Occlusion Culling and Z-Fail for Soft Shadow Volume Algorithms

Ulf Assarsson Tomas Akenine-Möller

Chalmers University of Technology
Hörsalsvägen 11
412 63 Gothenburg
Sweden

Abstract

This paper presents a significant improvement of our previously proposed soft shadow volume algorithm for simulating soft shadows. By restructuring the algorithm, we can considerably simplify the computations, introduce efficient occlusion culling with speedups of 3-4 times, thus approaching real-time performance, and also generalize the algorithm to produce correct shadows even when the eye is inside a shadowed region (using z-fail). We present and evaluate a three pass implementation of the restructured algorithm for near real-time rendering of soft shadows on a computer with a commodity graphics accelerator. However, preferably the rendering of the wedges should be implemented in hardware, and for this we suggest and evaluate a single pass algorithm.

CR Categories: I.3.7 [Computer Graphics]Three-Dimensional Graphics and Realism.

Keywords: soft shadows, graphics hardware, shadow volumes.

1 Introduction

Rendering realistic shadows in real time is highly desirable, both for increasing the level of realism, and because shadows give important spatial clues. For real-time purposes, it is common to approximate all light sources as point lights, i.e., with an infinitely small extension. This gives rise to so-called hard shadows, where the transition from no shadow to full shadow is instant. However, in reality, all light sources have some extension (area or volume), which gives a smooth transition, called the penumbra region, from no shadow to full shadow, called the umbra region. Several soft shadow algorithms exist, but most of them suffer from either 1) not being suitable for real-time rendering, or 2) only being able to handle planar shadow receivers, or 3) suffer from sampling artifacts. Our recently presented penumbra wedge algorithm [1] can handle all of the following goals:

I. The softness of the penumbra should increase linearly with distance from the occluder, starting at zero at the occluder [12].

II. The umbra region should diminish in size with increasing light source size.

III. Typical sampling artifacts should be avoided. Often a number of super-positioned hard shadows can be discerned [13]. The result should be visually smooth [12].

IV. The algorithm should be amenable for hardware implementation giving real-time performance (and interactive rates for a software implementation).

V. It should be possible to cast soft shadows on arbitrary surfaces, and work for dynamic scenes as well.

Our penumbra wedge algorithm is based on Crow's *shadow volume* (SV) algorithm [5], described in section 2. We do not require that the soft shadows are totally physically correct, but rather they should be perceptually pleasing without obvious artifacts. Although the algorithm is mainly targeted for spherical or circular light sources, it can approximate the soft shadow generated by any convex light source.

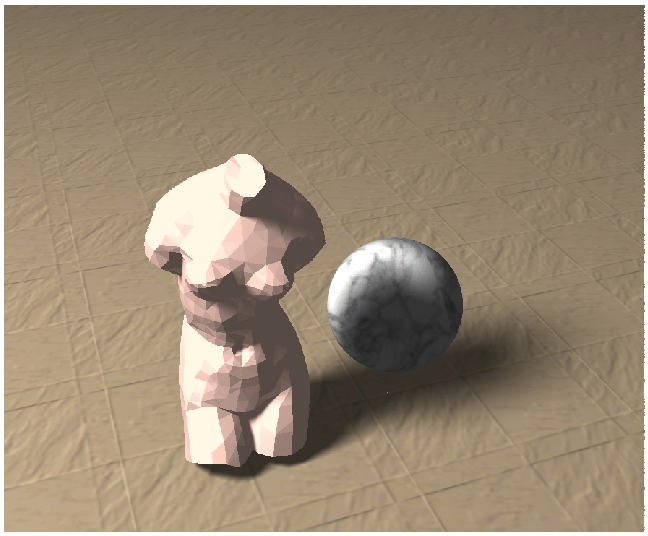


Figure 1: This image of Venus was rendered using the three pass algorithm (see Section 6.1) in 1 fps using a P4 1700MHz and a GeForce3 graphics accelerator. The Venus model casts soft shadows onto itself, the sphere and the floor. The image size is 640×427 pixels.

The algorithm still suffers from problems when automatically generating wedges from silhouette edges that are nearly parallel with the direction from the edge vertices to the light position. Artifacts can also appear if wedges incorrectly are generated for silhouette edges that are inside shadow. Furthermore, artifacts may appear when the light source is so large that there is no umbra region at all. Still, we strongly believe that the penumbra wedge algorithm is an important step in the right direction towards real-time soft shadows, because it is likely that those problems can be solved with a new light intensity interpolation method inside the wedges.

Therefore, we present some speedup techniques that gives near real-time performance, and generalizations to our previously presented penumbra wedge algorithm. In particular, we examine the algorithm from a hardware implementation perspective.

The contributions of this paper are as follows; 1) We present a restructured version of our original algorithm that significantly reduces the number of calculations to rasterize a wedge. 2) A method for very efficient occlusion culling, made possible by the restructuring, is presented, and it gives general speedups of 3-4 times for our test scenes. 3) We show

how the restructuring also enables the use of the *z-fail* algorithm [4] to correctly handle the case when the eye is inside a shadow region. Neither occlusion culling nor the z-fail algorithm can be incorporated in any obvious way in the originally proposed algorithm without the restructuring. 4) We suggest two different implementations of the restructured algorithm; a single pass algorithm for a possible hardware implementation of the wedge rasterization, and a three pass algorithm when no special hardware support of rasterizing the wedges is available. Furthermore, we evaluate software implementations of the two algorithms and present figures for the number of memory access used and frame rates.

The contributions of this paper are independent of the type of wedge construction being used, and what kind of light intensity interpolation that is done inside the penumbra wedges. Therefore, we strongly believe that the results in this paper applies for future improvements of the algorithm that may overcome the remaining problems with artifacts.

The paper is organized as follows. In the next section, the soft shadow volume algorithm is reviewed. In Section 3, we describe how to restructure the algorithm to reduce the number of calculations needed to rasterize a wedge. Section 4 introduces efficient occlusion culling, and then in Section 5 follows a generalization that correctly handles the case when the eye is inside shadow. In Section 6, two implementations are presented that suits software rasterization and hardware rasterization respectively, and that uses a different number of rendering passes. Section 7 gives the experimental results for the two implementations, and the paper ends with discussion, future work and a conclusion.

For related work, see our previously published paper [1]. More thorough presentations are presented by Woo et al [14] or Haines and Möller [9].

2 Review of the Soft Shadow Volume Algorithm

In 1977, Crow presented his shadow volume (SV) algorithm for hard shadows [5]. Heidmann extended the algorithm, in 1991, with hardware acceleration using the stencil buffer [10]. For each shadow casting object, its shadow volume is created. The shadow volume is created in the following manner. Each silhouette edge, as seen from the light position, and rays from the edge's two vertices in the direction from the light source forms a quadrilateral (quad). Together, all quads represent the shadow volume (see Figure 2). First the scene is rendered from the eye with only ambient light enabled. Secondly, all front facing quads, as seen from the eye, of the shadow volumes are rendered to the stencil buffer, incrementing each rasterized pixel that passes the depth test. Each pixel in the stencil buffer has now recorded the number of times a virtual ray from the eye through the pixel to the point represented by its z-value, enters a shadow region. Then, all the back facing quads are rendered, counting the number of times the virtual rays exits the shadow regions. Afterwards, if the stencil value for a pixel is larger than zero, the point is in shadow. That is, the virtual ray from the eye to the point enters shadow regions more times than it exits shadow regions on its way from the eye to the point. This algorithm has to be modified if the eye is inside a shadow volume (see Section 5). Finally, the stencil buffer is used as a mask when rendering the specular and diffuse contribution. For the stencil passes, the depth test is set to accept objects closer than the stored value, as usual, but no new depth values are written to the depth buffer and no color values are written to the frame buffer.

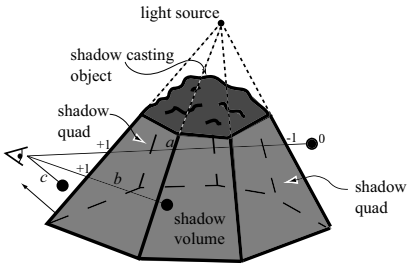


Figure 2: The standard shadow volume algorithm. The shadow volume here consists of seven quads. Ray b is inside shadow with a stencil value of 1. Ray a and c are outside shadow with stencil values of 0.

Our soft shadow algorithm [1] is based on Crows's SV algorithm in the sense that it also uses shadow volumes and a stencil buffer. However, instead of an instant transition from no shadow to full shadow, given by the quads, those are replaced by penumbra wedges, where the light intensity (LI) varies linearly inside the wedge (see Figure 5 and Figure 3). The wedge is rasterized into a 16-bit stencil buffer, which we call the light intensity (LI) buffer.

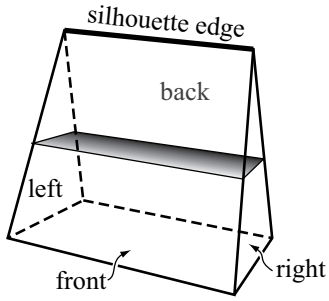


Figure 3: A penumbra wedge with its light intensity interpolation inside.

The light intensity buffer originally contains 255, and this represents fully lit pixels. The front facing planes of the wedges are rasterized, and as the scene content at the pixels, read from the z-buffer, is found in penumbrae, the rendering of the wedges dims down (subtracts) from this 255 level, and full umbrae subtract the full 255, due to the surface location being behind a wedge.

The penumbra wedges can be thought of as a new volumetric primitive, conceptually rasterized as outlined below:

```
1 :  rasterizeWedge()
2 :  foreach pixel(x, y) on front facing tris of wedge
3 :     p_f = computeEntryPointOnWedge(x, y);
4 :     p_b = computeExitPointOnWedge(x, y);
5 :     p = point(x, y, z);  −  z is depth buffer value
6 :     p_i = choosePointClosestToEye(p, p_b);
7 :     s_f = computeLightIntensity(p_f);
8 :     s_i = computeLightIntensity(p_i);
9 :     addToLIBuffer(round(255 * (s_i − s_f)));
10 :  end;
```

$\mathbf{p}_f$ is the point on the wedge where a ray from the eye through the pixel (x, y) enters the wedge. This disregards the case when the eye is inside the wedge, which we handle in section 5. $\mathbf{p}_i$ is the point stored at the pixel position in the z-buffer, if that point is inside the wedge. Otherwise it is the point where the ray exits the wedge (see Figure 4).

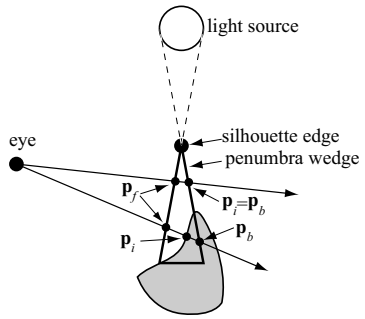


Figure 4: Illustration of the $\mathbf{p}_f$, $\mathbf{p}_b$, and $\mathbf{p}_i$ values for two rays.

The light intensity is represented with a value from 0 to 255, which makes the precision demands higher on the stencil buffer than for the SV algorithm in order to avoid overflowing when a virtual ray passes several shadow regions. Given a 16-bit signed stencil buffer, we can guarantee that at least 127 shadow volumes can have overlapping regions without causing overflow in the LI buffer. This is the same type of restriction as for the SV algorithm with an 8-bit stencil buffer.

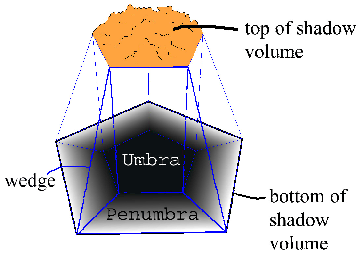


Figure 5: Example of a shadow volume for five silhouette edges. Each wedge is outlined.

The soft shadow algorithm works as follows: First all the geometry of the scene is rendered into the frame and depth buffer. with only specular and diffuse lighting enabled. Secondly, the LI buffer is cleared to 255, implying that everything in the scene is outside shadow. Then all shadow volume wedges are rasterized to the LI buffer with ordinary depth testing enabled, but without writing new z-values. In this way light intensity values will be written into the LI buffer. After this, the LI buffer is used to modulate the color intensity of each pixel in the frame buffer. Finally, the ambient contribution is added in a separate rendering pass. If the ambient contribution is rendered first into the frame buffer, as for the SV algorithm, we would have to multiply the diffuse and specular contribution in the following pass with the content in the LI buffer at the corresponding pixels before adding it to the frame buffer. In current hardware it is easier to instead multiply the whole frame buffer with the LI buffer between a first diffuse and specular rendering pass and postpone the ambient pass. The SV algorithm uses the stencil buffer as a binary mask, but the LI buffer holds 16-bit weights.

3 A Restructured Soft Shadow Algorithm

To compute $\mathbf{p}_f$ and $\mathbf{p}_b$, our original algorithm calculates the intersections between the four wedge planes (front, back, left and right) and the ray through the pixel, and finds the closest and furthest intersection points. This requires 4 divisions per pixel. There are ways to avoid at least two of these divisions, since we basically only are interested in finding the closest and the furthest point, but it is a bit messy and requires testing the signs of the numerator and the denominator with corresponding if-statements. If-statements are often undesirable, since they may cause branch-predict misses. However, in this section we will show how to avoid computing $\mathbf{p}_f$ and $\mathbf{p}_b$ at all.

Previously, we observed that the contribution of the left and right planes always cancels out with the neighboring wedges [1]. Now, we will further reduce the computations needed, and also restructure the algorithm to enable efficient occlusion culling and correctly handle the case when the eye is inside a shadow region (see Section 4 and 5).

In our implementation we have chosen 255 to represent full light and 0 to represent full shadow. This means that the back plane of the wedge will only contribute with entry or exit intensity values of 0, and their rasterization can thus be skipped. We could arbitrarily have chosen 0 to represent full light and 255 as full shadow, so that the front planes could be ignored instead. This could possibly have the advantage that fewer pixels need to be rasterized, since normally in a closed soft shadow volume, the total area of the back planes is smaller that that of the front planes.

Using the fact that the rasterization of the back planes can be skipped, the algorithm can be restructured as follows:

```
1 : rasterizeWedge()
2 :    rasterizeUmbra(frontplane, −255);
3 :    rasterizePenumbra(all front facing planes);
4 : end
5 :
6 : rasterizeUmbra(primitive, value)
7 : for each pixel(x, y) of primitive
8 :    if primitive is front facing
9 :       addToLIbuffer(value);
10 :   else addToLIbuffer(−value);
11 : end
12 : rasterizePenumbra(primitive)
13 : for each pixel(x, y) of primitive
14 :    p = point(x, y, z); − z is depth buffer value
15 :    if p is inside the wedge
16 :       s_p = computeLightIntensity(p);
17 :       addToLIBuffer(round(255 ∗ s_p));
18 : end;
```

rasterizeUmbra() is very similar to the SV algorithm, but is using the front planes of the wedges to define the shadow volumes and adds or subtracts 255 instead of 1. **rasterizePenumbra**() computes light intensities, s_p, for all pixels inside the wedges, and adds light contribution between 0 and 255 to the penumbra regions. This is done for all wedges by rasterizing all front facing triangles of the wedge and adding an interpolated intensity value for all pixels with corresponding depth buffer points located inside the wedge. The test if a point (x, y, z) is inside the wedge is done in screen-space to avoid transforming the point to world space with a full matrix multiplication, which would include a division of the w-component. The screen-space wedge-plane equations are precomputed once each frame per wedge.

For **rasterizeUmbra**(), depth testing is enabled but without writing new z-values. For **rasterizePenumbra**(), no depth test is needed. Instead we can use the occlusion culling described in Section 4.

With this restructured algorithm, no intersection points need to be computed at all, and all the corresponding divisions are eliminated. There is still one division required in the linear interpolation and one division required for transforming **p** to world-space, when computing the light intensity (line 16) [1]. However, these are done only for points located inside the wedge, that is, where penumbra is present.

4 Occlusion Culling

The function **rasterizePenumbra**() affects only the pixels with points located inside the wedge. With our restructured algorithm, very efficient occlusion culling can be implemented.

Normally, hardware occlusion culling avoids rendering for pixel tiles where the object to be rendered is behind everything that is stored in the z-buffer positions for the tile. This can be done in hardware by storing the maximum z-value, z_{max}, for each tile [11]. A common tile size is 8×8 pixels.

In our method, we cull rendering of the penumbra for pixel tiles that are totally *behind* or totally *in front* of the wedge. For this, we need to store both the z_{max} and the z_{min} for each tile, where z_{min} is the minimum z-value for the tile. With this occlusion culling, only tiles that intersect the wedge will be rasterized, resulting in significant speedup.

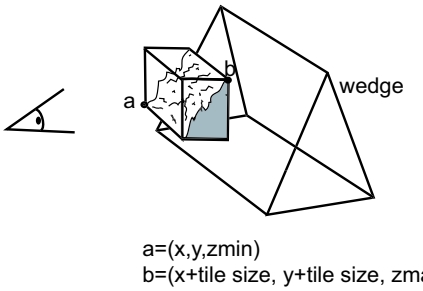


Figure 6: The screen space bounding box of a tile is tested for intersection with the wedge. Only if they intersect is the tile rasterized for the penumbra contribution. Here, the box contains a piece of a fractal mountain (see Figure 12). Since the box does not intersect the wedge, in this example, the tile will be culled from penumbra rasterization.

Before rasterizing a wedge, its screen space plane equations are precomputed in a setup routine. Upon rasterizing the wedge triangles, the screen space axis aligned bounding box of a tile that is about to be rasterized is tested for intersection with the wedge (see Figure 6). If the bounding box is outside the wedge, the whole tile is culled. The Separating Axis Theorem can be used to determine whether they overlap [6]. The theorem states that for two convex, disjoint polyhedra, A and B, there exists a separating axis where the projections of the polyhedra also are disjoint. Furthermore, it states that it is sufficient to test only the axes that are orthogonal (i.e., the planes with its normal orthogonal) to a face of A or B, or an edge from each polyhedron. If such an axis cannot be found, we know the box and the wedge are overlapping. Testing all the axes is often unnecessarily time consuming. It is usually better to only do the tests corresponding to the faces of A or B, and ignore the tests corresponding to the edges of the polyhedra. This may sometimes give incorrect indication of overlap, but that will only force the tile to be rasterized with occlusion tests for each pixel, and causes no visual error. The advantage is that the tile overlap test will be significantly faster.

The test is done by inserting the vertices of the bounding box of the tile into the wedge plane equations in screen space. If all vertices are outside any of the wedge planes, the box is outside the wedge. If all vertices are inside all wedge planes, the box is fully inside. Otherwise, we consider the box as intersecting, although there are circumstances where the box can be outside. To avoid some of these occasions, testing of the wedge vertices against the box planes could be added. Notice, that only the screen-space z_{min} and z_{max} of the wedge need to be tested against the z_{min} and z_{max} of the tile, since the wedge and box must intersect at the x- and y-coordinates due to the rasterization. Since this latter test is a so called quick rejection test and can cull the region with just one simple test, it should be done first of all tests if it is being used. The wedge's z_{min} and z_{max} are computed in a setup routine before rasterizing the wedge.

When testing the box vertices against a wedge plane, it is sufficient to test only the closest and the furthest of the vertices instead of all eight, which saves many computations [7, 8]. The two vertices are easily recognized by the signs of the x, y and z components of the normal of the wedge plane (see Figure 7).

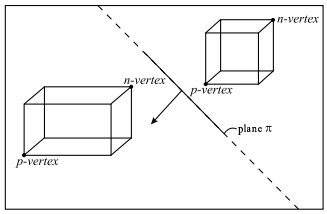


Figure 7: This figure illustrates the n and p vertices of two boxes with respect to a plane.

In our software implementation, the occlusion culling test only takes about 1.5% of the total execution time. We investigated different combinations of tests. The test of the box vertices against the wedge planes seem to be the most important. When the test of the wedge's z_{min} and z_{max} against the region z_{min} and z_{max} is included, there is hardly any noticeable increase in performance. If only this latter test is used, the performance drops significantly. For the scenes that we have tested, the occlusion culling generally provides a speedup of 3-4 times for the software penumbra rasterization. This results in an up to three-fold overall performance enhancement of the frame rate.

It should be noted that this occlusion culling for the penumbra rendering does not require that the rendering is done in any depth order to reach optimal results. This is opposed to occlusion culling for ordinary object rendering to the frame buffer, where the objects should be rendered in front-to-back order for maximum efficiency. The reason is that the penumbra rendering does not affect the depth buffer. It only uses the content of the depth buffer from the ordinary rendering of the scene, to apply soft shadows to the image. Also, therefore z_{min} and z_{max} need not be written to during wedge rasterization.

5 Eye Inside Shadow Regions

The original SV algorithm [5], does not properly handle the case when the eye is inside a shadow volume, and the same applies to our original algorithm. An elegant solution for the SV algorithm was officially documented by Everitt and Kilgard [4] in 2002, although the algorithm had been known to the gaming industry a while through Bill Bilodeau and Mike Songy (1999) and John Carmack (2000) [4]. Instead of

determining if a point is in shadow by testing intersections with the shadow volumes of a virtual ray from the eye to the point, a virtual ray from the point to the infinity could be tested. In this way, the testing will be independent of the eye position. In practice, this is achieved by modifying the stencil buffer passes. The first stencil buffer pass becomes: render all back facing shadow volume polygons and increment the stencil value when the polygon is equal to or farther than the stored z-depth. In the second stencil pass, all front facing shadow volume polygons are rendered, decrementing the stencil value when the polygon is equal to or farther than the stored z-depth. This algorithm is often called the z-fail algorithm [4].

Since the depth test has been altered, it is now also necessary to render the *top* of the shadow volume. The top consists of all polygons of the shadow generator that are front facing with respect to the light source center. If the shadow volume is of finite length, the volume must be closed at the *bottom*, by for instance adding a far capping polygon. The top and the bottom polygons should be rendered in exactly the same way as the shadow quads. Problems can still occur if the polygons are clipped by the far plane of the view frustum. This could, however, be solved by setting the far plane to infinity or using extensions in the rasterizing hardware. Such an extension is included in NVIDIA's GeForce3 [4].

The solution for the SV algorithm described above can be applied to our soft shadow algorithm as well, with just some minor additions. If the wedge planes are of finite length, a capping *bottom plane* of the wedge must be added too, and rasterized by **rasterizePenumbra**() when it is back facing. We use the front planes of the wedges as the shadow volume for the umbra contribution so that should be capped with top and bottom polygons. In **rasterizeUmbra**(), the front planes of all wedges plus the top and bottom polygons of the shadow volume are rasterized. The pseudo code for rasterizing a soft shadow volume now becomes:

```
1 : renderShadowVolume()
2 :     rasterizeUmbra(top + bottom polygons, 255);
3 :     for all wedges
4 :         rasterizeWedge()
5 : end
6 : rasterizeWedge()
7 :     rasterizeUmbra(frontplane, 255);
8 :     rasterizePenumbra(all back facing planes);
9 : end
```

The reason the back facing planes are chosen in **rasterizeUmbra**() is that if the eye is inside a wedge, none of the wedge's planes are front facing with respect to the eye position. Notice that since we are using the z-fail algorithm, **rasterizeUmbra**() is now called with a value of 255 instead of -255 (compare the listing in Section 3, line 2).

6 Implementation

In this section, we present two different implementations of the restructured algorithm. The first one, which we call the *three pass* algorithm, is most suitable when wedge rendering has to be done without special hardware support, but when a commodity graphics accelerator is available. It splits the wedge rendering into three passes, and uses commodity graphics hardware to render the umbra contribution (two passes), and software to render the penumbra contribution (one pass). The second implementation, which we call the *single pass* algorithm, is meant to be implemented as a new mechanism in graphics cards. It tries to minimize the number of memory accesses by rasterizing the umbra and penumbra contribution simultaneously when possible.

6.1 Three Pass Algorithm

To rasterize the penumbra wedges efficiently, without full hardware support, we suggest a three pass algorithm. First, common hardware is used to render the front planes, and then software is used to render the inside of the penumbra wedges. With occlusion culling for the software rendering, real-time performance can be achieved (see Section 7). The occlusion culling is very effective, since the contents of the z-buffer does not change when rendering the wedges, and only pixels potentially inside the wedges need to be considered (see Section 4).

Current graphics hardware normally do not have a 16-bit stencil buffer, but an 8-bit stencil buffer usually suffices for the rendering of the front planes. Initially, the 8-bit stencil buffer is cleared with a value of 0. In the first pass all front facing front planes of the wedges are rasterized, and for each time a pixel passes the depth-test, the corresponding stencil value is decremented by one. In the second pass all back facing front faces are rasterized similarly, but incrementing the stencil values by one instead. The buffer is then added to the 16-bit software light intensity (LI) buffer, which has been initialized with a value of 255, pre-multiplying each 8-bit stencil value with 255, or -255 if the z-fail algorithm is used. In the third pass, software rendering of the penumbra regions is done as outlined in the pseudo code for **rasterizePenumbra**() in section 3.

If the stencil buffer does not handle negative stencil values and clamps them to zero, an offset of, for instance, 128 could be used to circumvent the problem.

6.2 Single Pass Algorithm

If wedge rasterization is implemented in silicon in a new graphics hardware, it can be advantageous to minimize the number of memory accesses, since the memory bandwidth and latency often are the bottlenecks. For each pass and each pixel that is being rasterized, and is not culled by the occlusion culling (see Section 4), the z-value needs to be read from the z-buffer, and values are possibly written to the stencil buffer.

In the single pass algorithm, we want to rasterize the umbra and penumbra in the same pass. This is not obvious how to incorporate into our latest soft shadow volume algorithm [2, 3], since that uses the quads of the shadow volume for hard shadows to render the umbra. However, it is fairly easy for the original penumbra wedge algorithm [1], since this version rasterizes the umbra using the front planes of the wedges. To evaluate whether a single pass version has advantages over a three-pass version, we therefore use the latter in this case study.

The penumbra is rasterized by all front facing planes of the wedge, or all back facing planes of the wedge if the z-fail algorithm is being used. Thus, for front facing front planes, rasterization of the umbra and penumbra could be done in the same pass. This saves one stencil buffer write access and one z-value read access for each pixel of the front plane that will receive a contribution from both umbra and penumbra. The rasterization will be slightly more complex, since we want to use occlusion culling with z_{min} and z_{max} for the penumbra contribution, but only z_{max} for the umbra

contribution. In the z-fail algorithm, the back facing front planes should be rasterized in a single pass instead of the front facing front planes, and z_{min} should then be used for the occlusion culling for the umbra contribution.

The advantage is that the front plane of each wedge is rasterized only once, thus saving z-value read accesses and stencil value write accesses.

7 Simulation Results

In this section, experimental results for the three pass algorithm and the single pass algorithm are presented. All tests were done using software implementations., since the occlusion culling requires hardware support that is not yet present. The only hardware acceleration used was for rendering of the front planes for the umbra contribution in **rasterizeUmbra**() in the three pass algorithm. For this, we rendered quads into an 8-bit stencil buffer using a GeForce3 graphics accelerator.

We used the wedge rasterization method presented in section 3, since it is yet unclear how to create a single pass version for the rasterization method proposed in [2]. The latter rasterization is more expensive, and thus it is reasonable to conclude that occlusion culling can provide even higher speedups for that case.

The test scenes used are shown in Figure 11, 12, and 13. All the test scenes were rendered with an image size of 640×427 pixels. The rasterization is highly fill-rate limited.

The restructured algorithm reduces the number of calculations, compared to the original algorithm, and in itself contributed with a general speedup of $30-40\%$ (i.e., 1.3-1.4 times), without any occlusion culling. This was measured with the three pass algorithm, without using hardware acceleration for the umbra rasterization. If hardware acceleration is added, the speedup is $60-80\%$. The single pass algorithm is also $30-40\%$ faster than the original algorithm (see Figure 10a).

Next, the results using occlusion culling is presented. We investigated the performance with different tile sizes for the occlusion culling described in Section 4. Tiles of $2^n \times 2^n$ pixels with $n \in [1..8]$ where tested.

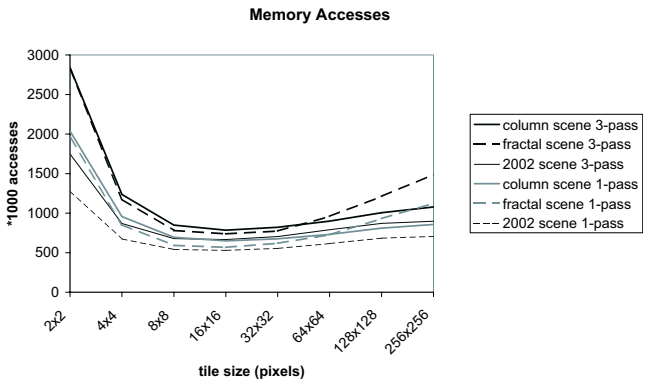


Figure 8: This graph shows the total number of z-buffer reads and stencil buffer writes for different tile sizes, when rendering one frame of the column scene, fractal scene, and 2002-scene. The results from using the three pass algorithm and the single pass algorithm are shown. The images can be seen in Figure 11, 12, and 13.

Figure 8 shows the total number of stencil buffer writes and z-buffer reads needed to render one frame of each test scene. As can be seen, the optimal tile size for minimizing

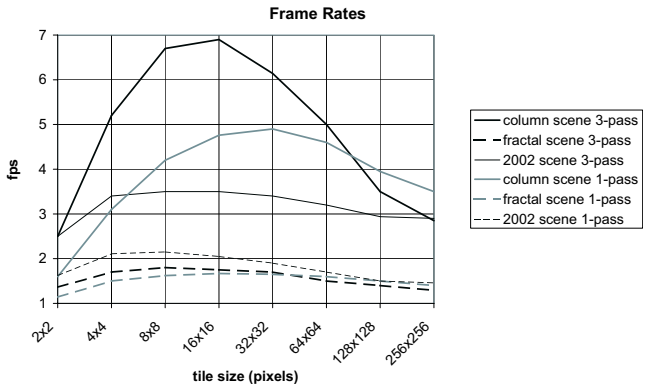
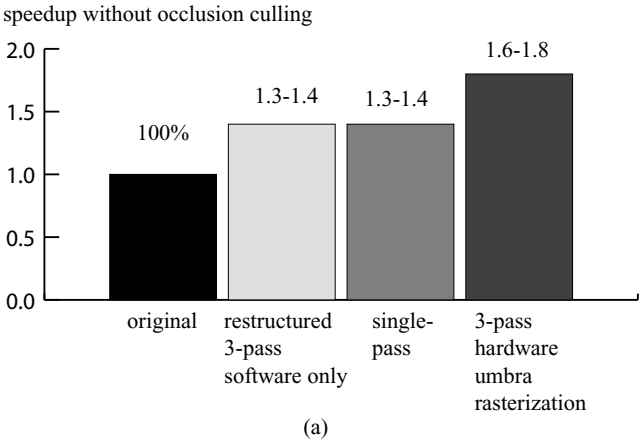


Figure 9: The graph shows the frame rates, using different tile sizes for rendering the column scene, fractal scene, and 2002-scene. Result from the single pass algorithm and the three pass algorithm are shown.



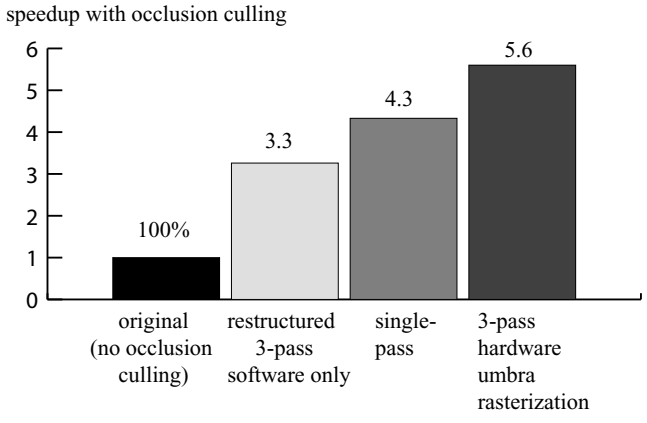


Figure 10: a) General speedups without using occlusion culling. b) Speedup using occlusion culling with the optimal tile size and using the test scene shown in Figure 11.

the number of memory accesses is from 8×8 to 32×32 pixels for these scenes. This correlates very well with the graph for the frame rates, in Figure 9. It means that the optimal tile size for minimizing memory accesses matches the optimal tile size for minimizing the rasterization work, which seems natural. As expected, the frame rates are higher for all scenes using the three pass algorithm, than the single pass algorithm, since we do not have any hardware implementation of the latter.

With occlusion culling, the single pass algorithm was generally $20-60\%$ faster than the three pass algorithm when software rasterization of the umbra regions was used for the latter (e.g. Figure 10b), i.e., when both algorithms were using only software rendering. This indicates that for a full hardware implementation of wedge rasterization, the extra complexity of a single pass algorithm could be worthwhile. Additionally, a hardware implementation could benefit even more from the savings in memory accesses, since these often are the bottlenecks. In the range of tile sizes from 8×8 to 32×32 pixels, the single pass algorithm used about 5% fewer stencil buffer write accesses and $30-60\%$ fewer depth buffer read accesses than the three pass algorithm. Together, this saves $20-30\%$ of the total number of memory accesses. The significant savings in execution time for the software implementation comes from not needing to rasterize a front facing front plane twice.

The occlusion culling described in section 4 generally gave a speedup of 3-4 times for the penumbra rasterization and using the optimal tile size, resulting in an up to three-fold overall frame rate improvement compared to not using occlusion culling. If only z_{max} is used, which is common in current hardware, and not z_{min}, then practically no speedup is obtained in any of the test scenes, since almost nothing in the scenes occludes the shadows. This is a strong argument for accepting the extra complexity of storing z_{min} in hardware as well. As an example of this, only using z_{max} lowers the frame rate from 7.0 frames per second (fps), to 2.6 fps, and increases the number of depth buffer read accesses with 50% for the three pass algorithm with hardware rendering of the umbra contribution, when rendering the column scene in Figure 11. The number of stencil buffer write accesses is unaffected by occlusion culling, since the occlusion culling only avoids unnecessary rasterization. If a tile is rasterized although it correctly could be culled, the corresponding points in the depth buffer are tested whether they are inside the wedge. Since all points will be found outside, no stencil buffer values will be written for this tile.

In total, the three pass algorithm with occlusion culling, is up to almost six times faster for the column scene, in Figure 11, than the original algorithm. Our conclusion for the single pass algorithm is that the added complexity of rendering the umbra and penumbra contributions in the same pass definitely could pay off, for a hardware implementation. To get efficient occlusion culling both z_{min} and z_{max} should be used, and the optimal tile size is from 8×8 to 32×32 pixels.

All test results were done using a standard PC with an Intel P4 1.7 GHz, and a GeForce3 graphics card. For the Venus scene in Figure 1, we used a different interpolation technique that we have developed recently [2]. This modification computes the light intensity more exactly than the previously presented linear interpolation [1], and thus avoids several of the artifacts with the old method.

8 Discussion and Future Work

The modifications to the original penumbra wedge algorithm [1] introduced in this paper are independent of how the shapes of the wedges are computed and what kind of interpolation that is used inside the penumbra. Thus, we strongly believe that the results presented in this paper will apply even for future modifications of the algorithm that may overcome the remaining artifacts. In fact, we have already presented several such modifications [2, 3].

If the eye-space distances are stored in the z-buffer, instead of the eye-space distances divided by w, one division can be eliminated for computing the light intensity of a point inside the penumbra. Currently, the point is transformed from screen-space to world-space, requiring one division with the w-component. That could be avoided, leaving just one division, and that is for the linear interpolation (see Section 3).

In software it could possibly be preferable to only compute and store z_{min} and z_{max} on demand, i.e., for the regions with values that are accessed. However, we could not measure any significant change in performance of our implementation when trying this.

The need for a 16-bit stencil buffer to store the light intensities could probably be circumvented by using HILO textures. HILO textures contains two 16-bit components for each element, and is available in for instance GeForce3. One of the 16-bit components could perhaps be used as the LI buffer.

Possibly, z_{min} could be used for ordinary hardware triangle rasterization as well, to save z-buffer reads when rendering visible geometry to the frame buffer.

9 Conclusion

We have presented a restructured version of our original soft shadow volume algorithm, which significantly reduces the number of calculations to rasterize a wedge. We have also shown how to incorporate occlusion culling using both z_{min} and z_{max}, to get a substantial speedup of the rasterization and reach near real-time performance on a standard PC. Empirical results show that the optimal tile size is between 8×8 and 32×32 pixels for our three test scenes. Furthermore, the restructured algorithm allows the use of the z-fail algorithm to correctly handle the case when the eye is inside a shadowed region. We want to emphasize that neither occlusion culling nor the z-fail algorithm can be incorporated in any natural way with the original penumbra wedge algorithm, presented in [1]. Thus, the restructuring presented in this paper significantly enhances the efficiency and usability of penumbra wedges for simulating soft shadows.

Two different implementations are presented and evaluated; a single pass algorithm for a possible hardware implementation of the wedge rasterization, and a three pass algorithm when only commodity graphics hardware is available. The single pass algorithm uses $20-30\%$ fewer memory accesses than the latter, to rasterize the wedges. This could be important for a hardware implementation, since the memory bandwidth and latency often are the bottlenecks. With the improvements presented in this paper, we believe that we have taken an important step forward for rendering soft shadows in real time.

References

[1] T. Akenine-Möller, U. Assarsson (2002) Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. 13th Eurographics Workshop on Rendering, Eurographics, 309-318.

[2] Assarsson, Ulf, and Tomas Akenine-Möller, "A Geometry-Based Soft Shadow Volume Algorithm Using Graphics Hardware," *Proceedings of ACM SIGGRAPH 2003*, Pages 511–520, 2003.

[3] Assarsson, Ulf, Michael Dougherty, Michael Mounier, and Tomas Akenine-Mller, "An Optimized Soft Shadow Volume Algorithm with Real-Time Performance," *Graphics Hardware 2003*, ACM SIGGRAPH / Eurographics Workshop Proceedings, Pages 33–40, 2003.

[4] C. Everitt, M. J. Kilgard (2002) Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Published on-line at http://developer.nvidia.com/

[5] F. C. Crow (1977) Shadow Algorithms for Computer Graphics. SIGGRAPH '77 Proceedings, 242–248

[6] S. Gottschalk,M.C. Lin, and D. Manocha (1996) OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. Computer Graphics (SIGGRAPH Proceedings '96), 171–180

[7] N. Greene (1994) Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. In: P. S. Heckbert (1994) Graphics Gems IV, pp. 74–82

[8] E. A. Haines, and J. R. Wallace (1994) Shaft Culling for Efficient Ray-Traced Radiosity. Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering), Springer-Verlag, New York, 122–138, also in SIGGRAPH '91 Frontiers in Rendering course notes

[9] E. Haines, and T. Möller (2001) Real-Time Shadows. Game Developers Conference

[10] T. Heidmann, (1991) Real shadows, real time. Iris Universe, Silicon Graphics Inc., No. 18, 23–31

[11] S. Morein (2000) ATI Radeon—HyperZ Technology. SIGGRAPH/Eurographics Graphics Hardware Workshop 2000, Hot3D session

[12] S. Parker, P. Shirley, and B. Smits (1999) Single Sample Soft Shadows. TR UUCS-98-019, Computer Science Department, University of Utah

[13] C. Soler, and F. X. Sillion (1998) Fast Calculation of Soft Shadow Textures Using Convolution. SIGGRAPH '98 Proceedings, 321–332

[14] A. Woo, P. Poulin, and A. Fournier (1990) A Survey of Shadow Algorithms. IEEE Computer Graphics and Applications, 10(6), 13–32

Author Biography

Ulf Assarsson received a M.Sc. degree in engineering physics from Chalmers University of Technology in 1997. Since 1998 he is a Ph.D student in Computer Graphics at the Department of Computer Engineering at Chalmers. His research interests include realistic real-time rendering, and he is currently focusing on real-time soft shadows.

Tomas Akenine-Möller is an assistant professor at the Department of Computer Engineering at Chalmers University of Technology, Sweden. He has received an MSc in Computer Science and Engineering from Lund University of Technology, and a PhD in Computer Graphics at Chalmers University. He is the coauthor with Eric Haines of the book "Real-Time Rendering", and his main research interests are rapid and realistic real-time rendering, interactive ray tracing, spatial data structures, and algorithms for future graphics hardware.

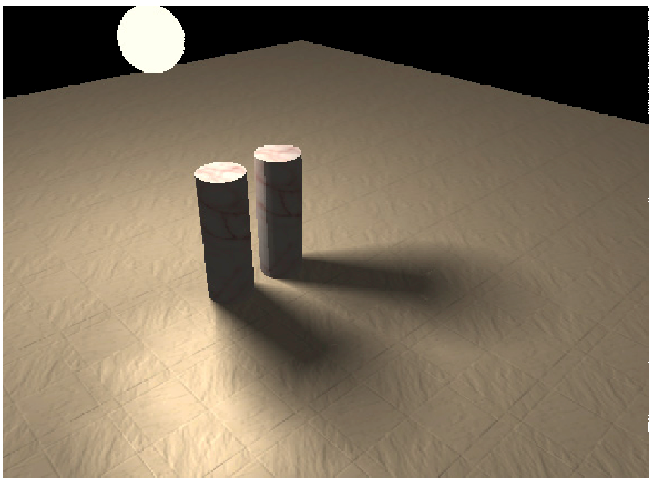


Figure 11: This image was rendered with the three pass algorithm in 7 fps on a Pentium4 1700 MHz using software and a GeForce3 graphics accelerator for the penumbra wedge rendering. The image size is 640×427 pixels.

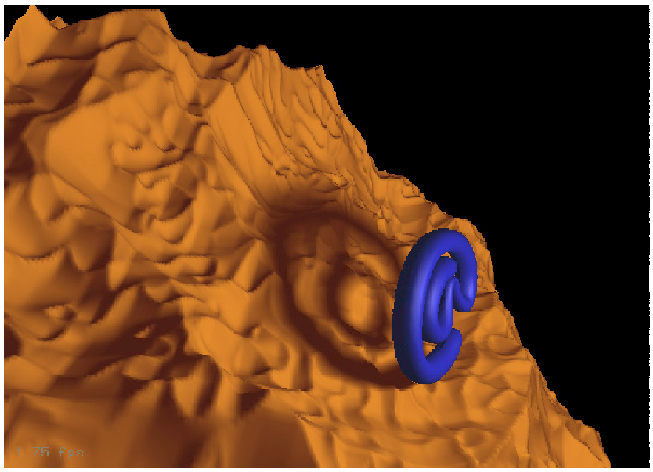


Figure 12: These images show a fractal landscape with 100k triangles used as a complex shadow receiver. The scene was rendered in 2.3 fps without soft shadows, and in 1.75 fps with soft shadows using the three pass algorithm.

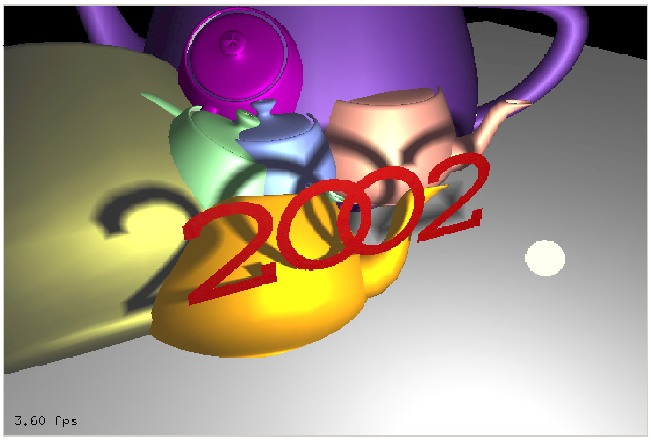


Figure 13: In this scene, the 2002 text casts its soft shadow onto a number of teapots and a floor. The image was rendered in 3.6 fps with the three pass algorithm.

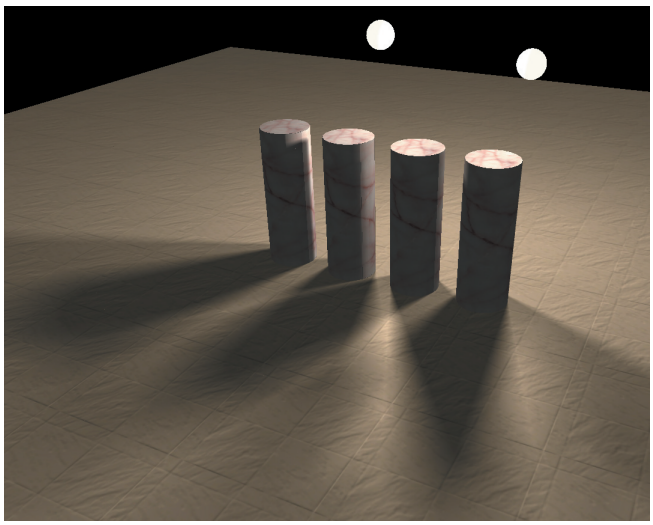


Figure 14: This is an example of using multiple light sources.