+8]     /* xmm6 = 10,  X, 14,  X */
        movaps      xmm3, xmm4                                   /* xmm3 =  0,  1,  4,  5 */
        shufps      xmm3, xmm6, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm3 =  0,  4,  8, 12 */
        shufps      xmm4, xmm6, R_SHUFFLE_PS( 1, 3, 1, 3 )       /* xmm4 =  1,  5,  9, 13 */
        shufps      xmm5, xmm7, R_SHUFFLE_PS( 0, 2, 0, 2 )       /* xmm5 =  2,  6, 10, 14 */

        mov         eax, a
        mov         edx, d
        add         edi, 4*12

        subps       xmm3, x0
        subps       xmm4, x1
        subps       xmm5, x2

        movaps      xmm6, xmm4
        mulps       xmm6, xmm2
        movaps      xmm7, xmm5
        mulps       xmm7, xmm1
        subps       xmm6, xmm7

        mulps       xmm5, xmm0
        mulps       xmm2, xmm3
        subps       xmm5, xmm2

        mulps       xmm3, xmm1
        mulps       xmm4, xmm0
        subps       xmm3, xmm4

        add         eax, 4*16

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

        addps       xmm3, xmm5
        addps       xmm3, xmm6
        rsqrtps     xmm3, xmm3

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

        movaps      xmm4, x0
        movaps      xmm5, x1
        movaps      xmm6, x2

        mulps       xmm4, xmm0
        mulps       xmm5, xmm1
        mulps       xmm6, xmm2

        addps       xmm4, xmm5
        addps       xmm4, xmm6
        xorps       xmm4, SIMD_SP_signBit

        // transpose xmm0, xmm1, xmm2, xmm4 to memory
        movaps      xmm7, xmm0
        movaps      xmm5, xmm2

        unpcklps    xmm0, xmm1
        unpcklps    xmm2, xmm4

        movlps      [edx+eax-8*16+0], xmm0
        movlps      [edx+eax-8*16+8], xmm2

        movhps      [edx+eax-7*16+0], xmm0
        movhps      [edx+eax-7*16+8], xmm2

        unpckhps    xmm7, xmm1
        unpckhps    xmm5, xmm4

        movlps      [edx+eax-6*16+0], xmm7
        movlps      [edx+eax-6*16+8], xmm5
```

```
        movhps      [edx+eax-5*16+0], xmm7
        movhps      [edx+eax-5*16+8], xmm5

        jle         loopPlane4

    done4:

        sub         eax, 4*16
        jge         done

    loopPlane1:
        mov         ecx, [edi+0]
        shl         ecx, VERTEX_SIZE_SHIFT
        mov         ebx, [edi+4]
        shl         ebx, VERTEX_SIZE_SHIFT
        mov         edx, [edi+8]
        shl         edx, VERTEX_SIZE_SHIFT

        movss       xmm0, [esi+ebx+VERTEX_POSITION_OFFSET+0]
        subss       xmm0, [esi+ecx+VERTEX_POSITION_OFFSET+0]
        movss       xmm1, [esi+ebx+VERTEX_POSITION_OFFSET+4]
        subss       xmm1, [esi+ecx+VERTEX_POSITION_OFFSET+4]
        movss       xmm2, [esi+ebx+VERTEX_POSITION_OFFSET+8]
        subss       xmm2, [esi+ecx+VERTEX_POSITION_OFFSET+8]

        movss       xmm3, [esi+edx+VERTEX_POSITION_OFFSET+0]
        subss       xmm3, [esi+ecx+VERTEX_POSITION_OFFSET+0]
        movss       xmm4, [esi+edx+VERTEX_POSITION_OFFSET+4]
        subss       xmm4, [esi+ecx+VERTEX_POSITION_OFFSET+4]
        movss       xmm5, [esi+edx+VERTEX_POSITION_OFFSET+8]
        subss       xmm5, [esi+ecx+VERTEX_POSITION_OFFSET+8]

        movss       xmm6, xmm4
        mulss       xmm6, xmm2
        movss       xmm7, xmm5
        mulss       xmm7, xmm1
        subss       xmm6, xmm7

        add         edi, 1*12

        mulss       xmm5, xmm0
        mulss       xmm2, xmm3
        subss       xmm5, xmm2

        mulss       xmm3, xmm1
        mulss       xmm4, xmm0
        subss       xmm3, xmm4

        mov         edx, d

        movss       xmm0, xmm6
        mulss       xmm6, xmm6
        movss       xmm1, xmm5
        mulss       xmm5, xmm5
        movss       xmm2, xmm3
        mulss       xmm3, xmm3

        add         eax, 1*16

        addss       xmm3, xmm5
        addss       xmm3, xmm6
        rsqrtss     xmm3, xmm3

        mulss       xmm0, xmm3
        mulss       xmm1, xmm3
        mulss       xmm2, xmm3

        movss       [edx+eax-1*16+0], xmm0
        movss       [edx+eax-1*16+4], xmm1
        movss       [edx+eax-1*16+8], xmm2

        mulss       xmm0, [esi+ecx+VERTEX_POSITION_OFFSET+0]
        mulss       xmm1, [esi+ecx+VERTEX_POSITION_OFFSET+4]
        mulss       xmm2, [esi+ecx+VERTEX_POSITION_OFFSET+8]

        xorps       xmm0, SIMD_SP_firstSignBit
        subss       xmm0, xmm1
        subss       xmm0, xmm2
        movss       [edx+eax-1*16+12], xmm0

        jl          loopPlane1
```

```
    done:
        pop         ebx
    }
}
```