Shadow Volume Construction

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

Abstract

Optimized routines to create shadow volumes are presented. Conditionally executed code is replaced with instructions that are always executed and the Intel Streaming SIMD Extensions are used to exploit parallelism and minimize the number of executed instructions. The optimized routines are significantly faster than the implementation in C on a Pentium 4.

1. Introduction

Shadows are important in many applications because they add realism and help in understanding spatial relationships between objects. For instance shadows can put an animating character in place, making it much easier to see when the character is airborne or on the ground. The following two screenshots show a monster from the computer game DOOM III without and with shadows. Clearly it is much easier to determine the monsters is standing on the ground when the model casts shadows.

Advances in graphics hardware have made it possible to accurately render shadows from point and directional light sources in interactive applications. There are several different approaches to rendering real-time shadows on today's hardware. The more popular approaches use shadow maps, shadow volumes or a combination of these to define the regions in space that are in shadow.

Approaches based on shadow maps can render shadows from point lights, spot lights and directional lights for any kind of occluder geometry. First the distance from the light source of all objects that cast shadows is rendered onto a shadow map texture from the point of view of the light source. Next objects that receive shadows are rendered on-screen and the distance from a rendered pixel to the light source is compared to the value of the pixel in the shadow map texture that projects onto the rendered pixel to determine whether or not the rendered pixel is in shadow. Shadow mapping often suffers from aliasing because the projection transform used to map

shadow map pixels onto on-screen pixels changes the screen size of such pixels. As a result very large shadow maps and filtering have to be used to achieve good quality.

The approaches based on shadow volumes define the regions in space that are in shadow of an occluder in object space with additional geometry. Shadow volumes can be constructed for point lights, spot lights and directional light sources and always produce pixel-accurate but hard shadows. Approaches using shadow volumes cannot deal with objects that have no polygonal structure such as alpha-tested or displacement mapped geometry. Shadow volumes are typically constructed on the CPU which can be expensive. In this article the Intel Streaming SIMD Extensions are used to optimize the construction of shadow volumes on the CPU.

Several methods have been proposed to create shadow volumes entirely in vertex programs on today's graphics hardware [13,17,18,19]. However, as Kilgard [16] points out, computing silhouette edges within a vertex program may not improve performance if the occluders have high triangle counts or if there are a lot of shadow casting light sources. The reason is the need to push many more vertices into the pipeline that all have to go through the silhouette edge determination within the vertex program [19]. As a result occluders with high triangle counts generate large amounts of wasted vertices (degenerate triangles), and the cost of testing all the extra vertices may very well exceed the CPU and geometry upload savings. As more light sources interact with the geometry the vertex program costs accumulate rapidly because intermediate results and common calculations cannot be saved and/or shared on today's graphics hardware.

1.1 Previous Work

Shadow volumes were introduced by Frank Crow [1]. Bergeron [2] generalized shadow volumes for non-manifold objects and non-planar polygons. BSP trees have been used to accelerate shadow volume computation [5], but they do not work well with moving lights or dynamic objects.

One of the first implementations using graphics hardware to render shadow volumes was demonstrated in Pixel-Planes [4]. Heidmann [6] implemented Crow's algorithm using the stencil buffer. This approach is known as the z-pass method and can produce incorrect results when the viewport cuts through a shadow volume. Diefenbach [7] presented capping methods, but these are not completely robust. To overcome these problems several researchers have proposed z-fail testing, also known as "Carmack's Reverse", for shadow volume rendering [9,10,11,12]. Brennan [13] presented methods for constructing shadow volumes entirely in vertex programs. Brabec and Seidel [17] described an algorithm for fast shadow volume computation using graphics hardware for silhouette edge determination where geometry is encoded as colors. McGuire and Hughes [19] described how to find silhouettes and extrude them into shadow volume sides entirely in a vertex program using a specially precomputed mesh.

1.2 Layout

Section 2 describes some details of shadow volumes. Section 3 describes the basic algorithm used to create a shadow volume for an arbitrary triangle mesh. Sections 4

through 7 describe how the different parts of the algorithm can be optimized using the Intel Streaming SIMD Extensions. The results of the optimizations are presented in section 8 and several conclusions are drawn in section 9.

2. Shadow Volumes

Shadow volumes can be constructed for point lights, spot lights and directional light sources and always produce pixel-accurate but hard shadows. A shadow volume defines the regions in space that are in shadow of an occluder in object space with additional geometry. This geometry is a regular mesh that is not visible but rendered into a separate buffer, usually a stencil buffer. The shadow volume geometry is derived from the occluder geometry given a light source. It is typically hard or not possible to construct shadow volumes for occluders that have no polygonal structure such as alpha-tested or displacement mapped geometry.

The algorithm presented here assumes occluder triangles that face away from the light source cast shadows. The shadow volume sides are constructed from the extrusion of silhouette edges of the occluder geometry. Such silhouette edges are the boundaries between lit and unlit triangles. The shadow volume is capped on one end by the triangles facing away from the light source. On the other end the shadow volume is capped by another copy of the same triangles but these triangles are projected away from the light source to infinity. To properly render the shadow volume the vertices of the shadow volume triangles must consistently wind counterclockwise so that the triangle normals point out of the shadow volume.

To determine the regions in space that are in shadow of an occluder the stencil buffer is first cleared to all zeros. The shadow volume for the occluder is then rendered to the stencil buffer with an appropriate depth test. Front facing shadow volume triangles increment and back facing triangles decrement the stencil buffer pixels. Pixels with a stencil buffer value unequal zero are now considered in shadow.

Not all the shadow volume geometry needs to be drawn at all times. The shadow volume caps can be culled and often completely omitted [20]. Special hardware features like two-sided stencil testing, scissor rectangles and depth clamping can be used to further improve the performance of shadow volume rendering [12].

3. Creating Shadow Volumes

The routine presented here creates a shadow volume for an arbitrary triangle surface. The shadow volume is created as a list with vertex indices. The actual vertices of the shadow volume are not copied and/or extruded in this routine.

Transferring vertex data is avoided by using a double length vertex buffer and a vertex program to perform the shadow volume extrusion in hardware. All the vertices are duplicated in this double length vertex buffer and it is usually more efficient to use a separate vertex buffer for shadow volume rendering with vertices that only store positions. Another vertex buffer with vertices described with texture coordinates,

normals, tangents etc. is used for rendering the visual representation of the surface. The vertex buffer for rendering shadow volumes contains two consecutive copies of each vertex with the positions stored as homogenous coordinates. The vertices at even positions in this vertex buffer are of the form (x,y,z,1) and the vertices at the odd positions are of the form (x,y,z,0). The light position is subtracted from the (x,y,z) of the vertices at the odd positions in the vertex program which allows all shadow volumes for a single surface to use the same vertex buffer. When the appropriate hardware is not available unique vertex buffers have to be used for each shadow volume and the light origin is subtracted from the vertices at odd positions on the CPU. Because the 'w' component of the homogenous coordinates for the vertices at the odd positions is zero these vertices are projected to infinity and not clipped by the far clipping plane when projected with an appropriate projection matrix [12].

The routine presented here creates indices for the shadow volume triangles to be used with a vertex buffer as described above. The code for this routine is listed below. The occluder triangle surface is specified as an array with indices. For each triangle this array contains three elements with the numbers of the vertices that create the triangle. To construct the shadow volume sides additional connectivity information of the occluder triangles is required. For this purpose an array with SilEdge objects is passed into the routine below. This array has a SilEdge object for every triangle edge in the surface that may potentially become a silhouette edge. Such an SilEdge object stores two triangle/plane numbers and the indices of two vertices that define an edge. It is important to construct the SilEdge objects with consistent edge orientations to make sure the shadow volume side triangles have the correct winding directions. The vertices of the shadow volume triangles must wind counterclockwise so that the surface normal points out of the shadow volume. The SilEdge object is defined in code as follows.

```
struct SilEdge {
    int p1, p2;     // triangles/planes defining the edge
    int v1, v2;     // vertices defining the edge
};
```

To make sure shadow volumes with correct winding orders are created the convention is used in which the edge from SilEdge::v1 to SilEdge::v2 is counterclockwise in the triangle SilEdge::p1 and clockwise in the triangle SilEdge::p2. Because a double size vertex buffer is used for the shadow volumes the vertex numbers in the SilEdge objects are multiplied by two so they point at the even vertices in the vertex buffer. The array with SilEdge objects is setup once for a surface and can be used to construct all shadow volumes.

The 'CreateShadowVolume' routine listed below takes a 'facing' parameter which is an array with 'number of triangles plus one' bytes. A byte in this array is set to one if the associated triangle faces the light source and set to zero if the triangle faces away from the light source. The last byte in the array is not associated with any particular triangle but is used for dangling edges. Edges in the surface that are used by only a single triangle have a SilEdge object defined for them with the second plane number pointing at this last byte in the 'facing' array. Because this last byte is always set to one a dangling edge will contribute to the shadow silhouette of the surface when a triangle with the dangling edge faces away from the light source.

Whether or not a triangle faces the light source can be calculated by testing if the light source is at the front side of the triangle plane. If ax + by + cz + d = 0 is the triangle plane equation then the facing information is calculated as shown in appendix A. In this routine the (light.x, light.y, light.z, light.w) is the light origin for spot lights or the light direction for directional lights. The triangle plane equations are calculated once per surface and are only recalculated when the vertex positions of the occluder triangles change for instance when the surface animates. All lights interacting with the surface can use the same plane equations to derive facing information.

The 'CreateShadowVolume' routine listed below also takes a 'cullBits' parameter which is an array with a byte for each vertex with culling information. A bit in such a byte tells whether or not the vertex is at the right side of one of the bounding planes of the light volume. Typically only 6 bits of each byte are used for lights with 6 bounding planes. If any of these 6 bits is set to one the vertex is outside the light volume. The 'cullBits' parameter can be null if no culling information is available or if the complete surface is inside the light volume. Usually the bounding volume of the surface can be quickly tested against the bounding planes of the light volume. The surface is often completely in front of at least several of the light volume bounding planes and all the bits for such bounding planes are set to zero in the 'cullBits' array. The routine to calculate the cull buts is listed in appendix B.

Whether or not the culling information should be calculated and used depends on the kind of lights and surfaces that interact with each other. For small surfaces in large light volumes the culling information is not used when the surface is completely inside the light volume. However, the culling information should be used for small lights interacting with very large surfaces. For instance the headlights of a car driving over terrain typically only interact with a few triangles of the terrain mesh. The terrain triangles that do not interact with the light volume should be culled to reduce the number of shadow volume triangles and to save fill rate.

```
int CreateShadowVolume( int *shadowIndices, const int *indices, int numIndices, const SilEdge *silEdges, int
numSilEdges, byte *facing, const byte *cullBits ) {
    int numShadowingTriangles, numSilhouetteIndices, numCapIndices;
    int numTriangles = numIndices / 3;

    if ( cullBits == NULL ) {
        // count the number of shadowing triangles
        numShadowingTriangles = numTriangles - CountFacing( facing, numTriangles );
    } else {
        // count the number of shadowing triangles and make all triangles that are outside the light frustum
        // "facing" so they won't cast shadows
        numShadowingTriangles = numTriangles - CountFacingCull( facing, numTriangles, indices, cullBits );
    }

    if ( !numShadowingTriangles ) {
        // no triangles are inside the light frustum and still facing the right way
        return 0;
    }

    // create triangles along silhouette planes
    numSilhouetteIndices = CreateSilTriangles( shadowIndices, facing, silEdges, numSilEdges );

    // put some triangles on the model and some on the distant projection
    numCapIndices = CreateCapTriangles( shadowIndices + numSilhouetteIndices, facing, indices, numIndices );

    return numSilhouetteIndices + numCapIndices;
}
```

The routine listed above calls four functions to do the actual work. The first two functions count the number of facing triangles. The second function can also modify the facing of some triangles based on the culling information. When a triangle is found to be completely off to one side of one of the light volume bounding planes the triangle is assumed facing and will not cast shadows. The last two functions that are called create the actual indices for the shadow volume silhouette triangles and cap triangles respectively. The following sections describe how these four functions can be optimized using the Intel Streaming SIMD Extensions. The routines to calculate the facing of triangles and the cull bits of vertices that are listed in appendix A and appendix B respectively can also be optimized using the Intel Streaming SIMD Extensions. However, these optimizations are straight forward and the assembler code is ommited here.

4. Counting Facing Triangles

The function that counts the number of facing triangles can be implemented with a simple loop in C as shown below.

```
int CountFacing( const byte *facing, const int numTriangles ) {
    int i, n;

    n = 0;
    for ( i = 0; i < numTriangles; i++ ) {
        if ( facing[i] ) {
            n++;
        }
    }
    return n;
}
```

Because a facing byte is always either one or zero there is no need to test whether or not the facing byte is unequal zero in order to conditionally increment a counter. Instead the facing bytes can simply be added together to avoid any conditional branches.

```
n += facing[i];
```

This function can be further optimized in C by unrolling the loop several times and removing the dependencies between the consecutive statements.

```
n0 += facing[i+0];
n1 += facing[i+1];
n2 += facing[i+2];
n3 += facing[i+3];
```

Here n0, n1, n2, n3 are four separate independent integer counters that are accumulated after the loop as follows.

```
n = n0 + n1 + n2 + n3;
```

Using the Intel Streaming SIMD Extensions the loop can be unrolled many more times. The SSE2 instruction set allows 16 unsigned bytes to be added in parallel with a single 'paddusb' instruction. The SSE2 instruction 'movdqa' can be used to load 16 facing bytes. If more than 256 facing bytes are added the bytes could overflow in case all the facing bytes happen to be set to one. Therefore the bytes have to be converted to word or double word integers. The instructions 'punpcklbw', 'punpckhbw', 'punpcklwd' and 'punpckhwd' can be used for this purpose. The following code shows how 16 unsigned

bytes in the 'xmm0' register are added together to form 4 double word integers in the same register.

```
movdqa      xmm1, xmm0
punpcklbw   xmm0, xmm7
punpckhbw   xmm1, xmm7
paddusw     xmm0, xmm1
movdqa      xmm1, xmm0
punpcklwd   xmm0, xmm7
punpckhwd   xmm1, xmm7
paddd       xmm0, xmm1
```

In the above code the 'xmm7' register contains all zeros. The 'punpcklbw' instruction is used to interleave the low-order bytes of the two operands. Because the 'xmm7' register contains all zeros the low-order bytes are interleaved with zeros and effectively zero extended to words. In the same way the high-order bytes are zero extended to words. The two registers with 8 words each are now added together and the result is once more zero extended to double words with the 'punpcklwd' and 'punpckhwd' instructions.

The complete optimized routine is listed in appendix C. The routine has four loops. The first loop adds 256 facing bytes per iteration, the second loop 16, the third loop 4 and finally the last loop adds any remaining facing bytes. The routine listed in appendix C assumes the array with facing bytes is aligned on a 16 byte boundary.

5. Counting Facing Triangles With Culling

The second function used to create shadow volumes can be implemented in C as follows.

```
int CountFacingCull( byte *facing, const int numTriangles, const int *indices, const byte *cullBits ) {
    int i, n;

    n = 0;
    for ( i = 0; i < numTriangles; i++ ) {
        if ( !facing[i] ) {
            int i1 = indices[0];
            int i2 = indices[1];
            int i3 = indices[2];
            if ( cullBits[i1] & cullBits[i2] & cullBits[i3] ) {
                facing[i] = 1;
                n++;
            }
        } else {
            n++;
        }
        indices += 3;
    }
    return n;
}
```

This function counts the number of triangles that face the light source just like the function described in the previous section. However, based on the cull bits of the vertices, this function also culls triangles that are outside the light volume by setting the facing byte for such triangles to one. If a triangle is completely off to one side of one of the light volume bounding planes the bitwise logical 'and' of the cull bits for the vertices will have a bit set to one and will as such be unequal zero. If the bitwise logical 'and' of the cull bits is unequal zero the facing byte for the triangle is set to one.

The function uses several conditional branches and there are several sections with conditionally executed code. The conditional branches are often hard to predict because the facing of triangles may change erratically while a surface animates. A surface may also interact with many different light sources at different positions for which the facing triangles are completely different. These hard to predict conditional branches typically result in numerous mispredictions and significant penalties on today's CPUs that implement a deep pipeline [22,23]. When a branch is mispredicted, the misprediction penalty is typically the depth of the pipeline.

As such the routine can be optimized in C by replacing the poorly predictable branches with some bit manipulation.

```
int c = cullBits[indices[0]] & cullBits[indices[1]] & cullBits[indices[2]];
facing[i] |= ( (-c) >> 31 ) & 1;
n += facing[i];
```

First the bitwise logical 'and' of the cull bits for the vertices is calculated and stored in the variable 'c'. If the triangle is completely off to one side of one of the light volume bounding planes the variable 'c' will have a bit set to one and will as such be unequal zero. By negating the variable 'c' the 31st bit will be set if and only if the variable is unequal zero. This 31st bit is then shifted to bit position zero and a bitwise logical 'and' with one is used to make sure no other bits are set. A bitwise logical 'or' is used to set the facing of the triangle to one if the triangle is outside the light volume. Next the triangle facing byte is added to the facing counter just like in the routine described in the previous section.

The routine can be further optimized using the Intel Streaming SIMD Extensions. Because the triangles may reference arbitrary vertices the cull bits for a single triangle can be scattered in memory which forces the cull bits to be loaded individually. The loop is unrolled four times and the bitwise logical 'and' of the cull bits for four triangles are stored in the lower double word of the SSE register 'xmm0'. The 'pinsrw' instruction is used to quickly insert the cull bits for the last two triangles into the SSE register. Instead of negating the cull bits and shifting the 31st bit, the 'pcmpgtb' instruction is used to compare the cull bits with zero and directly set each byte to either all zeros or all ones. A bitwise logical 'and' with bytes set to one is used to make sure all bits of each byte except the first are always set to zero. Four facing bytes are loaded and a bitwise logical 'or' is used to set any triangles outside the light volume to facing. The new facing bytes are stored back to memory and accumulated to count the total number of facing triangles. To accumulate the facing bytes they are zero extended to double words using the 'punpcklbw' and 'punpcklwd' instructions.

The complete optimized routine is listed in appendix D. The routine assumes no alignment for the arrays but for optimal performance the arrays should be at least aligned on a 4 byte boundary.

6. Creating Silhouette Triangles

The following function can be used to setup the triangle indices for the shadow volume sides.

```
int CreateSilTriangles( int *shadowIndices, const byte *facing, const SilEdge *silEdges, const int numSilEdges )
{
    int i;
    const silEdge_t *sil;
    int *si;

    si = shadowIndices;
    for ( sil = silEdges, i = numSilEdges; i > 0; i--, sil++ ) {

        byte f1 = facing[sil->p1];
        byte f2 = facing[sil->p2];

        if ( f1 != f2 ) {

            int v1 = sil->v1;
            int v2 = sil->v2;

            if ( f1 ) {
                si[0] = v1;
                si[1] = v2 + 1;
                si[2] = v2;
                si[3] = v1;
                si[4] = v1 + 1;
                si[5] = v2 + 1;
            } else {
                si[0] = v1;
                si[1] = v2;
                si[2] = v2 + 1;
                si[3] = v1 + 1;
                si[4] = v1;
                si[5] = v2 + 1;
            }
            si += 6;
        }
    }
    return si - shadowIndices;
}
```

The function loops over the array with SilEdge objects of a surface. For each potential silhouette edge the routine compares the facing of the two triangles that share the edge. If one of the triangles faces the light source and the other does not then the edge is part of the shadow silhouette. For each shadow silhouette edge the routine creates two triangles that represent the extruded edge. To properly determine the regions in space that are in shadow the vertices of the shadow volume triangles must consistently wind counterclockwise so that the triangle normals point out of the shadow volume. As such the triangle winding orders have to be set based on which of the two triangles faces the light source.

Just like the function in the previous section this function uses several hard to predict conditional branches. Mispredictions and significant penalties are the result. However, the triangle winding order can be set based on facing without using a poorly predictable branch as shown below.

```
si[0] = v1;
si[1] = v2 ^ f1;
si[2] = v2 ^ f2;
si[3] = v1 ^ f2;
si[4] = v1 ^ f1;
si[5] = v2 ^ 1;
```

It may not be immediately apparent the above code does the right thing. However, as described in section 2 the SilEdge vertex numbers are multiplied with two and always even. As such flipping the last bit of these vertex numbers is equivalent to adding one. Furthermore if 'f1' equals one then 'f2' always equals zero and vice versa because only one of the two triangles faces the light source.

The first conditional branch in the function above can be avoided by always writing out the silhouette triangles but only updating the shadow volume index pointer when one triangle faces the light source and the other does not.

```
si += 6 * ( f1 ^ f2 );
```

Changing the C code like this does not necessarily improve the performance. Even through the misprediction penalties are avoided the function will burn through all instructions in all cases. However, the Intel Streaming SIMD Extensions can be used to minimize the number of instructions and dependencies such that avoiding the misprediction penalties more than makes up for the additional instructions that would not have been executed in the former case of a properly predicted branch for an edge that is not part of the shadow silhouette.

In the optimized routine the loop is unrolled four times. The facing bytes are loaded individually and moved to SSE registers. The SilEdge vertex numbers are loaded into an SSE register with a single 'movq' instruction. Two shuffle and two bitwise logical exclusive-or instructions are used to spread, and increment the vertex numbers based on the facing of the triangles. In the C code above a bitwise logical exclusive-or with a constant of one is used to increment one of the vertex numbers. However, the optimized routine makes use of the fact that only ever one of the triangles faces the light source for a shadow silhouette edge. As such only one of the two facing bytes 'f1' and 'f2' is set to one which makes (v1 ^ f1 ^ f2) equivalent to (v1 ^ 1). The shadow index pointer is incremented based on the facing of the triangles and the pointer is moved between two general purpose registers to minimize the dependencies.

The complete optimized routine is listed in appendix E. The routine assumes the arrays with indices are aligned on a 16 byte boundary.

7. Creating Cap Triangles

The following function can be used to setup the triangle indices for the shadow volume caps.

```
int CreateCapTriangles( int *shadowIndices, const byte *facing, const int *indices, const int numIndices ) {
    int i, j;
    int *si;

    si = shadowIndices;
    for ( i = 0, j = 0; i < numIndices; i += 3, j++ ) {
        if ( facing[j] ) {
            continue;
        }

        int i0 = indices[i+0] * 2;
        int i1 = indices[i+1] * 2;
        int i2 = indices[i+2] * 2;

        si[0] = i2;
        si[1] = i1;
        si[2] = i0;

        si[3] = i0 + 1;
        si[4] = i1 + 1;
        si[5] = i2 + 1;

        si += 6;
    }
    return si - shadowIndices;
}
```

The same strategy as used for the routine in the previous section can be used to optimize the above function. The loop is unrolled 4 times to allow the indices for four triangles to be loaded with three 'movdqa' instructions. The triangle indices are rearranged with several shuffle instructions and a bitwise logical exclusive-or is used to increment some of the triangle indices. The shadow index pointer is incremented based on the facing of the triangles and the pointer is moved between two general purpose registers to minimize the dependencies.

The complete optimized routine is listed in appendix F. The routine assumes the arrays with indices are aligned on a 16 byte boundary.

8. 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 created shadow volumes for a realistic player character model with 1344 triangles and 2016 potential silhouette edges (SilEdge objects). Note that the number of potential silhouette edges is the maximum number of edges of a two-manifold mesh (2016 = 1344 * 3 / 2). Furthermore 50% of the triangles face the light source and 20% of the potential silhouette edges are part of the shadow silhouette. Different models may have different triangle counts but these ratios are typically similar for human-like characters.

The total number of clock cycles and the number of clock cycles per triangle or silhouette edge 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**
CountFacing (C)	22548	17	22688	17
CountFacing (SSE)	348	0.3	383	0.3
CountFacingCull (C)	24124	18	29835	22
CountFacingCull (SSE)	10848	8	13883	10
CreateSilTriangles (C)	32314	16	44983	22
CreateSilTriangles (SSE)	32292	16	36901	18
CreateCapTriangles (C)	23780	18	33848	25
CreateCapTriangles (SSE)	12080	9	14205	11
CreateShadowVolume, no culling (C)	79042	59	112443	83
CreateShadowVolume, no culling (SSE)	44924	33	51503	38
CreateShadowVolume, with culling (C)	80625	60	119590	89
CreateShadowVolume, with culling (SSE)	55648	41	66660	50

9. Conclusion

The conventional algorithms to create shadow volumes on the CPU as presented in literature typically use a lot of conditional branches and conditionally executed code. These conditional branches are often hard to predict because the facing of triangles may change erratically while a surface animates. A surface may also interact with many different light sources at different positions for which the facing triangles are completely different. These hard to predict conditional branches typically result in numerous mispredictions and significant penalties on today's CPUs that implement a deep pipeline like the Pentium 4. When a branch is mispredicted, the misprediction penalty is typically the depth of the pipeline.

The conditionally executed code can be replaced with instructions that are always executed. The optimized algorithm always burns through all instructions in all cases but the number of executed instructions is minimized using the Intel Streaming SIMD Extensions. Furthermore the instruction dependencies are minimized to exploit maximum parallelism through the dynamic execution engine of the Pentium 4. As a result the optimized algorithm is significantly faster while it does not suffer from penalties due to mispredicted branches.

10. References

1. Shadow Algorithms for Computer Graphics
   Frank Crow
   Computer Graphics, Vol. 11, No.3, Proceedings of SIGGRAPH 1977, July 1977
   Available Online: http://doi.acm.org/10.1145/563858.563901

2. Shadow volumes for non-planar polygons
   P. Bergeron
   In Graphics Interface '85 Proceedings, pp. 417-418, 1985

3. A General Version of Crow's Shadow Volumes
   P. Bergeron
   In IEEE Computer Graphics and Applications, 6(9), 17-28, 1986

4. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes.
   H. Fuchs, J. Goldfeather, J. P. HultQuist, S. Spach, J. D. Austin, JR. F. P. Brooks, J. G. Eyles, J. Poulton
   In Computer Graphics (SIGGRAPH '85 Proceedings), vol. 19, pp. 111-120, 1985

5. Near real-time shadow generation using BSP trees
   N. Chin, S. Feiner
   In Computer Graphics (SIGGRAPH '89 Proceedings), vol. 23, pp. 99-106, 1989
   Available Online: http://doi.acm.org/10.1145/74333.74343

6. Real Shadows Real Time
   Tim Heidmann
   IRIS Universe, 18 pp. 23-31, November 1991
   Available Online: http://developer.nvidia.com/object/robust_shadow_volumes.html

7. Multi-pass pipeline rendering: Interaction and realism through hardware provisions.
   Paul Joseph Diefenbach
   PhD thesis, University of Pennsylvania, 1996.

8. Shadow Volume Reconstruction from Depth Maps
   Michael D. McCool
   University of Waterloo, Waterloo, Ontario, Canada
   ACM Transactions on Graphics (TOG), Volume 19, Issue 1, pp. 1-26 (January 2000)
   Available Online: http://doi.acm.org/10.1145/343002.343006

9. Using the Stencil Buffer
   Sim Dietrich
   GDC, March 1999

10. Real Time Shadows
    Bill Bilodeau, Mike Songy
    Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England, May 1999.

11. On Shadow Volumes
    John Carmack
    id Software, May 2000
    Available Online: http://developer.nvidia.com/object/robust_shadow_volumes.html

12. 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

13. Shadow Volume Extrusion Using a Vertex Shader
    Criss Brennan
    ShaderX, Wolfgang Engel (editor), May 2002
    Available Online: http://www.shaderx.com/

14. Computing Optimized Shadow Volumes
    Alex Vlachos, Drew Card
    Game Programm Gems 3, 2002
    Available Online: http://www.GameProgrammingGems.com

15. Fast, Practical and Robust Shadows
    Morgan McGuire, John F. Hughes, Kevin Egan, Mark J. Kilgard, Cass Everitt
    nVidia Corporation, 2003
    Available Online: http://developer.nvidia.com/object/fast_shadow_volumes.html

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

17. Shadow Volumes on Programmable Graphics Hardware
    S. Brabec, H. Siedel
    Eurographics, 2003

18. GPU Shadow Volume Construction for Nonclosed Meshes
    Warrick Buchanan
    Game Programming Gems 4, 2003
    Available Online: http://www.GameProgrammingGems.com

19. Hardware Determined Edge Features
    Originally titled "NPR on Programmable Hardware"
    Morgan McGuire and John F. Hughes.
    Proceedings of the Non-Photorealistic Animation and Rendering 2004 (NPAR '04),
    Annecy, France, June 7-9, 2004.
    Available Online: http://www.cs.brown.edu/people/morgan/

20. GPU Gems - 9. Efficient Shadow Volume Rendering
    Morgan McGuire (Brown University)
    Randima Fernando (editor)
    Addison-Wesley, 2004

21. CC Shadow Volumes
    Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, Dinesh Manocha
    University of North Carolina at Chapel Hill, 2004
    Eurographics Symposium on Rendering, 2004
    Available Online: http://gamma.cs.unc.edu/ccsv/

22. Avoiding the Cost of Branch Misprediction
    Rajiv Kapoor
    Intel, December 2002
    Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/19952.htm

23. Branch and Loop Reorganization to Prevent Mispredicts
    Jeff Andrews

Intel, January 2004
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/optimization/66779.htm

Appendix A

```
/*
    Calculating Triangle Facing
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

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

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

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

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

void CalculateFacing( const Plane *planes, const int numTriangles, const Vec4 &light, byte *facing ) {
    int i;

    for ( i = 0; i < numTriangles; i++ ) {
        facing[i] = planes[i].a * light.x +
                    planes[i].b * light.y +
                    planes[i].c * light.z +
                    planes[i].d * light.w > 0.0f;
    }
    facing[numTriangles] = 1;   // for dangling edges to reference
}
```

Appendix B

```
/*
    Calculating Vertex Cull Bits
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

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

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

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

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

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

```
struct Bounds {
    Vec4    center;
    Vec4    extents;
};

const float CULL_EPSILON        = 0.0f;
const int NUM_LIGHT_PLANES      = 6;

bool CalculateCullBits( const Vertex *verts, const int numVerts, const Bounds &surfaceBounds, const Plane
lightPlanes[NUM_LIGHT_PLANES], byte *cullBits ) {
    int i, j, frontBits;

    assert( NUM_LIGHT_PLANES <= sizeof( cullBits[0] ) * 8 );

    frontBits = 0;

    // cull the triangle surface bounding box
    for ( i = 0; i < NUM_LIGHT_PLANES; i++ ) {
        const Plane &plane = lightPlanes[i];
        float d1 = plane.a * surfaceBounds.center.x +
                        plane.b * surfaceBounds.center.y +
                            plane.c * surfaceBounds.center.z +
                                plane.d;
        float d2 = fabs( plane.a * surfaceBounds.extents.x ) +
                        fabs( plane.b * surfaceBounds.extents.y ) +
                            fabs( plane.c * surfaceBounds.extents.z );

        if ( d1 - d2 >= CULL_EPSILON ) {
            frontBits |= 1 << i;    // front bits for the whole surface
        }
    }

    // if the surface is completely inside the light frustum
    if ( frontBits == ( ( 1 << NUM_LIGHT_PLANES ) - 1 ) ) {
        return true;    // return true if completely inside
    }

    memset( cullBits, 0, numVerts * sizeof( cullBits[0] ) );

    for ( i = 0; i < NUM_LIGHT_PLANES; i++ ) {
        // if completely infront of this clipping plane
        if ( frontBits & ( 1 << i ) ) {
            continue;
        }
        const Plane &plane = lightPlanes[i];
        for ( j = 0; j < numVerts; j++ ) {
            int bit = plane.a * verts[j].position.x +
                        plane.b * verts[j].position.y +
                            plane.c * verts[j].position.z +
                                plane.d < CULL_EPSILON;
            cullBits[j] |= bit << i;
        }
    }
    return false;       // return false if not completely inside
}
```

Appendix C

```
/*
    SSE Optimized Counting of Facing Triangles
    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 ALIGN4_INIT4( X, I0, I1, I2, I3 )   ALIGN16( static X[4] ) = { I0, I1, I2, I3 }
#define ALIGN16_INIT1( X, I0 )              ALIGN16( static X[16] ) = { I0, I0, I0, I0, I0, I0, I0, I0, I0, I0,
I0, I0, I0, I0, I0, I0 }
#define R_SHUFFLE_D( x, y, z, w )         (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | ( (y) & 3 ) << 2 | ( (x) & 3
))

int CountFacing( const byte *facing, const int numTriangles ) {
    ALIGN16( int n[4]; )

    __asm {

        mov         eax, numTriangles
        mov         edi, facing
        test        eax, eax
        jz          done

        pxor        xmm6, xmm6
        pxor        xmm7, xmm7

        sub         eax, 256
        jl          run16

    loop256:
        movdqa      xmm0, [edi+ 0*16]
        movdqa      xmm1, [edi+ 1*16]
        movdqa      xmm2, [edi+ 2*16]
        movdqa      xmm3, [edi+ 3*16]
        paddusb     xmm0, [edi+ 4*16]
        paddusb     xmm1, [edi+ 5*16]
        paddusb     xmm2, [edi+ 6*16]
        paddusb     xmm3, [edi+ 7*16]
        paddusb     xmm0, [edi+ 8*16]
        paddusb     xmm1, [edi+ 9*16]
        paddusb     xmm2, [edi+10*16]
        paddusb     xmm3, [edi+11*16]
        paddusb     xmm0, [edi+12*16]
        paddusb     xmm1, [edi+13*16]
        paddusb     xmm2, [edi+14*16]
        paddusb     xmm3, [edi+15*16]
        paddusb     xmm0, xmm1
        paddusb     xmm2, xmm3
        paddusb     xmm0, xmm2

        add         edi, 256
        sub         eax, 256

        movdqa      xmm1, xmm0
        punpcklbw   xmm0, xmm7
        punpckhbw   xmm1, xmm7
        paddusw     xmm0, xmm1
        movdqa      xmm1, xmm0
        punpcklwd   xmm0, xmm7
        punpckhwd   xmm1, xmm7
        paddd       xmm0, xmm1
        paddd       xmm6, xmm0

        jge         loop256

    run16:
        pxor        xmm0, xmm0
        add         eax, 256 - 16
        jl          run4

    loop16:
        paddusb     xmm0, [edi]
        add         edi, 16
        sub         eax, 16
        jge         loop16

    run4:
        add         eax, 16 - 4
        jl          run1

        pxor        xmm1, xmm1

    loop4:
        movd        xmm1, [edi]
        paddusb     xmm0, xmm1
        add         edi, 4
        sub         eax, 4
        jge         loop4
```

```
    run1:
        movdqa      xmm1, xmm0
        punpcklbw   xmm0, xmm7
        punpckhbw   xmm1, xmm7
        paddusw     xmm0, xmm1
        movdqa      xmm1, xmm0
        punpcklwd   xmm0, xmm7
        punpckhwd   xmm1, xmm7
        paddd       xmm0, xmm1
        paddd       xmm6, xmm0

        movdqa      n, xmm6
        add         eax, 4-1
        jl          done

        mov         edx, dword ptr n

    loop1:
        movzx       ecx, [edi]
        add         edx, ecx
        add         edi, 1
        sub         eax, 1
        jge         loop1

        mov         dword ptr n, edx

    done:
    }

    return n[0] + n[1] + n[2] + n[3];
}
```

Appendix D

```
/*
    SSE Optimized Culling and Counting of Facing Triangles
    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.
*/

ALIGN16_INIT1( unsigned char SIMD_B_one, 1 );

int CountFacingCull( byte *facing, const int numTriangles, int *indices, const byte *cullBits ) {
    ALIGN16( int n[4]; )

    __asm {
        push        ebx
        mov         eax, numFaces
        mov         esi, indices
        mov         edi, cull
        mov         ebx, facing
        test        eax, eax
        jz          done
        add         ebx, eax
        neg         eax

        pxor        xmm5, xmm5
        pxor        xmm6, xmm6
        movdqa      xmm7, SIMD_B_one

        add         eax, 4
        jg          run1

    loop4:

        mov         ecx, dword ptr [esi+0*4]
        movzx       edx, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+1*4]
```

```
        and         dl, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+2*4]
        and         dl, byte ptr [edi+ecx]

        mov         ecx, dword ptr [esi+3*4]
        mov         dh, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+4*4]
        and         dh, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+5*4]
        and         dh, byte ptr [edi+ecx]
        movd        xmm0, edx

        mov         ecx, dword ptr [esi+6*4]
        movzx       edx, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+7*4]
        and         dl, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+8*4]
        and         dl, byte ptr [edi+ecx]

        mov         ecx, dword ptr [esi+9*4]
        mov         dh, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+10*4]
        and         dh, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+11*4]
        and         dh, byte ptr [edi+ecx]
        pinsrw      xmm0, edx, 1

        add         esi, 12*4

        movd        xmm1, [ebx+eax-4]
        pcmpgtb     xmm0, xmm6
        pand        xmm0, xmm7
        por         xmm1, xmm0
        movd        [ebx+eax-4], xmm1

        add         eax, 4

        punpcklbw   xmm1, xmm6
        punpcklwd   xmm1, xmm6
        paddd       xmm5, xmm1

        jle         loop4

    run1:
        sub         eax, 4
        jge         done

    loop1:
        mov         ecx, dword ptr [esi+0*4]
        movzx       edx, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+1*4]
        and         dl, byte ptr [edi+ecx]
        mov         ecx, dword ptr [esi+2*4]
        and         dl, byte ptr [edi+ecx]

        neg         edx
        shr         edx, 31
        movzx       ecx, byte ptr [ebx+eax]
        or          ecx, edx
        mov         byte ptr [ebx+eax], cl
        movd        xmm0, ecx
        paddd       xmm5, xmm0

        add         esi, 3*4
        add         eax, 1
        jl          loop1

    done:
        pop         ebx
        movdqa      dword ptr n, xmm5
    }

    return n[0] + n[1] + n[2] + n[3];
}
```

Appendix E

```
/*
    SSE Optimized Construction of Shadow Volume Silhouette Triangles
    Copyright (C) 2005 Id Software, Inc.
    Written by J.M.P. van Waveren

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

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

struct SilEdge {
    int p1, p2;     // planes defining the edge
    int v1, v2;     // verts defining the edge
};

int CreateSilTriangles( int *shadowIndices, const byte *facing, const SilEdge *silEdges, const int numSilEdges )
{
    int num;

    __asm {
        push        ebx
        mov         eax, numSilEdges
        mov         ebx, shadowIndexes
        mov         esi, facing
        mov         edi, silEdges
        shl         eax, 4
        jz          done
        add         edi, eax
        neg         eax
        shr         ebx, 3

        add         eax, 4*16
        jg          run1

    loop4:
        mov         ecx, dword ptr [edi+eax-4*16+0]
        movzx       ecx, byte ptr [esi+ecx]
        movd        xmm2, ecx
        mov         edx, dword ptr [edi+eax-4*16+4]
        movzx       edx, byte ptr [esi+edx]
        pinsrw      xmm2, edx, 2
        movq        xmm0, qword ptr [edi+eax-4*16+8]
        pshufd      xmm1, xmm2, R_SHUFFLE_D( 2, 0, 1, 1 )
        xor         ecx, edx
        pshufd      xmm0, xmm0, R_SHUFFLE_D( 0, 1, 1, 0 )
        lea         edx, [ecx*2+ecx]
        pxor        xmm0, xmm1
        add         edx, ebx
        movlps      qword ptr [ebx*8+0*4], xmm0
        pxor        xmm2, xmm0
        movhps      qword ptr [ebx*8+2*4], xmm0
        movlps      qword ptr [ebx*8+4*4], xmm2

        mov         ecx, dword ptr [edi+eax-3*16+0]
        movzx       ecx, byte ptr [esi+ecx]
        movd        xmm3, ecx
        mov         ebx, dword ptr [edi+eax-3*16+4]
        movzx       ebx, byte ptr [esi+ebx]
        pinsrw      xmm3, ebx, 2
        movq        xmm0, qword ptr [edi+eax-3*16+8]
        pshufd      xmm1, xmm3, R_SHUFFLE_D( 2, 0, 1, 1 )
        xor         ecx, ebx
        pshufd      xmm0, xmm0, R_SHUFFLE_D( 0, 1, 1, 0 )
        lea         ebx, [ecx*2+ecx]
        pxor        xmm0, xmm1
        add         ebx, edx
        movlps      qword ptr [edx*8+0*4], xmm0
        pxor        xmm3, xmm0
        movhps      qword ptr [edx*8+2*4], xmm0
        movlps      qword ptr [edx*8+4*4], xmm3

        mov         ecx, dword ptr [edi+eax-2*16+0]
```

```
        movzx       ecx, byte ptr [esi+ecx]
        movd        xmm2, ecx
        mov         edx, dword ptr [edi+eax-2*16+4]
        movzx       edx, byte ptr [esi+edx]
        pinsrw      xmm2, edx, 2
        movq        xmm0, qword ptr [edi+eax-2*16+8]
        pshufd      xmm1, xmm2, R_SHUFFLE_D( 2, 0, 1, 1 )
        xor         ecx, edx
        pshufd      xmm0, xmm0, R_SHUFFLE_D( 0, 1, 1, 0 )
        lea         edx, [ecx*2+ecx]
        pxor        xmm0, xmm1
        add         edx, ebx
        movlps      qword ptr [ebx*8+0*4], xmm0
        pxor        xmm2, xmm0
        movhps      qword ptr [ebx*8+2*4], xmm0
        movlps      qword ptr [ebx*8+4*4], xmm2

        mov         ecx, dword ptr [edi+eax-1*16+0]
        movzx       ecx, byte ptr [esi+ecx]
        movd        xmm3, ecx
        mov         ebx, dword ptr [edi+eax-1*16+4]
        movzx       ebx, byte ptr [esi+ebx]
        pinsrw      xmm3, ebx, 2
        movq        xmm0, qword ptr [edi+eax-1*16+8]
        pshufd      xmm1, xmm3, R_SHUFFLE_D( 2, 0, 1, 1 )
        xor         ecx, ebx
        pshufd      xmm0, xmm0, R_SHUFFLE_D( 0, 1, 1, 0 )
        lea         ebx, [ecx*2+ecx]
        pxor        xmm0, xmm1
        add         ebx, edx
        movlps      qword ptr [edx*8+0*4], xmm0
        pxor        xmm3, xmm0
        movhps      qword ptr [edx*8+2*4], xmm0
        add         eax, 4*16
        movlps      qword ptr [edx*8+4*4], xmm3

        jle         loop4

    run1:
        sub         eax, 4*16
        jge         done

    loop1:
        mov         ecx, dword ptr [edi+eax+0]
        movzx       ecx, byte ptr [esi+ecx]
        movd        xmm2, ecx
        mov         edx, dword ptr [edi+eax+4]
        movzx       edx, byte ptr [esi+edx]
        pinsrw      xmm2, edx, 2
        movq        xmm0, qword ptr [edi+eax+8]
        pshufd      xmm1, xmm2, R_SHUFFLE_D( 2, 0, 1, 1 )
        pshufd      xmm0, xmm0, R_SHUFFLE_D( 0, 1, 1, 0 )
        pxor        xmm0, xmm1
        movlps      qword ptr [ebx*8+0*4], xmm0
        movhps      qword ptr [ebx*8+2*4], xmm0
        pxor        xmm2, xmm0
        movlps      qword ptr [ebx*8+4*4], xmm2
        xor         ecx, edx
        lea         edx, [ecx*2+ecx]
        add         ebx, edx

        add         eax, 16
        jl          loop1

    done:
        shl         ebx, 3
        mov         num, ebx
        pop         ebx
    }

    return ( num - (int)shadowIndexes ) >> 2;
}
```

Appendix F

```
/*
    SSE Optimized Construction of Shadow Volume Cap Triangles
    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_INIT4( unsigned long SIMD_DW_capTris_c0, 0, 0, 0, 1 );
ALIGN4_INIT4( unsigned long SIMD_DW_capTris_c1, 1, 1, 0, 0 );
ALIGN4_INIT4( unsigned long SIMD_DW_capTris_c2, 0, 1, 0, 0 );

int CreateCapTriangles( int *shadowIndices, const byte *facing, const int *indices, const int numIndices ) {
    int num = numIndexes / 3;

    __asm {
        push        ebx
        mov         eax, numIndexes
        mov         ebx, shadowIndexes
        mov         esi, facing
        mov         edi, indexes
        shl         eax, 2
        jz          done
        add         edi, eax
        mov         eax, num
        add         esi, eax
        neg         eax
        shr         ebx, 3

        movaps      xmm6, SIMD_DW_capTris_c0
        movaps      xmm7, SIMD_DW_capTris_c1
        movaps      xmm5, SIMD_DW_capTris_c2

        add         eax, 4
        lea         edx, [eax*2+eax]
        jg          run1

    loop4:
        movdqa      xmm0, [edi+edx*4-4*3*4+0]                   // xmm0 =  0,  1,  2,  3
        paddd       xmm0, xmm0
        pshufd      xmm1, xmm0, R_SHUFFLE_D( 2, 1, 0, 0 )       // xmm1 =  2,  1,  0,  0
        movzx       ecx, byte ptr [esi+eax-4]
        pshufd      xmm2, xmm0, R_SHUFFLE_D( 1, 2, 1, 2 )       // xmm2 =  1,  2,  1,  2
        sub         ecx, 1
        pxor        xmm1, xmm6
        and         ecx, 3
        movlps      qword ptr [ebx*8+0*4], xmm1
        add         ecx, ebx
        movhps      qword ptr [ebx*8+2*4], xmm1
        pxor        xmm2, xmm7
        movlps      qword ptr [ebx*8+4*4], xmm2

        movdqa      xmm3, [edi+edx*4-3*3*4+4]                   // xmm3 =  4,  5,  6,  7
        paddd       xmm3, xmm3
        shufps      xmm0, xmm3, R_SHUFFLE_D( 3, 3, 1, 0 )       // xmm0 =  3   3,  5,  4
        movzx       ebx, byte ptr [esi+eax-3]
        movdqa      xmm2, xmm3                                  // xmm2 =  4,  5,  6,  7
        sub         ebx, 1
        pxor        xmm0, xmm5
        and         ebx, 3
        movhps      qword ptr [ecx*8+0*4], xmm0
        add         ebx, ecx
        movlps      qword ptr [ecx*8+2*4], xmm0
        pxor        xmm2, xmm7
        movlps      qword ptr [ecx*8+4*4], xmm2

        movdqa      xmm0, [edi+edx*4-1*3*4-4]                   // xmm0 =  8,  9, 10, 11
        paddd       xmm0, xmm0
        shufps      xmm3, xmm0, R_SHUFFLE_D( 2, 3, 0, 1 )       // xmm3 =  6,  7,  8,  9
        pshufd      xmm1, xmm3, R_SHUFFLE_D( 2, 1, 0, 0 )       // xmm1 =  8,  7,  6,  6
        movzx       ecx, byte ptr [esi+eax-2]
        pshufd      xmm2, xmm3, R_SHUFFLE_D( 1, 2, 1, 2 )       // xmm2 =  7,  8,  7,  8
        sub         ecx, 1
        pxor        xmm1, xmm6
        and         ecx, 3
        movlps      qword ptr [ebx*8+0*4], xmm1
        add         ecx, ebx
```

```
        movhps      qword ptr [ebx*8+2*4], xmm1
        pxor        xmm2, xmm7
        movlps      qword ptr [ebx*8+4*4], xmm2

        pshufd      xmm1, xmm0, R_SHUFFLE_D( 3, 2, 1, 1 )
        movzx       ebx, byte ptr [esi+eax-1]
        pshufd      xmm2, xmm0, R_SHUFFLE_D( 2, 3, 2, 3 )
        sub         ebx, 1
        pxor        xmm1, xmm6
        and         ebx, 3
        movlps      qword ptr [ecx*8+0*4], xmm1
        add         ebx, ecx
        movhps      qword ptr [ecx*8+2*4], xmm1
        pxor        xmm2, xmm7
        movlps      qword ptr [ecx*8+4*4], xmm2

        add         edx, 3*4
        add         eax, 4
        jle         loop4

    run1:
        sub         eax, 4
        jge         done

    loop1:
        lea         edx, [eax*2+eax]
        movdqu      xmm0, [edi+edx*4+0]
        paddd       xmm0, xmm0
        pshufd      xmm1, xmm0, R_SHUFFLE_D( 2, 1, 0, 0 )
        pshufd      xmm2, xmm0, R_SHUFFLE_D( 1, 2, 1, 2 )
        pxor        xmm1, xmm6
        movlps      qword ptr [ebx*8+0*4], xmm1
        pxor        xmm2, xmm7
        movhps      qword ptr [ebx*8+2*4], xmm1
        movzx       ecx, byte ptr [esi+eax]
        movlps      qword ptr [ebx*8+4*4], xmm2
        sub         ecx, 1
        and         ecx, 3
        add         ebx, ecx

        add         eax, 1
        jl          loop1

    done:
        shl         ebx, 3
        mov         num, ebx
        pop         ebx
    }

    return ( num - (int)shadowIndexes ) >> 2;
}
```